PhD-FSTC-2014-26
The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 08/09/2014 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

Alexandre Bartel
Born on 22 November 1986 in Strasbourg (France)

Security Analysis of Permission-Based Systems using Static Analysis: An Application to the Android Stack

Dissertation defense committee

Dr. Yves Le Traon, Dissertation supervisor
Professor, Université du Luxembourg

Dr. Lionel Briand, Chairman
Professor, Université du Luxembourg

Dr. Martin Monperrus, Deputy Chairman
Professor, Université de Lille

Dr. Andreas Zeller, Member
Professor, Universität des Saarlandes

Dr. Benjamin Livshits, Member
Professor, Microsoft Research

Dr. Eric Bodden, Invited Member
Professor, Technische Universität Darmstadt

Dr. Jacques Klein, Advisor
Université du Luxembourg

In memory of my father.
To my mother.
To my wife, Claire.

Abstract

In recent years, mobile devices, such as smart phones, have spread at an exponential rate. The most used system running on these devices, accounting for almost 80% of market share for smart phones world-wide, is the Android software stack. This system runs Android applications that users download from an application market. The system is called a permission-based system since it limits access to protected resources by checking that applications have the required permission(s). Users store and manipulate personal information such as contact lists or pictures using applications on their devices and trust that their data is safe. Analyzing applications and the system on top of which they are running would be an objective method to evaluate if the data is well-protected.

In this thesis we aim at analyzing Android applications from the security point of view and answering to the following challenging questions: How can Android applications be analyzed? Are permissions well-defined for Android applications? Can applications leak protected data? How can dynamic analysis complement static analysis? To answer these questions we structure the thesis around four objectives.

The first objective is to analyze Android applications with static analysis tools. The challenge is that Android applications are packaged with Dalvik bytecode, different in many aspects from the Java bytecode. We developed Dexpler, a tool to transform Dalvik bytecode into Jimple, an understandable format for Soot, one of the most used static analysis framework for Java-based programs. With Dexpler we can now analyze Android applications.

The second objective is to check that developers do not give too many permissions to the Android applications they develop. Reducing the number of permission reduces the attack surface of an malicious user exploiting an application. We analyze the code of applications to check which permissions they really require. This requires to deeply analyze the Android framework to extract a mapping between API methods (that Android application call) and required permissions. We present an Andersen-like field-sensitive approach using novel domain-specific optimizations to extract the mapping from the Android framework.

Permissions protect sensitive data. Nevertheless, applications having the right permission(s) to access the data could leak the data. This is for instance the case with malware or application packaged with aggressive advertisement libraries. The third objective is to statically analyze Android applications to detect such leaks. Android applications are different from traditional Java applications. One of the most important differences is that Android applications are made of components. Analyzing Android applications to find leaks requires to link components that communicate together and to model every component. We developed IccTA to detect privacy leaks. It connects components at the code level to perform inter-component and inter-application data-flow analysis.

Analyzing Android applications statically enables to find security issues such as the GPS coordinates leaking out of the device. However, static analyses do not run directly on users' devices and thus do not take the device's context into account. The last objective of this thesis is to have an insight of how dynamic approaches can complement static analyses. We are the first to present a tool-chain to dynamically instrument Android applications in vivo, i.e. directly on the device. We present two use cases instrumenting applications to show that dynamic approaches are feasible, that they can leverage results from static analyses, and that they are beneficial for the user from the point of view of security or privacy. One of the use case is a fine-grained permission system prototype enabling the user to disable or enable application permissions at will.

The four contributions have been validated through rigorous experiments as complete as possible.

Through this thesis we provide solutions to analyze Android applications using static analysis, to check the permission set of applications, to find private data leaks in Android applications and to analyze permission-based frameworks. By analyzing what goes wrong, we can improve the security and privacy of mobile applications.

Acknowledgements

First and foremost, I want to thank my supervisor, Yves Le Traon. It has been an honor to be his first Ph.D. student in his SERVAL team. I am grateful for the time he spent discussing with me on new ideas and contributions. When faced with a technical problem he insisted that I take a step back and look at the global picture, which really helped. Yves told me how good software engineering research is done.

Then, I would like to thank my day-to-day advisor, Jacques Klein. He is the one who initiated the first Android research projects by bringing Android devices onto my desk and steering me towards the right research questions. Always in a good mood, it was a real pleasure to work with him. He taught me to focus my thoughts and how to reason at an abstract level to clearly present new ideas.

Martin Monperrus was my remote advisor from the University of Lille. I would like to thank him for his constructive and insightful feedback on my work as well as great advices regarding paper writing and time management. Always motivated he pushed me to work hard to reach top-level conferences and journals.

During my Ph.D. I had very interesting technical discussions with numerous people from the University of Luxembourg, especially with Kevin Allix, Tegawendé François D'Assise Bissyande and Li Li, whom I would like to thank in particular for his great work on IccTA.

I had the opportunity to work with and visit researchers at the University of Pennsylvania in the USA and at the Technical University of Darmstadt in Germany. I would like to thank Damien Octeau and Patrick McDaniel from the University of Pennsylvania for initiating and leading the successful collaboration we had on EPICC. I also would like to thank Eric Bodden, Steven Arzt and Siegfried Rasthofer from the Technological University of Darmstadt for their amazing work on FlowDroid and our fruitful collaboration on taint analysis.

I would like to thank Lionel Briand, who did me the the honor of being the chairman of the dissertation defense. I thank the members of my oral dissertation defense committee, Andreas Zeller, Benjamin Livshits and Eric Bodden, for their time and insightful questions. For this dissertation I would also like to thank my readers, Jacques Klein, Yves Le Traon, Martin Monperrus, Li Li and Panuwat Trairatphisan for their time, interest, and helpful comments.

I would like to thank my Ph.D. candidate colleagues and friends who made my time enjoyable at the University of Luxembourg. I am grateful to Panuwat Trairatphisan, Wei Dou, Lamia Bekkour, Li Li, Phu-Hong Nguyen and Rustam Mazitov for memorable badminton and tennis games. I thank Kevin Allix for the interesting discussions we had on a very broad range of topics. More generally, I thank all people from the SERVAL team and SnT that I have had the chance to meet, in particular Christopher Henard, Donia El Kateb, Anestis Tsakmalis, Iram Rubab, Jabier Martinez, Thomas Hartmann, Jorge Meira, and Grégory Nain.

I gratefully acknowledge the National Research Fund of Luxembourg (FNR), my funding source that made my Ph.D. work possible during three years and a half.

Lastly, I would like to thank my family for their encouragements and love. For my parents who always supported me during my studies. And most of all for my loving supportive and encouraging wife Claire whose support during this Ph.D. has been much appreciated. Thank you.

Alexandre Bartel
University of Luxembourg
September 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, mobile devices, such as smart phones, have spread at an exponential rate. The most used system running on these devices, accounting for almost 80% of market share for smart-phones world-wide, is the Android software stackHowever, with popularity comes more attacks tailored for the Android system.

In this thesis we claim that static analyses can help to prevent such attacks. We use static analysis to extract permissions from a permission-based framework [1]. This enables to check that applications, running on top of the permission-based framework, adhere to the principle of least privilege and do not declare too many permissions. Moreover, to evaluate if resources and user data are well-protected, we statically analyze applications to find suspicious leaks that could indicate a malicious behavior. Finally, we show that information extracted from a static analysis of a framework can be leveraged at run-time to harden applications from a privacy point of view. For instance, we implemented a user-driven dynamic policy enforcement which enables users to enable or disable permissions for any applications.

To better understand the Android Permission System, let us briefly come back to the premise of *access control*. Indeed, ever since multiple users can access the same computer, people have had the need to protect their data from other users, be them malicious or clumsy, using access control. In this chapter we introduce the notion of access control and motivate our choice of analyzing one permission-based system called Android. Sections 1.1 and 1.2 introduce the notion of access control starting from the first computer to introduce password to the latest permission-based system, Android.

Section 1.3 illustrates shortcomings of permission-based systems. Section 1.4 explains the challenges we face when studying and analyzing a permission-based system. Finally, Section 1.5 lists the contributions of this work.

[1] Android, for instance, is a permission-based system since it limits access to protected resources by checking that applications have the required permission(s).

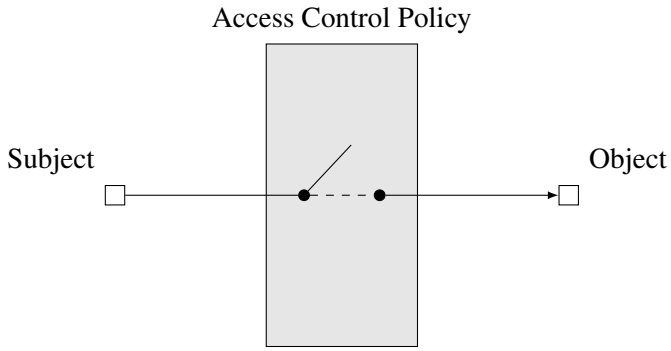


Figure 1.1: Generic Access Control: A subject (e.g., a user or process) is only authorized to access an object (e.g., a file) if the access control policy allows the access to the object by the subject.

1.1 A Brief History of Access-Control

One of the first computer to allow multiple users to work at the same time was the Compatible Time Sharing System (CTSS) in 1963 [32]. Each user had access to the computer and could send commands to it via a terminal and was given a personal directory to store files. As soon as more than one user store information on the same machine, questions about privacy, safety and security arise. Should anyone be able to see who created a file? What if someone erases another user's file by mistake? Should anyone be allowed to see any file of any user?

In the 1960's users of the CTSS were concerned about other users modifying their files. To deal with this problem, and CTSS was probably the first operating system introducing the idea, passwords were used to authenticate users. Once the user is authenticated by the system, she would only be authorized to access his/her files and not files of any other users [32, 112, 133]. The system makes sure that files are only accessed by authorized users. In other words, it protects resources of the system and prevents malicious or clumsy users from accessing or tampering with them. A user could share files with other users by allowing them to place a special file called *link*, referencing the shared file, in their directory. This is a first approach for access control. A generic model for access control is represented in Figure 1.1. Whenever a subject (e.g., the user) tries to access an object (e.g., a file), the system checks the access control policy to allow or deny the access.

The successor of CTSS introduced in 1965, called MULTICS (Multiplexed Information and Computing Service) [33, 112], was designed from the start with information protection in mind. As for CTSS, every user is given an identifier and authenticates herself to the system using a password. However, MULTICS introduces the concept of *group* of users and the concept of *Access Control List* (ACL). The ACL is an adaptable list of users who are permitted to read, write, execute or append an object (e.g., a program). Objects are organized in a single hierarchy tree of directories. If a user has access to modify directories, she can modify the ACL of all objects under that directory. Since users can change access rights of the objects belonging to them, the access control is called *Discretionary Access Control* (DAC).

Inspired by MULTICS came UNIX [108] in the late 1960's early 1970's. In UNIX the file-system is a tree of files and directories. Every file and directory belongs to a user and a group. Moreover, each file and directory has permission bits allowing the owner to activate or deactivate read write or execute permissions for the user owning the file, the group to which the file belongs or other users.

In the following years computers were already spreading and used in companies which could afford one and by the military. This raised interest in computer security and particularly in access control. Effort have been made to formalize access control into mathematical models. The Bell-LaPadula access control model [16] was designed in 1973 and focuses on confidentiality, which is especially useful in military applications. In this model, subjects (e.g., users) can observe, alter or modify objects (e.g., files). A pair, defining the category and classification, called *security level* (e.g., category "cryptography" and classification "unclassified", category "nuclear" and classification "secret", category "chemical" and classification "top-secret", ...) is assigned to subjects and objects. A system implementing the Bell-LaPadula model must make sure that some properties hold (e.g., a user at classification "secret" cannot read a "top-secret" document). Those access rules are defined in a centralized policy by the administrator and cannot be modified or bypassed by users. This kind of access control is called *Mandatory Access Control* (MAC).

The Bell-LaPadula focuses on confidentiality and thus only limit access to data but does nothing to protect against corruption of data. In short it does not ensure data integrity. In 1977, another model was designed to tackle this issue: the Biba model [21]. This model makes sure that a subject cannot corrupt data at a higher level of security than the level of the subject.

In the late 1970's, early 1980's, the original UNIX system gave birth to many versions based on the same philosophy. The most known systems are probably the GNU/Linux, BSD-based and MAC X operating systems. Their basic access control type is DAC but MAC implementations also exist, for instance SELinux [121] is a MAC extension for the Linux kernel.

1.2 Access Control for the Masses

Configuring security policies of a system (i.e., the set of all access control rules) is a complex task that requires in-depth knowledge about the access control model of the system. Today, computers are ubiquitous and are used both by novice and experts. Configuring access control policies is not an easy task and may even be disregarded by novice users who only want the system to work and do not care about security considerations.

Cell phones have a population coverage rate of nearly 100% in Europe [73]. They have evolved from devices running a system whose main purpose was to give and receive phone calls to fully fledged computers. These devices are able to connect to the Internet, watch high resolution movies and play the latest 3D game, bundled with a phone application. Those computers, or smart-phones, run applications that users download from application *market*s available on the Internet. For smart-phones, applications are usually limited in the actions they can perform on system resources (e.g., GPS, Internet access, ...) by the set of permissions the developers defined for them.

When downloading an application the user can see the list of permissions the application requires and can decide to either allow the application to be granted all the permissions in the

list and install the application or choose not install the application at all. This permission model for access control puts the user in the position of administrator: he/she has to update the access control policy of the device every time a new application is installed. We call such systems *permission-based system*. The permission-based system we study in this work is Android.

1.3 Motivation for Permission-Based System Analysis

The three following Sections (1.3.1, 1.3.2 and 1.3.3) present examples motivating the need of analyzing applications running on top of a permission-based system. Those examples represent flaws in the design of a permission-based system and are thus independent of the permission-based system under study. In addition, Section 1.3.4 illustrates the fact that developers may unintentionally create vulnerabilities in the applications they develop because the documentation is incomplete. This motivates the analysis of the permission-based system itself to improve the documentation and/or the development process of applications. Finally, Section 1.3.5 highlights a limitation of the Android system regarding the freedom given to users about permissions and suggests an alternative which relies on results from the analysis of the Android permission-based system.

1.3.1 Confused Deputy

Applications can communicate together. If one application is given a permission, another application could *exploit* this application to misuse its authority. This kind of attack is called confused deputy attack and is illustrated in Figure 1.2. This attack frequently occurs because the confused deputy application wrongly assumes that only a limited and trusted application can access its interface. For that reason its interface is not well-protected and can be abused by malicious applications aware of the vulnerability. In the example of Figure 1.2 the confused deputy application has the Network permission and is thus able to enable or disable the network. It wrongly assumes that only trusted application can access its interface, and thus did not protect it correctly. The attacker exploits this non-protected interface by making the confused deputy use its authority to disable the network on behalf of the attacker.

1.3.2 Application Collusion

Applications are granted permissions and can communicate together. Thus, nothing prevents an application to get data from a permission protected resource (e.g., GPS coordinates) and to share this data with another application which may not have the permission to access the protected resource. Multiple applications *collude* when they collaborate for a common malicious goal.

Application collusion is illustrated in Figure 1.3. The attacker has to have two (or more) applications installed on the target device. Once applications are installed they communicate together to share their permissions. When installing individual applications, the user only see a limited set of permissions for each application. In the example of Figure 1.3, the first application only has permission GPS and the second application only declares permission Internet.

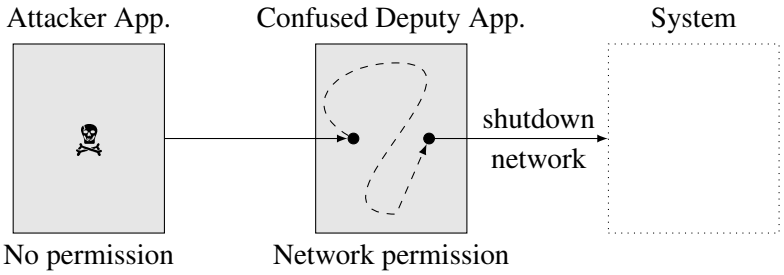


Figure 1.2: Confused Deputy: The application with a permission is not protected enough. Other applications can manipulate the interface of the application to misuse its authority. Despite the fact that the attacker application does not have the required permission, it uses the confused deputy application to shutdown access to the network.

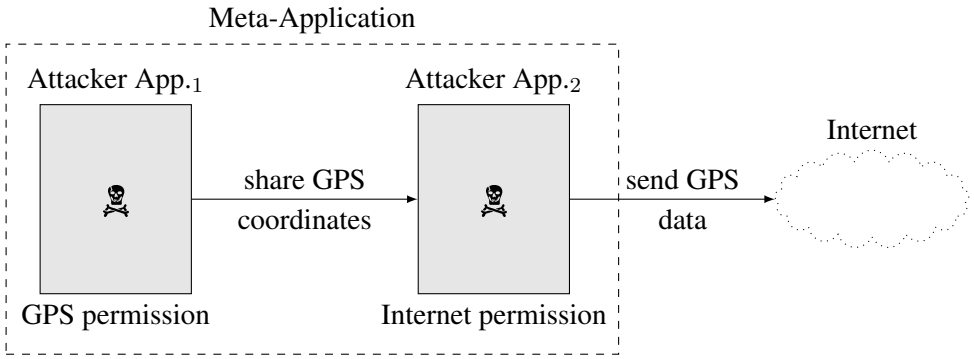


Figure 1.3: Application Collusion: An attacker could develop multiple applications which share permissions. Users installing applications separately do not see those as a meta-application with a big list of permissions.

However, application$_1$ can share data it obtain with its GPS permission with application$_2$. Application$_2$ having permission Internet, it can send the GPS coordinate to a remote host on the Internet.

1.3.3 Data Leakage

The example of Figure 1.3 illustrates two applications sharing GPS coordinates and sending it to the Internet. We say that the GPS data is leaked from the statement in App$_1$ retrieving the GPS coordinates (the *source*) to the statement in App$_2$ sending the coordinates to the Internet (the *sink*). This particular data leak occurs between two applications which is not common among malware applications. To be more efficient, most malware leak data within a single application.

1.3.4 Incomplete Documentation

An Android application contains a list of permissions describing which resources the application can have access to (e.g, access to the GPS). Developers of Android applications are responsible of writing this list of permissions. To achieve this, they rely on the documentation which, unfortunately, is incomplete [53]. In addition, they rely on code snippets that come with a permission list that they found on forums or websites [53]. However, a permission list may contain more permissions than necessary. Thus, a developer may write a permission list containing more permissions than what the application need, increasing the attack surface (i.e., all manners an attacker can enter the system and potentially cause damage [85]) of the application. Indeed, if an attacker compromises the application she has access to more resources than she would have access to with a reduced permission list.

In this thesis, a *permission gap* is defined as a set of permissions an application declares but does not use. To detect permission gaps, one has first to compute a set of permissions for all API methods. A permission list can then be computed automatically from the application code by looking at the API methods the application calls. Permissions mapped to those API methods form the list of permissions the application needs in order to work properly. The difference in permission between this automatically computed permission list and the permission list written by the developer of the application is called permission gap.

1.3.5 Fine-Grained Protection of User Data

Users tend to store, voluntarily or involuntarily, considerable amount of information they consider private on their device such as pictures, contact information, GPS coordinates, e-mail conversations or calendar information. On the first hand, they consider those information as private and protected by the device. On the other hand, they would like to install applications from remote repositories on the Internet (trusted and untrusted) and precisely control the permission list of those applications. However, they do not have the possibility to configure a fine-grained permission policy.

In this work, we explore the challenges of implementing such a system directly on the user device. The new software modifies the bytecode of Android applications and weaves in the code the access control policy. The policy can be defined at runtime by the user who can decide, for instance, to disable a permission for all installed applications.

In a nutshell, we have seen in this Section that (1) the confused deputy, application collusion and data leak examples motivate for leak detection in applications, (2) the incomplete documentation suggests that the framework itself should be analyzed, and (3) to improve data protection, users should be given more control over the security policy of the device.

In this thesis, we aim at analyzing Android applications and the Android framework from the security point-of-view. From the motivating examples comes the following challenging questions that we will answer in this thesis: How can Android applications be analyzed? Are permissions well-defined for Android applications? Can applications leak protected data? How can dynamic analysis complement static analysis? The next Section describes the technical challenges that we face when answering those questions.

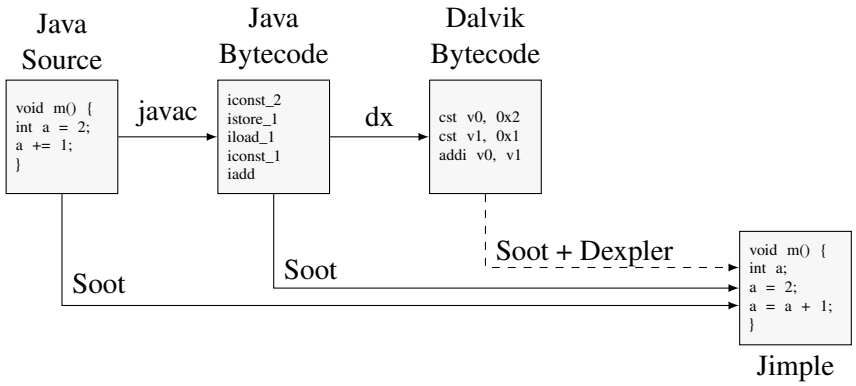


Figure 1.4: Even though Android applications are written in Java, which Soot can analyze, only the Dalvik bytecode is available in practice. To Analyze it Dexpler converts it to Jimple, Soot's internal representation of code.

1.4 Challenges for Permission-Based System Analysis

In this Section we introduce the technical challenges that we face when analyzing the Android framework and Android applications to answer to the research questions listed in Section 1.3.

Section 1.4.1 explains that Android applications use a special kind of bytecode that needs to be transformed to an analyzable representation. Sections 1.4.2 and 1.4.3 describe the difficulties to analyze the Android framework and Android applications, respectively. Finally, Section 1.4.4 highlights the challenges of running analysis of Android applications directly on a device.

1.4.1 Dalvik Bytecode

Android applications are written in Java then compiled to Java bytecode and finally compiled to Dalvik bytecode. Existing static analysis tools can analyze Android applications when either the source code or the Java bytecode of the application is available. However, this is not often the case as most applications are distributed through markets which only provide the final Dalvik bytecode. For instance there are more than one million applications available on the official Google Play market [2] whereas there are only about a thousand applications available on the F-Droid (Free and Open Source Android applications) website [3]. This motivates the use of a software module to converts Dalvik bytecode into an analyzable representation.

At the beginning of this thesis, in 2010, there was no available tool to perform complex static analysis of Dalvik bytecode. In order to be able to analyze Android applications, we developed a module called Dexpler to convert Dalvik bytecode into an analyzable representation. We leverage an existing tool called Soot whose internal representation of code is called Jimple. As represented in Figure 1.4, Soot is able to analyze Java source code and Java bytecode by

[2] `http://www.appbrain.com/stats/number-of-android-apps`
[3] `https://f-droid.org/`

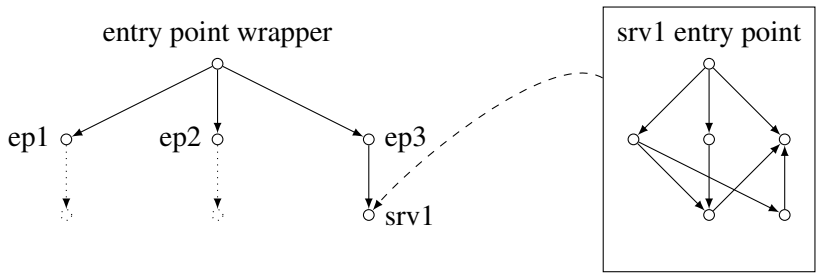


Figure 1.5: Analyzing a Framework requires (1) to generate wrapper code to call all entry points of the framework's API and (2) to handle domain-specificities such as calls to services for Android.

converting them to the Jimple representation. Dexpler converts Dalvik bytecode to Jimple in order for Soot to be able to analyze Dalvik bytecode.

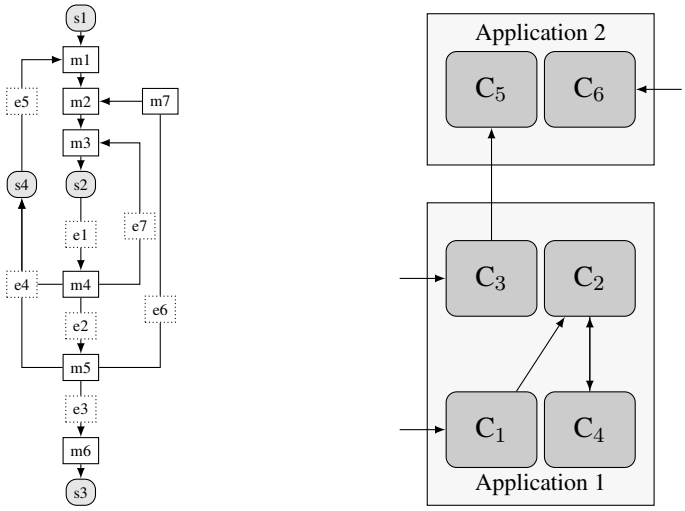
When converting Java bytecode to Dalvik bytecode some information about the type of variables are lost. For every method of the bytecode having such information loss, Dexpler finds those information back by analyzing the code of the method.

1.4.2 Analysis of the Framework

Static analysis has been initialy used on program or applications but not on APIs. With a program there usually is a single entry point from where the analysis begins. With a framework or API, however, there is no entry point: one has to construct *wrapper* code for all the entry points. The role of the wrapper code is to construct the object on which the entry point method is called as well as the parameters of the called entry point method. Moreover, constructing the wrapper code is not an easy task since one has to take into account how the methods of the API are called and how their parameters are initialized.

Static analysis of a framework starts by constructing a call graph from the wrapper code. In the case of Android, the API code is an interface to other programs or applications run by the system. Only constructing the call graph from the wrapper code would not work as system programs (responsible for checking permissions) would not have been initialized properly. Since system programs are launched and initialized when the Android device boots, the initialization code is not reachable from entry point methods. The solution we adopt is to find all system programs, initialize them separately from the entry points call graph, and refer to the initialized program whenever they are encountered in the entry points call graph.

Figure 1.5 presents a framework with three entry points (e.g., ep1, ep2 and ep3). Analyzing this framework requires to wrap those entry points. This is achieved by the "entry point wrapper" node which handles entry points initialization and parameter instantiation. During the call graph construction, ep3 calls code from service "srv1". This service has been initialized separately (Figure 1.5, right) and the call graph construction refers to the initialized service. Without service initialization, the call graph is incomplete since the service would be supposed non-existent.



(a) Component Lifecycle. (b) Links Between Components.

Figure 1.6: An Android Appication features Components. Each Component has its own Lifecycle. Two Components can communicate and form links between them.

1.4.3 Analysis of Applications

Analyzing an Android application differs from analyzing a Java application where there is a single entry point (i.e., the main method). Android application feature *components*, each of them having a lifecycle managed the the Android system. Figure 1.6a represents a simplified view of the lifecycle of an Android component called *activity*. The lifecycle features four states, s1 to s4, and moves from one state to another after events are produced (e1 to e7). Events trigger specify methods of the component (m1 to m7). The lifecycle of components is not present in Android applications. The Android system handles the lifecycle of Android applications' components. Since it would be too costly to analyze the Android system for every application to take into account the lifecycle of components, we use models of lifecycles instead.

Furthermore, any component could be called by another Android component from the same application or another application. An Android application does not have a single entry point but at least as many entry points as components. The components of an application are listed in the application's *manifest* file. In Figure 1.6b, black arrows with no source component represents entry points for components C_1 and C_3 of application one and for components C_5 and C_6 for application two.

Some components can be defined in the code of an application's component and not in the manifest. Thus, it is not sufficient to analyze the manifest, one also must check the code of components to find dynamically registered components. Moreover, components can communicate between one another. Components are loosely coupled in the sense that the connection between them is done at runtime, and communication between them relies on abstract messages called *Intent*s. In Figure 1.6b, there are intra-application communication between components C_1 and

C_2 and between C_2 and C_4 of application one and inter-application communication between C_3 of application one and C_5 of application two.

In a nutshell, when analyzing an Android application using static analysis, one has to model the lifecycle of the components and compute the communication links between components.

1.4.4 Analysis Directly on Devices

While is it interesting to perform analysis and transformation of applications off the device, it would be more interesting to perform them directly on the devices. For instance, users would directly benefit from having software allowing them to have a fine-grained permission policy (i.e., a policy allowing fine-tuning over the permissions. For instance instead of allowing full network access, the INTERNET permission could limit network access to a user-defined list of URLs). The main challenge is to be able to perform some analysis on devices with constraints on available memory and processing power and hard-coded constraint on available heap for a process. An analysis which requires 100MB of heap on the desktop cannot run on standard Android device where the heap limit is fixed to 80MB or less.

1.5 Contributions

In this Section we present our contributions to answer the research questions presented in Section 1.3 and tackle the associated challenges presented in Section 1.4. There are seven main contributions in this work to advancing the state-of-the-art in Android and permission-based system research:

- **An algorithm to convert Dalvik bytecode Jimple.** As Jimple is the internal representation of the static analysis framework Soot, Dexpler enables Soot to statically analyze Android applications. We evaluate Dexpler on a set of 25 thousand applications. Dexpler correctly transforms 99.9% of the applications' methods.

- **An algorithm to map API methods to permissions.** We propose an algorithm that first generates entry points from API methods then builds a call graph from API methods of a permission-based framework and finally use depth-first search to find permission checks and extract permission names.

- **An empirical analysis of the Android permission-based system.** We have implemented our algorithm to map API methods to permissions and tailored it to analyze the Android framework. Android-specific modifications include service redirection, service identity inversion, system services and managers initialization.

- **An empirical analysis of permission gaps.** The analysis is performed on two sets of Android applications. In the first one from the official Android market 18% of the applications present a permission gap and the second one from an alternative market 12% have a permission gap.

- **A tool to detect leaks within and between Android applications.** Our tool called IccTA links Android components at the Jimple level. This allows to detect inter-component and inter-application data leaks.
- **An empirical evaluation of IccTA to detect inter-component leaks.** We evaluate IccTA on DroidBench, a set of Android applications specially designed for testing tools which find intra- and inter-component leaks. Our algorithm outperforms existing tools with a precision 95% and a recall of 82%. We also evaluate our algorithm on a set of 3000 Android applications. It detects leaks in 450 of these applications.
- **The first empirical analysis of in vivo instrumentation.** This study shows the feasibility of instrumenting Android applications directly on devices. We also present two use-cases: the first use-case removes advertisement from application and the second use-case allows users to compose a fine-grained permission policy which is not possible with a native Android system. The use-cases show that the approach is possible and that the main limitation for performing advanced analyses is the heap size imposed by the system.

1.6 Roadmap of this Dissertation

This dissertation is organized as follows. In Chapter 2, we go through the fundamentals of static analysis, call graph construction, Android applications and the Android permission-based system. In Chapter 3 we discuss *Dexpler* a software to convert Dalvik bytecode to Jimple in order to analyze Android applications and the Android system. In Chapter 4 we analyze the Android framework to map permission to entry point methods. We use this knowledge to find applications that declare too many permissions which increases the attack surface for the end user. In Chapter 5, we describe a technique to find leaks of private data in Android applications. Next, in Chapter 6 we present our first results regarding in vivo Android applications instrumentation and analysis. Finally, in Chapter 8 we conclude the dissertation and discuss future work and open research questions.

Chapter 2

Technical Background

This chapter introduces the main technical background required to understand this PhD thesis. This chapter is divided into three Sections. Section 2.1 presents the Android system: it covers Android applications and their components, system services and access control with permissions. Then, Section 2.2 introduces the reader to static analysis and in particular call graph constructions for Java-based programs and inter-procedural data-flow analysis.

2.1 Introduction to the Android Stack

Android is a software system developed for smart-phones, tablet devices and more generally for a large palette of any kind of personal devices. It was originally developed at Android Inc. in the early 2000's [78]. Google bought the company in 2005 to further develop the system. The first publicly available device appeared in 2008 [79] and was running Android 1.0. Since then, there has been a new version released about every three months. At the time of writing (2014) the latest Android version is 4.4.

In this Section, we first give an overview of the Android system in Section 2.1.1. Then, we present the structure of an Android application in Section 2.1.2. Finally, in Section 2.1.3, we detail system services, one major part of the Android system.

2.1.1 Overall Architecture

Android is a *software stack* meaning that it features four main software *layers* as presented in Figure 2.1 (from top to bottom): the application layer, the framework layer, the runtime and native libraries layer and the kernel layer.

The top layer features Android applications. Typical Android applications are: the *Home* application which is the first running application that displays icons to start other applications; the *Contact* application to manage the list of contact; the *Phone* application to give phone calls; and the *Browser* application to visit web resources. Users of devices running Android can install more applications on their device, usually by downloading them from a repository such as F-

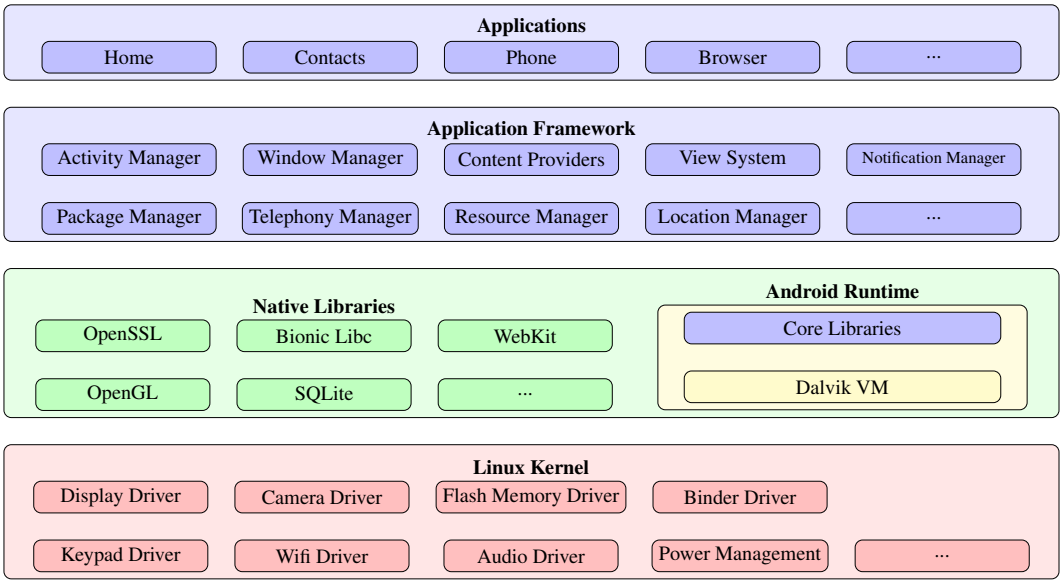


Figure 2.1: Android is a Software Stack.

Droid[1] or the official Google market named Play Store[2]. Applications are mainly written in the Java programming language but can also contain native code. Applications rely on the *framework* layer to communicate with the system.

The *framework* layer is an interface written in Java between applications and the rest of the system. It provides facilities to retrieve information from a system resource (e.g. the application can retrieve GPS coordinates through the *LocationManager*) or to ask the system to call them back when there is a new event (e.g. ask the *TelephonyManager* to notify the application when there is a phone call).

The third layer features two distinct entities: the Android runtime and the native libraries.

- The Android runtime consists of the *Dalvik virtual machine*, which executes Android applications' Dalvik bytecode[3], and Android core libraries, basically Java classes, which applications can leverage (e.g. application can use the `HttpsURLConnection` class to open a secure connection to a website). Some libraries contain wrappers around native libraries. For instance Java classes for the core library handling secure connections to websites such as `HttpsURLConnection` may use the *OpenSSL* native library depending on the environment's configuration.

- The native libraries[4] provide basic building blocks that can be used by applications, the framework layer or core libraries. Applications can have native code that directly use the

[1] `https://f-droid.org/`

[2] `http://play.google.com`

[3] applications are written in Java and compiled to Dalvik bytecode

[4] *native* because their instructions are directly executed by the CPU contrarily to the bytecode which requires to be interpreted by a virtual machine

native *OpenGL* library for fast graphic processing. The framework layer can use the native *SQLite* library to store data.

The lowest layer is the Linux kernel. From upper software layers it can be seen as an interface to the hardware (CPU, memory, ...). Indeed, it is responsible for running programs on the CPU[5] and it has a number of drivers to handle different hardware such as the display, the audio, and drivers to manager network communication. It also feature a special driver for efficient Inter-Process Communication called the *Binder* driver [116].

As we have seen, the layers are not clearly separated. An Android application can use elements from the *framework* layer, core and native libraries as well as directly communicate with the kernel. The Android system implements security features to prevent applications from having access to every part of the system. In short, developers give a list of permissions to every application they write. This list specifies what the application is allowed to do on the system and has to be validated by the user at installation time. When an application is installed, it is given a User ID (UID). Every Android application can be seen as a Linux user. Moreover, the Android system has a list mapping each permission to a Group ID (GID). For every permission the application declares, the system adds the application (or more precisely the corresponding Linux user) to the corresponding GID. So, if an application does not have the GPS permission and wants to retrieve the GPS coordinates through the LocationManager or the Linux driver for the GPS, the Android system detects that the application is not in the GPS group and prevents it from accessing GPS data.

2.1.2 Structure of Android Applications

An Android application is a compressed zip file signed with the private key K of the developer. It contains the Dalvik bytecode of the application (compiled from the Java source code), data the application needs (pictures, sound, ...) and a manifest file describing the application's structure and permissions the application requires. In short,

$$Application = Sign(Zip(DalvikBytecode, Manifest, Data), K).$$

The fact that Android applications are signed with the private key of the developer ensures that applications can only be updated by code signed by the same developer and that applications signed with the same key have the possibility to share permissions and UID. However, it does not guarantee the authenticity of the author of the application since certificates can be self-signed (e.g., anyone could claim to be John Doe).

Components

Android applications are made of *components*. There exists four kinds of components: *activity*, *service*, *content provider* and *broadcast receiver*. Activity components are used for the Graphical User Interface (GUI). They display graphical elements such as buttons, lists or pictures. Service components are used for computational intensive tasks or tasks that take a long time

[5] an instance of a program being executed is called a `process`

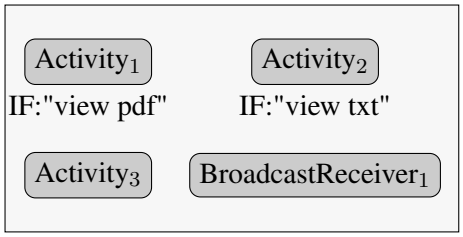


Figure 2.2: An Android Application with Four Components: Three Activities and One Broadcast Receiver.

such as playing an audio file. Content providers are used to share data between applications. For instance, the list of contact is implemented as a content provider so that any application can have access to it (if it has the proper permission). Finally, broadcast receiver components receive messages from the system or other applications (e.g. an SMS has been received by the system). Concretely, every component is a Java class which inherits from a specific super class such as Activity, Service, etc. Figure 2.2 represents an Android application made of three activities and one broadcast receiver.

Cummunication with Intent and URI

Components of an Android application usually communicate using special system methods called Inter-Component Communication (ICC) methods. There are about forty ICC methods which a component can use to communicate with another component. The most used ICC method is `startActivity(Intent)`. This method is used to tell the system to start a new activity component described by the method's parameter.

Intent. Components can communicate with one another using an abstract object called *Intent*. Communications can take place between components of a single application or between components of multiple applications. When component A wants to communicate with component B, it initializes an Intent and sets component B as the destination. This kind of communication is said to be *explicit* because the target component is explicitly specified. A communication can also be *implicit* in which case the source component initializes the Intent with the *action* it would like to perform (e.g. view a pdf document). When the component sends the Intent, the system checks for components having the action in their *intent filter*. The selection of the target component can be done automatically by the system or may require user intervention if multiple components can handle the action. For instance, if Activity$_3$ in Figure 2.2 sends an Intent with action "view txt" the system starts Activity$_2$ since it is the only component having the "view txt" intent filter. Intents can encapsulate data in form of key/value pairs in objects called *Bundle*s. Intents are used for communications between activities, services and broadcast receivers.

URI. A URI, or Uniform Resource Identifier, identifies an abstract or physical resource [19]. In short a URI is used to communicate with content providers. They may also be used to ini-

```
<manifest package="com.android.providers.calendar">
  <application android:process="com.android.calendar">
    <provider android:name="CalendarProvider" />
    <service android:name="CalendarSyncAdapterService" >
      <intent-filter>
        <action android:name="SyncAdapter" />
      </intent-filter>
    </service>
    <activity android:name="CalendarContentProvider" >
      <intent-filter>
        <action android:name="MAIN" />
        <category android:name="UNIT_TEST" />
      </intent-filter>
    </activity>
    <receiver android:name="CalendarReceiver">
      <intent-filter>
        <action android:name="BOOT_COMPLETED" />
      </intent-filter>
    </receiver>
  </application>
  <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Figure 2.3: Example of Android Manifest.

tialize Intents to target specific resource. Take the following URI: `content://com.android.calendar/events`. It can be cut in three parts. The first one, `content`, called the *scheme*, identifies how to access the resource. The reader may already know the `http` scheme for accessing web pages through the HTTP protocol. Here, `content` means that access to the resource is done through a content provider. The second part, `com.android.calendar`, called the *authority* identifies the holder of the resource. The reader may be familiar with authorities such as `mywebsite.com` which identify a registered host on the Internet. In our example, the authority identifies the content provider called `com.android.calendar` which has been register to the Android system. Finally, *events*, called the path, is the part identifying the target resource. The reader may be familiar with paths such as `index.html` identiying web page resources. In our example, this is the database table `events` of the content provider.

The Manifest

The manifest describes the application's structure in terms of components. A component can be exported so that other applications can use it. It can also declare intent filters to specify to the system what kind of action or data it handles. The manifest also lists all the permissions that the application requests (e.g. INTERNET, GPS). An example of manifest is presented in Figure 2.3. It declares an application with one content provider, one service, one activity and one broadcast receiver. The service only accepts intent with action *SyncAdapter*, the activity intents with action *MAIN* and category *UNIT_TEST* and the broadcast receiver intents with action *BOOT_COMPLETED*. The manifest declares one permission for the application: the *INTERNET* permission.

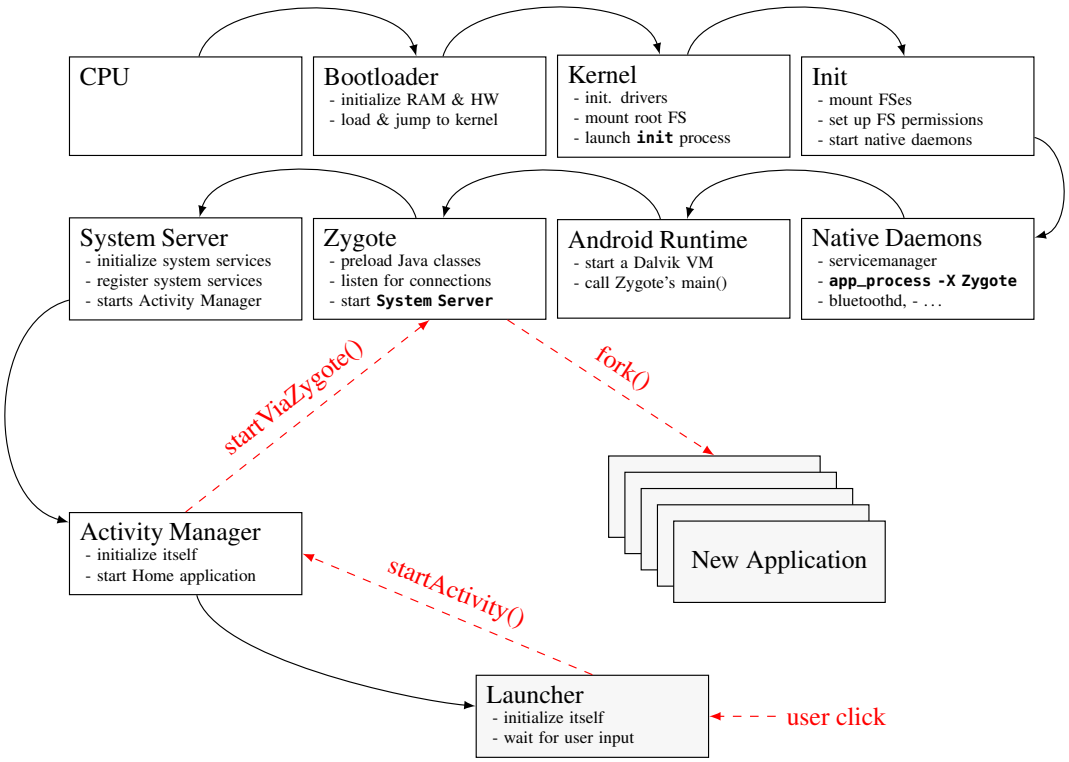


Figure 2.4: Android Boot (solid arrows) and Application Creation (dashed arrows). (Figure adapted from [139])

2.1.3 Structure of the Android System

Boot Process

The Android boot process is illustrated in Figure 2.4. When the Android device is switched on, the CPU first executes the boot-loader which initializes the RAM (Random Access Memory) and the hardware then loads the kernel and jumps to it. The kernel initializes drivers, mounts the root file-system and launches the first process: `init`. The init process mounts all file-systems, set up permissions on file-systems and starts native daemons. Native daemons are programs running in the background. Such programs include `bluetoothd` (the bluetooth daemon) which manages all bluetooth devices and `rild` (the radio interface layer daemon) an interface for radio devices (e.g. Global System for Mobile communication radio devices). One native daemon, `app_process`, starts an instance of the Dalvik virtual machine and initializes the Zygote, a program used to fork new Android applications. When the Zygote starts it launches the System Server, a process which initializes all system services [6] (e.g. the system service for GPS) and starts the Activity Manager. Finally, the Activity Manager starts the Home application.

[6] there are about 50 system services

At this point the user faces the graphical interface of the Home application. This GUI displays icons of Android applications installed on the device. When the user clicks an icon, the Home application calls the `startActivity` method. The Activity Manager handles this method call and asks the Zygote to fork itself to create and start the new Android application corresponding to the icon the user clicked on.

Processes Right after Boot.

When the boot sequence is completed, the Android system has been initialized and its kernel is running the processes shown in Figure 2.5. On the left is a stack of processes on top of which is the `bluetoothd` process. Right of the `bluetoothd` process is the service manager process which has been brought forward because it is used by the system server to register system services. Those are all native daemons started by the init process. In the middle is the Zygote, the process used to start new Android applications. The first process that the Zygote started is the system server which initializes, registers to the service manager, and runs all system services. Finally, on the right of the Figure is a stack representing Android applications. The first running Android application is the Home application. Through the interface of the Home application, the user can start other Android applications.

The Zygote, system server and the Android applications stack are running a process with a Dalvik virtual machine executing Dalvik bytecode. The virtual machine can also execute native code or shared libraries through the Java Native Interface (JNI). On the other hand, native services do not run a Dalvik VM.

Native services and the Zygote run as the root user. The system server runs as the system user to adhere to the principle of least privilege [113] (e.g., as a regular user, the system user cannot access data of other users). As already mentioned in Section 2.1.2, Android applications run as normal users and one user ID is assigned per application.

Android Access Control

The Android system runs Android applications. Those applications may access system resources such as GPS, the contact list or the camera. Android protects system resources with permissions. The system prevents applications from accessing a resource if they do not have the proper permission(s).

Figure 2.6 illustrates how an Android application accesses the GPS resource. The GPS resource requires specific hardware and thus a kernel driver to send commands to the hardware. An Android application is running in a Dalvik VM and can execute native code through the JNI. Theoretically, the application could use a native library to directly communicate with the device using the appropriate kernel driver. This communication from the application to the device driver is represented by dashed arrows in Figure 2.6. However, this direct communication is not allowed by the Android system because the application is running as a normal user whereas the driver can only be read and written to by the `system` user.

Instead, the application has to go through the binder driver and communicate with the system server (technical details about the application accessing the binder are hidden by the Manager class). Since the binder is running as a kernel module, it can certify to the system server the

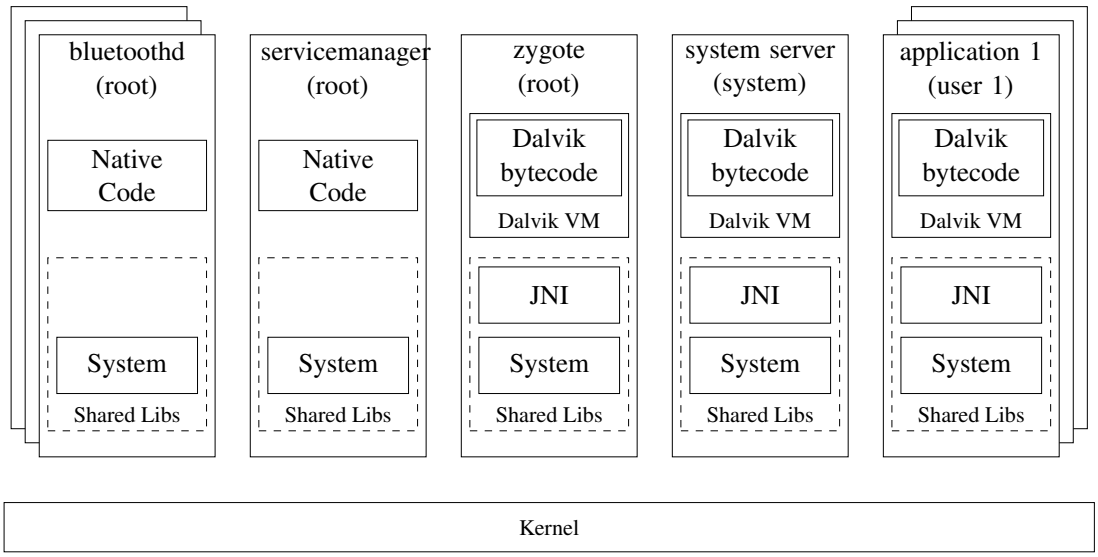


Figure 2.5: Android Processes after Boot. (Figure adapted from [57])

user ID of the calling application (i.e. the application cannot fake its ID). The system server then checks that the calling application has the right to access the resource (i.e. has the correct Android permission). If it has not, then the system server throws an Exception and the communication ends. If it has, the system server talks to the driver, retrieve the information and sends it back to the caller application. Since the system server runs as the `system` user, access to the driver is allowed.

Android Permissions

Permissions are classified into four categories or permission levels: normal, dangerous, signature and signature or system. Normal permission protect resources that are at low risk for the user (e.g., permission to read the battery status). Dangerous permissions protect resources that can either harm the user if they are stolen from her/him (e.g. the contact list) or cost her/him money (e.g. send SMS). The Android system only grants a signature permission to an application if it has been signed with the same certificate as the one used to signed the application that declared the permission. The Android system only grants a signature or system permission to an application if it has been signed with the same certificate as the one used to signed the application that declared the permission or if the application is in the Android system image.

Android 4.2 defines 200 permissions. Out of them, 29 are at permission level *normal*, 47 at permission level *dangerous*, 63 at permission level *signature* and 61 at permission level *signature or system*. In practice, most of the time, developers only deal with *normal* and *dangerous* permissions (29 + 47 = 76 permissions).

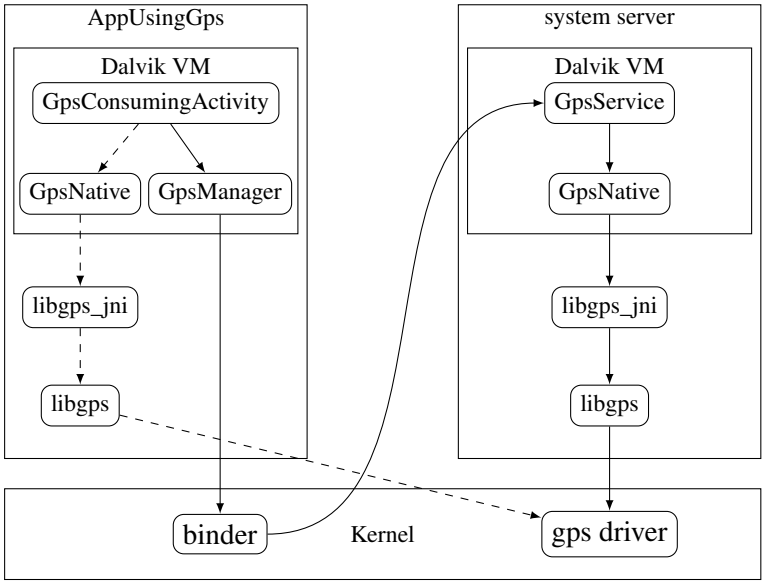


Figure 2.6: Android Access Control. (Figure adapted from [57])

2.2 Introduction to Static Analysis

Popular programming languages such as Java use classes and methods to represent concepts of the real world and manipulate those concepts (i.e., change their state), respectively. Statically analyzing programs built using classes and methods can be done using a *intra-procedural* approach (i.e, method by method) or using an *inter-procedural* approach (i.e, connecting the methods together and analyzing the program as a whole). This later approach requires to connect methods. This is done by computing a *call-graph* of the program.

In this Section we introduce call graphs construction and focus on the case of object-oriented languages, more particularly the Java language, since we analyze code from this language in Chapters 4 and 5.

2.2.1 Call Graph

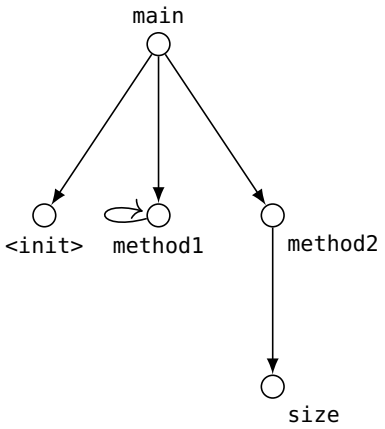
In Object-Oriented Programming a program is made of classes. Each class represents a concept (e.g. a car) has a set of fields to represent other objects (e.g., wheels) and a set of methods containing code to manipulate objects (e.g, drive the car forward). The code can call other methods to create instances of objects or manipulate existing objects.

A program usually starts with a single entry point method called the main method. By analyzing the code of the main method we can find out which method(s) it is calling. Then, by analyzing the code of the called method(s) we can find out what method(s) it/they are calling. The process can be repeated as long as there are methods calling other methods.

The result is a directed graph, called *call graph*, that links methods together. The graph starts from the node representing the main method and expands by going down through the

```
1  public class MyOjbect {
2    public static void main(String[] args) {
3      MyOtherObject o = new MyOtherObject();
4      if (args.length == 2) {
5        o.method1(2);
6      } else {
7        o.method2("hi!");
8      }
9    }
10 }
11
12 public class MyOtherObject {
13   int a = 0;
14   public MyOtherObject() {
15     this.a = 3;
16   }
17   public void method1(int i) {
18     this.a += i;
19     if (i == 55)
20       this.method1(55)
21   }
22   public void method2(String s) {
23     this.a += s.size();
24   }
25   public void method3(int j) {
26     this.method2(j);
27     this.method2(j);
28   }
29 }
```

(a) Two Java Classes



(b) Resulting Call Graph

Figure 2.7: Source Code of Two Java Classes and Call Graph Generated from main Method

called methods.

Figure 2.7 illustrates the call graph generation process on a Java program (a). There are two Java classes, *MyObject* and *MyOtherObject*. The starting point of the program is the main method in *MyObject*. This main method is also the starting point of the call graph (b). The main method first creates an instance by calling the constructor method of *MyOtherObject* (`<init>` method in Java). Then it calls methods `method1` and `method2` on the newly created object. `Method1` calls no other method but itself. `Method2` calls only size. `Method3` it not reachable from the main method and thus does not appear in the callgraph.

Precision

Inheritance and Polymorphism Object-oriented languages such has Java [61] use concepts such as inheritance and polymorphism. Figure 2.8 illustrates the two concepts. Inheritance is the ability to model an abstraction (here the abstract *Animal* class) and then to extend this abstraction to concrete elements of this abstraction (here classes *Human* and *Cat*). The abstraction defines a behavior through method `walk`. The *Human*s and *Cat*s both walk but in a different fashion that they describe in their own `walk` method.

Suppose we model a world of *Cat*s and *Human*s using the Java language. We would store every reference to *Cat*s and *Human*s in a container for *Animal*s. *Animal*s walk by iterating through the elements of the container and calling the `walk` method on them. When executing this code the method call `walk` on *Animal* will be redirected to `Human.walk` if the *Animal* is a *Human* or to `Cat.walk` if the *Animal* is a *Cat*. Providing a single interface for entities of different kinds is called polymorphism. When analyzing a Java program, the way polymorphism

```java
public abstract class Animal {
  public abstract void walk() {}
}

public class Human {
  public void walk() {
    // code to walk like a human
  }
}

public class Cat {
  public void walk() {
    // code to walk like a cat
  }
}
```

(a) Inheritance

```java
List<Animal> animals = new ArrayList<Animal>();
animals.add(new Human());
animals.add(new Cat());

for (Animal a : animals) {
  // humans walk like humans
  // cats like cats
  a.walk();
}
```

(b) Polymorphism

Figure 2.8: Inheritance and Polymorphism

is handled has a direct impact on the precision of the call graph.

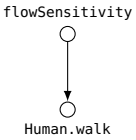
Flow Sensitivity A flow-sensitive analysis takes the order of statements into account. Take the code snippet of Figure 2.9a as an example. At line two, a new *Human* instance is created and is referred to by the *Animal* reference `a`. At line three, method `walk` is called on `a`. This means that at execution time only method `Human.walk` is called at line three. At line four, `a` now refers to a new instance of *Cat*. At line three a flow-sensitive analysis gives that `a` only points to a *Human* object and not a *Cat* object. Thus the flow-sensitive call graph contains a single edge to `Human.walk`. On the other hand, a flow insensitive analysis gives that `a` may point to either a *Human* or a *Cat* object since it does not take the order of statements into account. For a flow-insensitive analysis a program with line four between line two and line three would give the same result. Thus the flow-insensitive call graph contains two edges: one to `Human.walk` and another to `Cat.walk`.

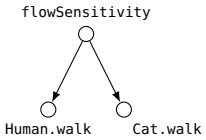
Path Sensitivity A path-sensitive analysis takes the execution path into account. Take the code snippet of Figure 2.9b as an example. At line two the *Animal* reference `a` points to no object. If the condition at line three is true, `a` points to a new *Human* object (line four). If the condition at line three is false, `a` points to a new *Cat* object (line six). A path-sensitive analysis would yield two paths for this example: path_1, `l2-l3-l4-l8`, and path_2, `l2-l3-l6-l8`. At line eight, path_1 has `a` pointing to a *Human* object and thus method `Human.walk` is in the call graph. Path_2 has `a` pointing to a *Cat* object and thus method `Cat.walk` is in the call graph. A path-insensitive approach does not take paths into account and would have `a` pointing to both a *Human* object and a *Cat* object at line eight. Thus, the path-insensitive call graph would contain both method `Human.walk` and method `Cat.walk`. A path-sensitive approach can produce a graph for every possible path. The number of path can explode exponentially and the approach becomes not scalable.

Field Sensitivity A field-sensitive approach models each field of each object. Take the code snipper of Figure 2.9c as an example. At line two, `c1` points to a new *C* object. This object contains two *Animal* fields. At line three, the first field, `f1`, points to a new *Human* object. At

Code Snippet	*-sensitive call-graph	*-insensitive call-graph†

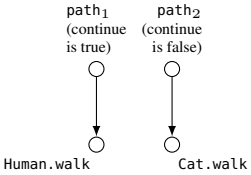
```
1  public void flowSensitivity() {
2    Animal a = new Human();
3    a.walk();
4    a = new Cat();
5  }
```

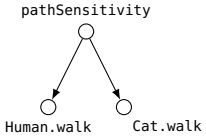




(a) Flow Sensitivity

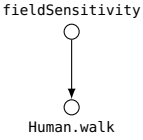
```
1  public void pathSensitivity() {
2    Animal a = null:
3    if (condition) {
4     a = new Human();
5    } else {
6     a = new Cat();
7    }
8    a.walk();
9  }
```

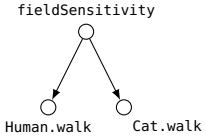




(b) Path Sensitivity

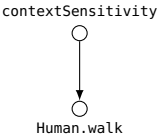
```
1  public void fieldSensitivity() {
2    C c1 = new C();
3    C c2 = new C();
4    c1.f1 = new Human();
5    c2.f1 = new Cat();
6    c1.f1.walk();
7  }
8  public class C {
9    Animal f1;
10 }
```

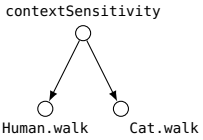




(c) Field Sensitivity

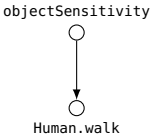
```
1  public void contextSensitivity() {
2    Human h = new Human();
3    Cat c = new Cat();
4    Animal a = method(c);
5    a = method(h);
6    a.walk();
7  }
8  public Animal method(Animal a) {
9    return a;
10 }
```

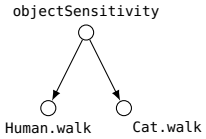




(d) Context Sensitivity

```
1  public void objectSensitivity() {
2    Contains c1 = new Contains();
3    Contains c2 = new Contains();
4    c1.setAnimal(new Human());
5    c2.setAnimal(new Cat());
6    c1.animal.walk();
7  }
8  public class Contains {
9    Animal animal;
10   public void setAnimal(Animal a) {
11     this.animal = a;
12   }
13 }
```





(e) Object Sensitivity

† except for field-sensitivity where a field-based call graph and not a field-insensitive call graph is represented

Figure 2.9: Sensitivities

line four, the second field, `f2`, points to a new *Cat* object. A field-sensitive analysis models each field of each object. Thus, at line five the model of `f1` can only points to a *Human* object and only method `Human.walk` is in the field-sensitive call graph. A field-based approach only models each field of each class of objects. This means that in the example field `c1.f1` and `c2.f1` have the same model. Thus, at line five `f1` points to a *Human* object and a *Cat* object and both method `Human.walk` and `Cat.walk` are in the field-insensitive call graph. Finally, note that a field-insensitive approach only models "objects". All fields are aggregated to their corresponding objects. A field-insensitive approach is unlikely to be used with languages such as Java, but only with languages featuring type-unsafe pointer operations such as C.

Context Sensitivity A context-sensitive approach models each *context* in which a method is called. There are two main ways of modeling the context: modeling the call site, which is described in this paragraph and modeling the allocation site of method calls, also called object-sensitivity, described in the next paragraph. Take the code snipper of Figure 2.9d as an example. At line two a *Human* object in instantiated. Variable `h` refers to this object. At line three, a *Cat* object is instantiated. Variable `c` refers to this object. At line four, `method` is called with `c`. The method returns an animal reference stored in `a`. At line five, `method` is called with `h`. The method returns an animal reference stored in `a`. At line six, method `walk` is called on `a`. A context-sensitive approach models each method call independently. That is for the first method call the model of the parameter points to `c` and the return value model points to `c`. For the second method call the model of the parameter points to `h` and the return value model points to `h`. Thus, only method `Human.walk` is in the call graph. On the other hand, a context-insensitive approach has only a single model of the parameter and a single model of the return value for a given method. In a context-insensitive the model of the parameter points to `c` and `h` and the return value to `c` and `h`. Thus, a context-insensitive approach has both method `Human.walk` and `Cat.walk` in the call graph.

Object Sensitivity An object-sensitive approach is a context-sensitive approach that distinguishes invocations of methods made on different objects. Take the code snippet of Figure 2.9e as an example. At line two and three, two *Contains* objects are instantiated. Variables `c1` and `c2` refer to these objects. The class *Contains* has an instance field `animal` of type *Animal* and an instance method `setAnimal` to associate a value with field `animal`. At line four, method `setAnimal` is called on `c1` with a *Human* object as parameter. At line five, method `setAnimal` is called on `c2` with a *Cat* object as parameter. Finally, at line six, method `walk` is called on the `animal` field of object `c1`. At lines four and five, an object-insensitive approach would consider `c1` and `c2` as the same receiver. The result would be that the method calls at line four and six cannot distinguish between the receiver and model `c1` and `c2` as a unique object `cu` of type *Contains*. Thus, method `walk` called on object `cu` at line six is represented by two methods in the call graph: `Human.walk` and `Cat.walk`. On the other hand, an object-sensitive approach would have model `c1` and `c2` separately for each call of `setAnimal`. Thus, the call at line six would only be represented by method `Human.walk` in the call graph.

Undecidability The problem of finding which pointers can refer to which objects at run-time using a static analysis is undecidable [77]. This means that no exact solution (i.e., both sound and complete) can be computed for all programs. However, many conservative solutions exist. These solutions are sound because they compute an over-approximation of the "real" result. As we have seen in the above paragraphs, they differ in precision (i.e., some approach generate more false-positives than other). For instance, a context-sensitive approach is in general more precise than a context-insensitive approach.

Algorithms to Compute a Call Graph for Java

Java is an object-oriented language which supports polymorphism. Thus, the exact types of the object on which a method is called (this kind of object is also called *receiver*) may not be known when performing the program static analysis. In the following paragraphs we briefly go through some algorithms that statically construct a call graph for Java programs.

CHA The first approach called *Class Hierarchy Analysis* (CHA) was proposed by Dean et al. [39]. The approach is very conservative and assumes that for a given receiver, the declaring type T of the receiver as well as all the subtypes of T (e.g., all subclasses) are possible types at runtime. For instance, when running CHA on a program having a method call `walk()` on a receiver r of type *Animal*, CHA assumes that *Animal*, *Human* and *Cat* are possible types for r at runtime. This will be the case even if no *Cat* object is instantiated in the analyzed program.

RTA A later approach called *Rapid Type Analysis* (RTA) was proposed by Bacon et al. [10]. This approach is similar to CHA, but only considers type T as possible type if an object of type T is created in the analyzed program. For instance, when running RTA on a program having a method call `walk()` on a receiver r of type *Animal*, RTA assumes that *Animal*, *Human* and *Cat* are possible types for r at runtime if and only if objects of types *Human* and *Cat* are created somewhere in the analyzed program.

VTA Sundaresan et al. [126] later proposed an other approach called *Variable-Type Analysis* (VTA). This approach is more precise than CHA or RAT since it only considers types that can reach a receiver as possible types.

Andersen The Java extensions [83, 110] of Andersen's algorithm [2] perform field-sensitive subset-based points-to analyses. A points-to analysis describes to what memory locations (i.e., local variables, global variables and dynamically allocated memory) a pointer expression may refer to. A call graph can almost directly be computed from the result of a points-to analysis. As described above, a field-sensitive approach models every field of every created objects separately. Given a statement such as `a = b` a subset approach adds the following constraint: $a \supseteq b$. This means that the points-to set of b is a subset of the points-to set of b. The algorithm collects all such constraints and uses a worklist to solve the constraints and construct the points-to set for all pointers. The worst-case complexity for Andersen-like pointer analyses on realistic Java programs is quadratic [122].

```
1  public void rfMethod() {
2    int x = 1;
3    int y = 2;
4    x = 3;
5    y = 4;
6  }
```

Figure 2.10: Example for the Reaching Definition Data-Flow Analysis

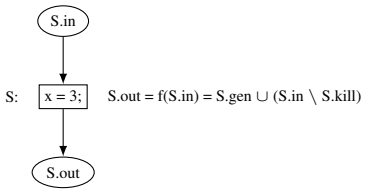


Figure 2.11: In, Out, Gen, Kill and Transfer Function

Steensguard Contrarily to Andersen's, Steensguard's algorithm [124] is equality based. This means that instead of subset constraints it uses equality constraints. Given the statement `a = b`, the constraints is $a = b$. This constraint means that the points-to set of a is the same as the points-to set of b. This approach is less precise than Andersen's but is more scalable (near to linear time complexity [124]).

Other Approaches Other points-to approaches have been developed [135, 119, 134, 64, 65] to either improve the efficiency of the subset-based points-to analysis introduced by Andersen or to improve the precision of the equality-based points-to analysis introduced by Steensguard. The interested reader may refer to [80] for an overview of these approaches.

2.2.2 Data-Flow Analysis

Intra-Procedural Analysis

A data-flow analysis [1] is a technique to compute at every point in a program a set of possible values. This set of values depends on the kind of problem that has to be solved using data-flow analysis. For instance, in the *reaching definition problem*, one wants to know the set of definitions (e.g., statements such as `int x = 3;`) reachable at every program point. In that particular problem, the set of possible values at program point `P` is the set of definitions that reaches `P` (i.e., the variable is not redefined before it reaches `P`). Take the example of Figure 2.10. Definition at line two reaches lines three and four but not five since variable `x` is redefined at line four. Definition at line three reaches lines four and five. Definition at line four reaches line five.

A data-flow analysis uses a system of equations to compute information at each program point or statement. Each statement has a set of possible values called *in*, which represents

the information valid before the statement. Each statement has a set of possible values called *out*, which represents the information valid after the statement. Each statement has an equation describing the effect of the statement on the *in* set. A statement `Stmt` can create new possible values represented by *Stmt.gen* and kill existing values, represented by *Stmt.kill*.

Figure 2.11 represents the statement at line four of Figure 2.10. This statement `S` has an *in* set containing the two definitions `x = 1;` and `y = 2;` which are the definitions that reach the statement at line four. Note that for a statement with multiple predecessors, the set *in* is defined as the union of the *out* sets of all the predecessors: $S.in = \cup_{p \in predecessors} p.out$. The statement `S` generates a new definition, `x = 3;`, and kills definitions for variable `x` in the *in* set. This behavior of statement `S` is represented by an equation $S.out = S.gen \cup (S.in \setminus S.kill)$. The function `f` $S.out = f(S.in)$ is called the *transfer function* (it can also be called *flow function* or *data-flow function*).

Forward and Backward The reaching definition problem is solved using a *forward* analysis. This means that the analysis starts at the first statement of the program and goes forward until it reaches the end of the program. Other problems are solved using a *backward* analysis where the analysis starts at the last statement and goes backward up to the first statement.

Inter-Procedural Analysis

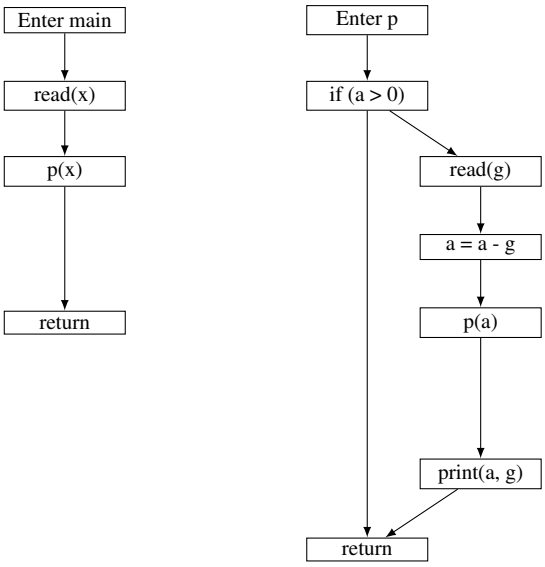
So far we have been looking at intra-procedural (i.e. within a single method) data-flow analyses. An inter-procedural analysis performs the analysis on connected procedures (or methods in Java). As we have seen in Section 2.2.1, computing a *call-graph* gives information about how methods are connected.

To illustrate an inter-procedural analysis we rely on the Inter-procedural Finite Distributive Subset (IFDS) framework [106]. The IFDS framework solves problems in polynomial time by transforming them in a graph-reachability problem. The algorithm used to solve the problem is called the "tabulation" algorithm. It is an improvement over previous approaches such as the "iterative" or the "call-strings" algorithms [87] which can take exponential time in the worst case.

IFDS Take Figure 2.12 as an example. The example is taken from [106] and illustrates the inter-procedural possibly uninitialized variables data-flow problem. Figure 2.12a represents a Java class with two methods, `main` and `p`. In this example, we suppose that method `read(a)`, is a system method reading an integer and storing it in local variable `a`, and that method `print(a,b)` is a system method printing the value of integers `a` and `b` on the screen. Figure 2.12b represents the control flow graphs of methods `main` and `p`, respectively.

A call graph for the code of Figure 2.12 would show that method `main` calls method `p` at line six and that method `p` calls itself at line 13. Each method call is represented by two nodes: the call node `c` and the return-site node `r`. Connecting the methods is done by adding three edges: one intra-procedural from `c` to `r`, one inter-procedural from `c` to the called method start node and one inter-procedural from the called method exit node to `r`. Individual method CFGs and connections between the methods using two nodes for the each call site and three edges form

```
1   public class C {
2    public int g;
3    public void main() {
4      int x;
5      read(x);
6      p(x);
7      return;
8    }
9    public p(int a) {
10     if (a > 0) {
11       read(g);
12       a = a - g;
13       p(a);
14       print(a, g);
15     }
16     return;
17   }
18  }
```



(a) Java class *C* contains two methods `main` and `p`.

(b) CFG of methods `main` (left) and `p` (right).

Figure 2.12: Example for the Possibly Uninitialized Variable Problem (Figure adapted from [106]).

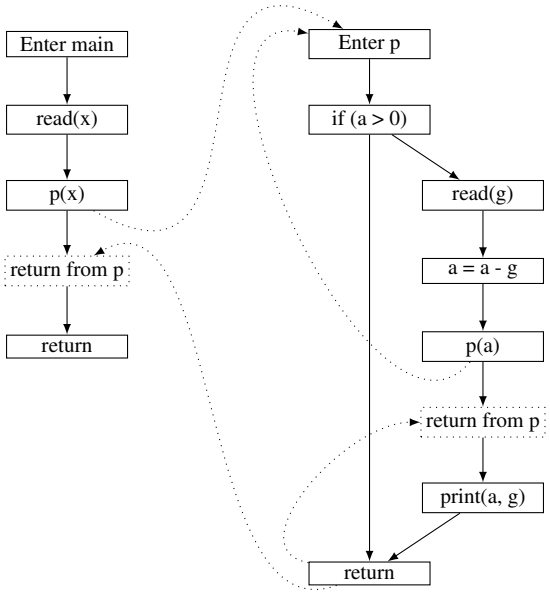


Figure 2.13: The Supergraph for the Possibly Uninitialized Variable Problem (Figure adapted from [106]).

the supergraph. The supergraph for example in Figure 2.12 is represented in Figure 2.13.

The set of data-flow *facts* for the possibly uninitialized variables is the set of local and global variables available in each method. Method `main` has local variable `x` and global variable `g`. Method `p` has local variable `a` and global variable `g`. The effect of each statement on the set of data-flow facts is represented by functions at every edge of the graph. For instance, the effect of `read(x)` is to initialize variable `x`. The function is represented by $\lambda S.(S - \{x\})$. This means that the output set of data-flow facts is the input set S from which element x is removed. Indeed, since x in now initialized by the current statement, it can be removed from the set of potentially uninitialized variables. Similarly, statement `a = a - g` is represented by function $\lambda S.$ if $((a \in S)$ or $(g \in S))$ then $(S \cup a)$ else $(S - \{a\})$. This means that if either a or g is in the input set, the output set is the input set plus element a (i.e., a is also undefined). Otherwise, the output set is the input set from which element a is removed (i.e., neither a nor g are undefined, so the new value of a cannot be undefined).

The IFDS framework represents every flow function f as a graph containing the following set of edges: $\{(0,0)\} \cup \{(0,y) \mid y \in f(\emptyset)\} \cup \{(x,y) \mid y \in f(x)$ and $y \notin f(\emptyset)\}$. The two functions described in the previous paragraph are represented as graphs in Table 2.1. Once all

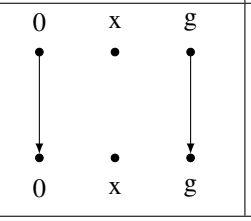
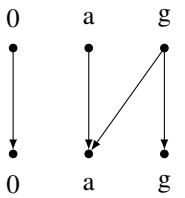
$\lambda S.(S - \{x\})$	$\lambda S.$ if $((a \in S)$ or $(g \in S))$ then $(S \cup a)$ else $(S - \{a\})$
	

Table 2.1: Two Functions and their Compact Graph Representation.

functions have been represented as compact graphs, they can be assembled to form the exploded supergraph. The exploded supergraph is illustrated in Figure 2.14. This graph converts an IFDS problem to a graph-reachability problem. In brief, if a node of the exploded supergraph is reachable from the enter node of the main method, it means that the data-flow fact associated with it holds. For instance, node for variable g for statement `p(x)` in method `main` is reachable from node 0 of the enter node of the main method. This means that at statement `p(x)` in method `main`, variable g is potentially uninitialized.

IDE The IFDS frameworks handles problems where the set of data-flow facts D is finite and where data-flow functions are in $2^D \rightarrow 2^D$. This framework is enough for problems such as "reaching definitions", "available expressions" or "possibly uninitialized variables". However, some problems such as the "linear constant propagation" problem cannot be encoded with the IFDS framework because the set of data-flow facts would be infinite.

The Inter-procedural Distributive Environment (IDE) framework [111] solves problems in which the data-flow information at a program point is represented by an "environment". In other words, the data-flow facts are maps from a finite set of symbols D to a set of values L. This mapping is called the environment and is denoted $Env(D, L)$. Data-flow functions are called "environment transformers" and are of the form $Env(D, L) \xrightarrow{d} Env(D, L)$.

The IDE framework is a generalization of the IFDS framework: all IFDS problems can be represent as IDE problems, but not all IDE problems can be represented as IFDS problems. An IDE problem can be represented by a supergraph G^* (for a given program the IDE supergraph is the same as the IFDS supergraph), a set of program symbols D, a semi-lattice L and an assignment of environment transformers to the edges of G^*: $M : E^* \rightarrow (Env(D, L) \xrightarrow{d} Env(D, L))$.

2.3 Conclusion

In this chapter, we have first introduced the Android system and the structure of Android applications. This domain-specific knowledge is necessary to fully understand Chapters 4, 5 and 6 where approaches are always applied either on the Android framework or on Android applications. Then, we have introduced concepts of static analysis such as call graph construction or inter-procedural analysis. Call graph construction is the basic of the Android framework analysis

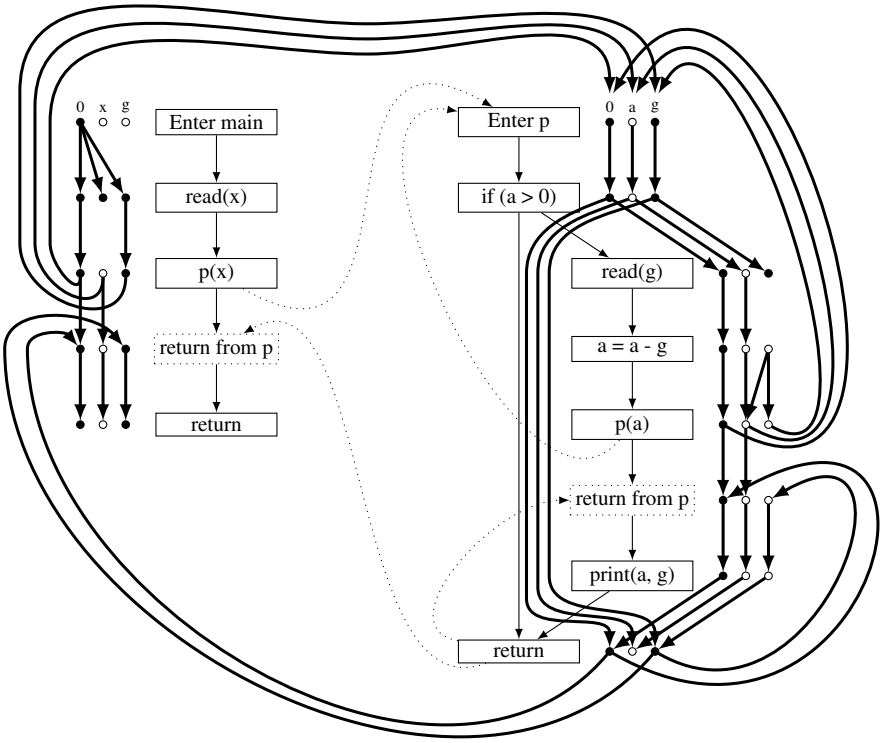


Figure 2.14: The Exploded Supergraph for the Possibly Uninitialized Variable IFDS Problem (Figure adapted from [106]).

presented in Chapter 4 while Chapter 5 heavily relies on inter-procedural analyses.

Chapter 3

Dexpler: Converting Dalvik Bytecode to Jimple to Enable Static Analysis of Android Applications

The goal of the chapter is to make possible the static analysis of Android applications. This Chapter describes `Dexpler` a software module which transforms Dalvik bytecode (i.e., the code of Android applications) to Jimple. Jimple is the internal representation of code of Soot, one of the most popular static analysis tool for Java-based programs. Converting Dalvik bytecode to Jimple enables to perform static analyses and transformations on Android applications.

This chapter is based on work that has been published in the following paper:

• **Alexandre Bartel**, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP@PLDI)*, 2012.

3.1 Introduction

Android applications are written in Java. However, they are not distributed as Java bytecode but rather as Dalvik bytecode. One possibility to analyze Android applications, would be to use a Dalvik disassembler such as Smali [62] or Androguard [41]. However, they are not designed to perform advanced static analysis such as data-flow analysis and they generally use their own representation of the bytecode which prevents them to use existing tools to perform the analysis.

Furthemore, analyzing Android applications with existing Java static analysis tools means that the Java source code or the Java bytecode of the Android application must be available. Most of the time, Android applications developers do not distribute the source code of their applications making the analysis of Android applications impossible with existing tools for analyzing Java programs. This is especially true for malware applications for which the source code is almost never available.

Another possibility to analyze Android applications is to first convert the Dalvik bytecode to Java bytecode, using Ded [47], Dex2jar [99] or undx [115], and then use a Java tailored static analysis tool such as Soot [131], BCEL [35] or WALA [70]. Tools which generate Java bytecode can leverage existing Java bytecode analyzers. However, the conversion from Dalvik to Java bytecode takes time and could be avoided by directly converting Dalvik bytecode to the internal representation of a tool.

To overcome these limitations, we introduce Dexpler[1], a module for Soot which directly reads Dalvik bytecode converts it to Jimple, Soot's internal representation of code, and fully type local variables of the Jimple representation. Any static analysis and/or transformation can then be applied on the Jimple representation. Using this method eliminates the intermediate Dalvik to Java bytecode conversion step and enables to use a faster and simpler tool chain for static analysis.

The contributions of this chapter are the following:

- we describe a Dalvik to Jimple converter tool called Dexpler
- we describe an algorithm to type the Dalvik bytecode
- we evaluate Dexpler on more than 25 thousand Android applications

The reminder of this chapter is organized as follows. Section 3.2 is an overview of the Dalvik bytecode. In Section 3.3 we describe Dexpler, the software which enables Soot to analyze Dalvik bytecode. In Section 3.4 we evaluate Dexpler on more than 25 thousand Android applications, present and discuss the results. Section 3.5 explains the current limitation of our tool. Finally we conclude the chapter and discuss open research challenges in Section 3.6.

3.2 Dalvik Bytecode and its Peculiarities

An Android application comes as a zip file containing the bytecode of the application, the Android manifest describing the structure of the application in terms of components and permissions it require, and data files (e.g., pictures, sounds). In this Section, we focus on the file containing the Dalvik bytecode of the application, also called the *dex* file in reference to the extension of the file name. Even if the original Android application is written in Java, no Java bytecode is found within an Android application. The Java code is first compiled into Java bytecode which is then transformed into Dalvik bytecode by the *dx* tool[2]. The reason behind the use of Dalvik bytecode it that it is register based and optimized to run on devices where memory and processing power are scarce. The structure of the `dex` file is described in Section 3.2.1. Dalvik instructions are presented in Section 3.2.2. Then, specificities of the Dalvik bytecode are explained in Section 3.2.3.

[1] Dexpler webpage: `http://www.abartel.net/dexpler/`

[2] *dx* is part of the Android SDK available at `http://developer.android.com/sdk/index.html`

3.2.1 Overall Structure

In this Section we first describe the structure of Java classes. Then, we describe the process to generate the `dex` file, containing all Dalvik classes, from Java files.

Java Classes As represented in Figure 3.1a, there is only a single place where literal constant values are stored (constant pool) per Java class. In Java, the constant pool is heterogeneous since different kind of Objects are mixed (e.g., Class, reference to Method, Integer, String). Every Java class contains a constant pool.

Dalvik Classes A single Dalvik executable is produced from N Java bytecode classes processed by the *dx* compiler. The resulting Dalvik bytecode is stored in a `.dex` file as represented in Figure 3.1b. A `dex` file contains description of Dalvik classes (name, fields, methods, ...) and Dalvik bytecode (a structure representing the code of concrete methods). Moreover, a `dex` file contains four homogeneous constants pools: for Strings, Class, Fields and Methods. All Dalvik classes share those four constant pools. Furthermore, a `.dex` file contains multiple *Class Definitions* each containing one or more *Method definition*. Each *Method definition* is linked to Dalvik bytecode instructions present in the *Data* section.

3.2.2 Dalvik Instruction

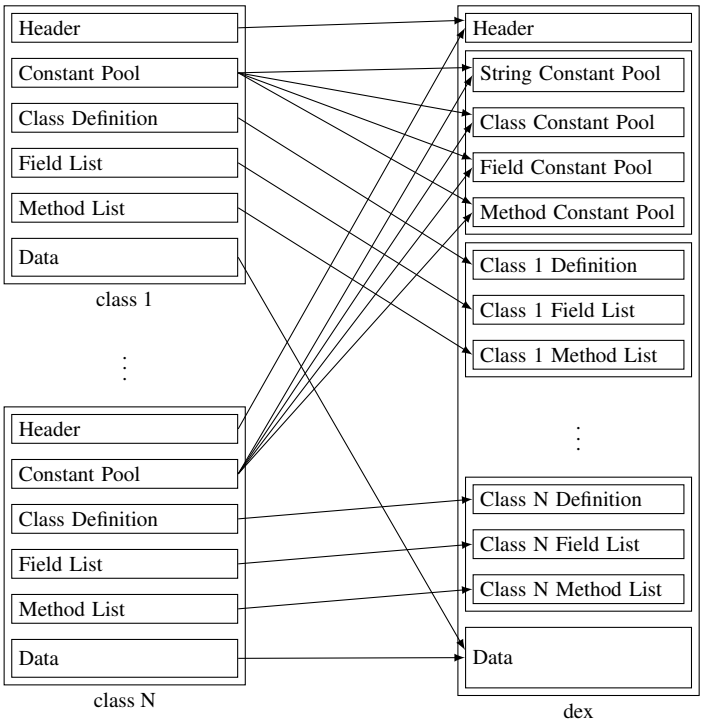
The Java virtual machine is stack based. This means that operands are pushed and popped from the stack according to the instructions semantic. On the other hand, the Dalvik virtual machine is register based. This means most instructions specify the name of registers they manipulate. This makes the Dalvik bytecode syntactically close to Jimple code since Jimple also uses a register-based representation of code. Figure 3.2.a represents Java bytecode where values are pushed to the stack whereas Figure 3.2.b represents Dalvik bytecode where values are assigned to registers `v0` and `v1`. The corresponding Jimple code would be `v0 = 0; v1 = 0` which is syntactically close to the Dalvik bytecode.

There are 237 opcodes present in the Dalvik opcode constant list. However, 12 odex (optimized dex) instructions cannot be found in Android applications' Dalvik bytecode as they are unsafe instructions generated within the Android system to optimize Dalvik bytecode. Moreover, 8 instructions were never found in application code [94]. According to those numbers, it is highly probably to only find 217 instructions in Android applications in practice.

The set of instructions can be divided between instructions which provide the type of the registers they manipulate (e.g., `sub-long v1, v2, v3` adds two long registers and stores the results to a long register) and those which do not (e.g., `const v0, 0xBEEF` stores a value with undefined type in register v0). Moreover, there is no distinction between `null` and `0` which are both represented as the `0` value.

3.2.3 Primitives and Null

In this Section we highlight the characteristics of the Dalvik bytecode which have an impact on the typing resolution of bytecode registers. Note that we use the term *register* when we refer to



(a) Class Files (b) Dex File

Figure 3.1: Dalvik Dex and Java Class

```
int i = 0;
Object o = null;
```

(a) Java Source

```
iconst_0
istore_2
aconst_null
astore_3
```

(b) Java bytecode

```
const/4 v0, 0x0
const/4 v1, 0x0
```

(c) Dalvik Bytecode

Figure 3.2: Zero and *null* Representation in Java Source Code, Java Bytecode and Dalvik Bytecode.

```
int i = 2;
i = i + 1;
```

```
iconst_2
istore_1
iload_1
iconst_1
iadd
```

```
const v0, 0x2
const v1, 0x1
add-int v0, v1
```

```
float f = 3.3f;
f = f + 1.1f;
```

```
ldc 3.3f
fstore_1
fload_1
ldc 1.1f
fadd
```

```
const v0, 0x40533333
const v1, 0x3f8ccccd
add-float v0, v1
```

(a) Java Source (b) Java bytecode (c) Dalvik Bytecode

Figure 3.3: Typing Differences between Java and Dalvik Bytecode Instructions

Dalvik bytecode variable and the terms local or variables when referring to Jimple variables.

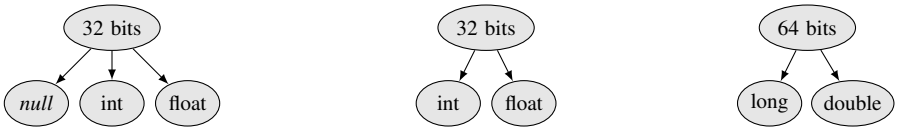
Primitives

In Java bytecode a primitive variable is initialized by an instruction which specifies its type (e.g., int, float, long, double). This is not the case in Dalvik where constants initializations have no type information. The code snippet on Figure 3.3 highlights those differences between Java and Dalvik bytecodes. In Java the type can be determined at every instruction: integer constants are initialized with specific instructions for integers while float constants are loaded from the constant pool in which they are tagged with the appropriate type (float in the example). In Dalvik however, the type information cannot be determined at a constant initialization instruction. However, for arithmetic operations, Dalvik uses instructions specifying the type of the operands. The type of a register can also be determined when it is given as a method parameter since the signature of every called method indicate the type of its parameters. In brief, when analyzing Dalvik bytecode the type of a register initialized by a constant can only be determined when the register is used.

Furthermore, in Dalvik, *float* and *integer* constants are both encoded on 32-bits. As illustrated in Figure 3.4, if a register is initialized with a 32-bits constant, the uses of the register have still to be analyzed to determine the register type. Similarly, *long* and *double* constants are both encoded on 64-bits. Therefore, a 64-bits constant assigned to a register does not directly give the type of the register. However, analyzing how the register is used will give the register type.

Null

Null is assigned to an object reference to indicate that it has no reference. In Java bytecode *null* is handled through a special *load constant* instruction (see Figure 3.2.b) and two *if* instructions to check whether an object reference is *null* or is non-*null*. In Dalvik bytecode there are no such instructions: *null* is represented as the integer value zero. Checking whether an object reference is *null* or is non-*null* consists in checking whether the object reference is an integer equal to zero or different than zero, respectively. Figure 3.2 illustrates that in the Java source code (a)



(a) 32-bits constant = 0 (b) 32-bits constant != 0 (c) 64-bits constant

Figure 3.4: Type Information From Constant Initialization

and bytecode (b) there is a clear difference between `0` and `null` whereas in Dalvik bytecode (c) there is no difference at all. As we will see in Section 3.3, the lack of type and the `null` representation becomes problematic when translating the Dalvik bytecode to Jimple.

3.2.4 Exceptions

In Dalvik as in Java bytecode, the bytecode contains exception handlers. A handler is a special set of instructions from a method's bytecode which is called by the virtual machine when a portion of the code throws an exception. Not handling peculiarities of Dalvik exceptions leads to untypable code. This is why we use a Dalvik specific exception model during the typing process and not the original Java exception model.

When constructing the CFG for a method, one has to add edges from instructions which could throw an exception to the first instruction of the correct exception handler. Exceptions in Dalvik are almost handled the same way as in Java but there are a few differences. Instructions returning from a method can throw an exception in Java but not in Dalvik. Instructions to store a class constant and a string constant can throw an exception in Java but not in Dalvik. Regarding arrays, Dalvik only throws exceptions for array instructions when the index is out-of-bound or if there is a null pointer on the array reference.There is no exception for instructions storing data into an array in Dalvik whereas there is in Java.

Exception handling could be considered as a technical detail. However, if not handled correctly it can break the typing of the code's variables. Consider the code presented in Figure 3.5. If an instruction between *label1* and *label2* can throw an exception, the handler instruction at label *handler* is called. This is the case for the `throw v4` instruction. If the Dalvik return instruction is handled as a Java bytecode return instruction, there would be an edge in the CFG from the instruction before the return instruction (i.e. `v1 = <Object getObject>`) to the exception handler. This edge is represented as a dashed arrow in Figure 3.5. In that case, when register `v1` is used in the handler its type could be both int (from `getInt` method) and Object (from `getObject` method). It would not be possible to obtain a typing for this code.

3.3 From Dalvik to Typed Jimple Code

This section describes Dexpler, the Dalvik to Jimple converter tool. It leverages the *dexlib2* library from the Smali disassembler [62] to parse Dalvik bytecode and the Soot *fast typing*, a Jimple component implementing a type inference algorithm [17], to type local variables. However, the type inference algorithm does not work on Jimple code naively generated from Dalvik

```
 v1 = <int getInt()>
label_1:
...
 throw v4
...
 v1 = <Object getObject()>;
 return v1;
label_2:

handler:
 v0 = @exception
 v3 = v1 + 2;
...
```

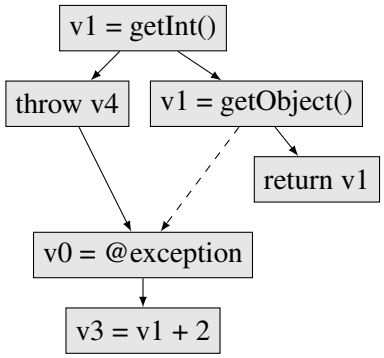


Figure 3.5: Incorrectly Handling Dalvik Exceptions in the CFG Introduces Typing Inconsistencies.

bytecode. We describe this typing problem in Section 3.3.1. Then, we describe how Dexpler solves this problem in Section 3.3.2.

3.3.1 Requirements of the Translation

Figure 3.6 represents the type lattice of Dalvik bytecode. Note that 32-bits and 64-bits abstractions are present since the type of constants is not known in the Dalvik bytecode: 32-bits constants could be either of type float, char, short, byte, boolean or int (recall from Figure 3.3 that initialization instructions do not give type information) and 64-bits constants of type double or long. Figure 3.7 represents the type lattice of Java bytecode. In Java bytecode constants for float, double, long and int are initialized with instructions specifying their type. Furthermore, there is a clear distinction between *null* that can only be assigned to objects and the value int 0 that can be assigned to int-like types.

Our aim is to convert Dalvik bytecode to Jimple and then remove type ambiguities so that existing typing algorithms can fully type the code. In order to achieve that goal we consider a simplified version of the the Dalvik type lattice represented in Figure 3.8. Existing typing algorithm such as [18] can type objects and sub-types of int, so we do not fully represent them in the simplified lattice. We want to go from the lattice in Figure 3.8, representing types in Dalvik bytecode, to the lattice in Figure 3.9 where there can be no ambiguity between types.

> More precisely, we want to distinguish (1) between 0 used as *null* and 0 used as the integer value zero, (2) between 32-bits constants used as integers and 32-bits constants used as floats (3) between 64-bits constants used as double and 64-bits constants used as long.

Once the typing of variables matches the lattice represented in Figure 3.9, the typing ambiguities have been removed and the full typing algorithm can be used to fully type variables.

The process of converting Dalvik bytecode to Jimple and typing the code is illustrated in Figure 3.10. First, the original Dalvik bytecode file (1) is parsed by the *dexlib2*[3] library and

[3] `http://code.google.com/p/smali/`

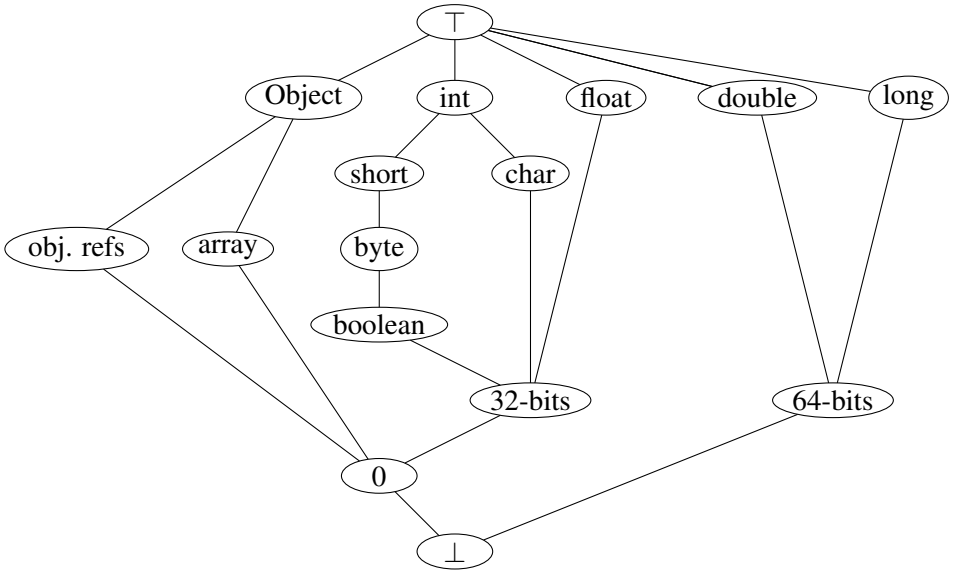


Figure 3.6: Dalvik Type Lattice

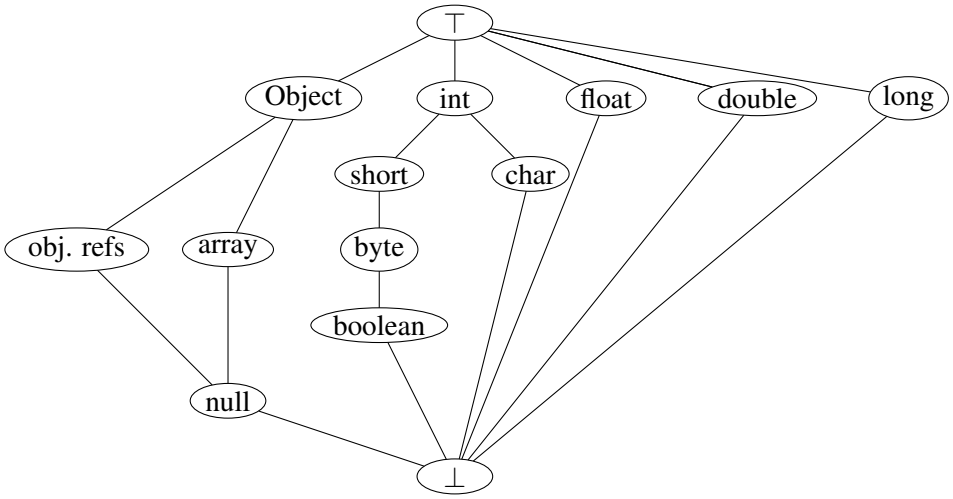


Figure 3.7: Java Type Lattice

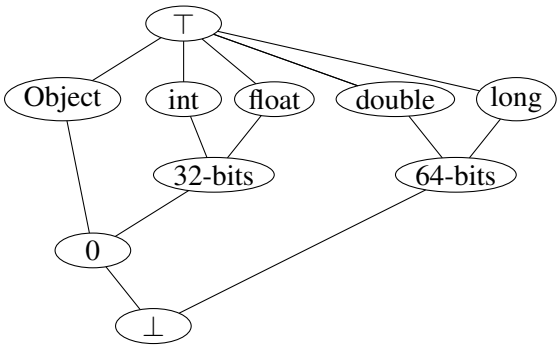


Figure 3.8: Simplified Type Lattice for Dalvik

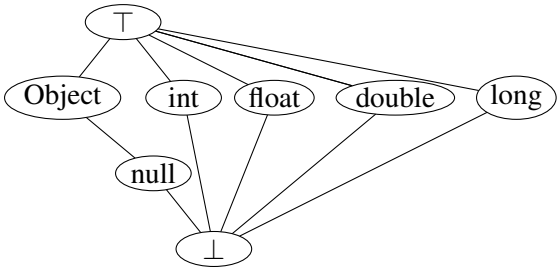


Figure 3.9: Simplified Target Type Lattice

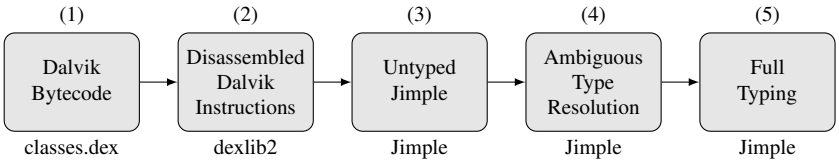


Figure 3.10: From Dalvik Bytecode to Full Typed Jimple

```
Coordinate newCoord = null;
while (newCoord != null) {
 newCoord = new Coordinate(1, 1);
}
if (newCoord == null) {
 [...]
}
```

Figure 3.11: Illustration of the *null* init problem.

every instruction is disassembled (2). From this intermediate representation, untyped Jimple statements are generated and connected together to from a Control Flow Graph (CFG) (3). The next step (4) resolves ambiguous types. Finally, in step (5), all Jimple locals are typed using the efficient local type inference algorithm presented by Bellamy et al. [18]. Step (5) is used to validate our approach. The next section describes step (4) in detail.

3.3.2 Ambiguous Type Resolution

We have seen in Section 3.2.3, that the following instructions lack type information: zero value constant initialization instructions (is it zero or null?) and constant initialization instructions (32 bits: integer or float?, 64 bits: long or double?). The following illustrates how we type the registers of these instructions by looking at how they are used in the code.

Null Initialization Figure 3.12 illustrates the problem with a bytecode snippet generated from the Java code of Figure 3.11. Register `v0` is initialized with 0 at line `01`. At this point we

```
00: const/4 v1, #int 1
01: const/4 v0, #int 0
02: if-eqz v0, 000a
04: new-instance v0, LCoordinate;
06: invoke {v0, v1, v1}, LCoordinate;.<init>:(II)V
09: goto 0002
0a: if-nez v0, 0013
[...]
13: ...
```

Figure 3.12: Resulting Dalvik Bytecode from Figure 3.11

do not know if `v0` is an integer, a float or a reference to an object. At line `02` we still do not have the answer. We have to wait until instruction at line `04` to known that the type of `v0` is `Coordinate`. At this point, the Jimple instruction generated for `01` has to be updated to be a *null* constant instead of the default integer constant with value zero. If this is not handled correctly, the typing component fails. Indeed, it is not possible for register `v0` to be both an integer and an object of type `Coordinate`.

Numeric Constant Initialization Similarly, initialization of *float* constants cannot be distinguished from initialization of *int* constants and initialization of *double* constants from initialization of *long* constants. Thus, we go through the graph of Jimple statements to find how constants are used and correct the initializations Jimple statements when needed. For instance, if a *float/int* constant (initialized by default to *int* in the Jimple statement) is later used in a *float* addition, the constant initialization changes from int constant to float constant.

Algorithm

Our algorithm can be divided in three steps executed sequentially:

1. an array type propagation step
2. a *null* versus zero differentiation step
3. an integer version float differentiation and long versus double step

Step 1: Array Type Propagation For every untyped constant we are looking at uses of local variables initialized with a constant value. The local can be stored in a field, used as a method parameter and stored in an array. For fields and methods, the type of the constant is known because the field signature and the method signature provide enough typing information. For an array on the other hand, the type information is not always known. A typical example would be when the array is aliased (i.e. assigned to another local).

To propagate type information we start at array initialization statements. For every such statement we find where the array is used. If the array is aliased we transfer the type information from the array to the new local. When a fix point is reached all local referencing arrays are typed.

Step 2: *null* and Zero For this step we designed Algorithm 1 on page 50. The algorithm starts at method `nullOrZero` (line 1). The algorithm starts by collecting statements that initialize local variables with an integer whose value is zero (line 3). Then, for every use of the local variable, method `forEveryUse` adds a type, either "used as object" or "used as integer", to the set of types (lines 5-6) for the local variable. If not all types for a local variable are consistent (i.e. they are not all the same), there is a type inconsistency in the method's bytecode making it impossible to type the bytecode. In that case, the algorithm ends with an error code (line 8) and the code executing the algorithm replaces the method code by a default code block which would throw a runtime exception. On the other hand, if all types are consistent, the local variable

definition(s) is/are updated accordingly: if the constant is used as an object, the zero value is replaced by *null* (line 10).

More precisely, method `forEveryUse` first collects all uses of the local defined by definition d (line 16). For every use, it checks wether the use type is an object or not an object. If the use type can be determined, the type is added to the set of types defined previously (line 19). If the use type cannot be determined there are three possibilities. The first possibility is that the statement under consideration is an alias statement. In that case the alias local variable is fetched and method `forEveryUse` is called again for that variable. The second possibility is that the statement under consideration is an *if* statement. If the *if* statement's condition contains another local, the method `collectDefinitionsWithAliases` is called with the other local. As its name suggests, this method collects all definitions of the local given as parameter. It checks the type for all definitions and also for all uses of the definitions. The last possibility is that the algorithm did not find any valid type for the statement. In that case the type is set to `UnknownType`. Just before the end of the loop over the use statements, the method `collectDefinitionsWithAliases` is called. This is useful in the case where there is not enough information to type the current analyzed local variable. Indeed, the algorithm may get more information about its type by looking at all of the definitions of the local variable and also by looking at how those other definitions are used.

Method `collectDefinitionsWithAliases` is called with a local variable as only parameter. It first collects all definitions of the local variable (line 38). For each definition, it checks if the local is defined as an object or not (line 40). If a type has been found it adds it to the set of types (line 42). Otherwise, it checks if the definition is an alias statement. It it is, it retrieves the alias and recursively calls `collectDefinitionsWithAliases` with the alias (line 44-45). Otherwise, if the statement is a zero constant definition, no type is added, since at this point types of zero constants are unknown (line 47). In all other case, the unknown type is added to the set of types. Finally, the method calls `forEveryUse` to check the types of all uses of all definitions (line 51).

Step 3: Float vs. Int and Long vs. Double After step 1 and step 2, the zero integer constants have been converted to *null* if they are used as objects. At this point we still have to correctly type 32-bits constants (float and int) and 64-bits constants (long and double). The approach is very similar to step 2. Algorithm 1 can be reused with slight changes. Method `getZeroDefs` is replaced by method `getNumDefs` which returns definitions where local variables are assigned a numerical constant (line 3). Methods `isObjectOrNot_Use` and `isObjectOrNot_Def` are replaced by methods `checkNumType_Use` and `checkNumType_Def`, respectively (lines 19 and 40). Those two new methods return the local variable use type and def type, respectively. Method `areTypesConsistent` checks that all types are the same, either all float, all int, all long or all double (line 7). Finally, method `correctDefs` update the definition according to the correct use type.

Step 4: Full Type Resolution Once ambiguities among constant definitions are resolved, a traditional typing algorithm for Java code is run. We use the one introduced by Bellamy et al. [17]. At the end of this process the code for a method is fully typed.

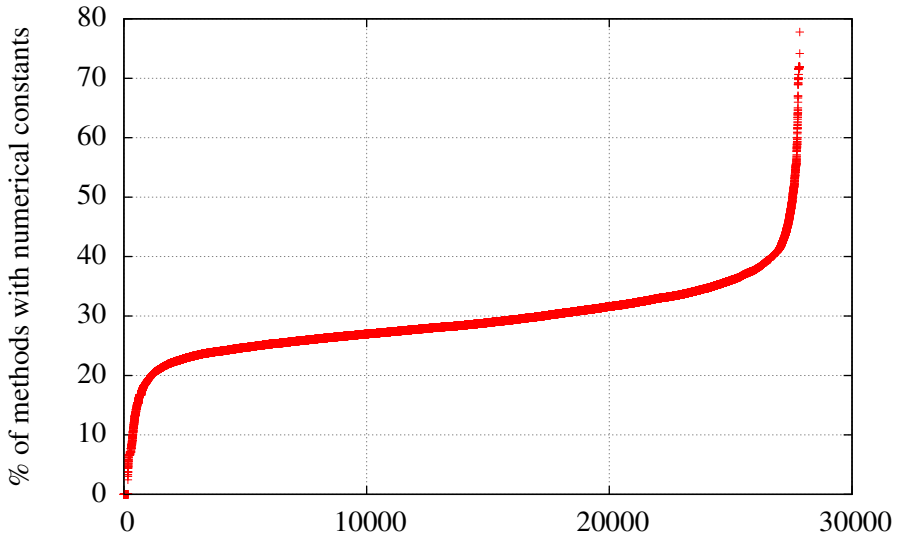


Figure 3.13: Ratio of Methods with Numerical Constants per Application

3.4 Evaluation

We evaluate Dexpler on 27,846 Android applications downloaded from Google Play. Together those applications account for 135,289,314 methods. Among them, 37,720,245 (28%) methods have a numerical zero constant definition statement. Figure 3.13 represents, for each application, the ratio of methods containing numerical zero constants. In total, 27,682 (99.41%) applications contain numerical constants. This results indicate that handling numerical constants is a must when analyzing Android applications.

The 27,846 applications are distributed on 100 nodes of the University of Luxembourg's High Performance Computing (ULHPC) [132]. The processing time plus the storage of metadata (e.g., size of each method, processing time per method) in a database requires 36 hours.

Dexpler correctly types variables for 135,288,415 (99.99%) methods. Dexpler fails to analyze 56 (0.20%) applications and all their methods due to exceptions generated by Soot. For 135 methods, step (5) did not run successfully.

3.4.1 Discussion on Failed Apks

Out of 27,682 applications, 56 (0.20%) were not correctly handled by Dexpler. Thirteen (23%) are non-valid Android applications produced by a bug in our application crawler or by an invalid magic number. Five (9%) because of out-of-memory exception. Six (11%) because of invalid field reference and 32 (57%) because of bytecode containing invalid instructions (e.g., *virtual-*

invoke instead of *interface-invoke* or *interface-invoke* instead of *virtual-invoke* [4]).

The 135 methods that did not pass step (5) have a similar characteristic: they all have variables used with different types in multiple branches. For instance, a variable initialized with zero could be used in the first branch as an integer and in the second as a null reference. One way of solving this would be to propagate the definition of the variable along the branches and remove the original definition. This would generate two variables each having its own type.

3.5 Limitations

Although our algorithm works in more than 99% of the 25 thousand applications we analyzed, they are situations (other than those in Section 3.4.1) that it does not handle yet.

3.5.1 Invalid Bytecode Never Executed and Never Checked by the VM

The Dalvik Virtual Machine (DVM) does only check the validity of classes as they are needed at runtime. Thus, the attacker could create a fake class containing methods with invalid bytecode. This class would never be used by the application itself so it will never be checked by the DVM and the application could be successfully installed on a device. However, a tool analyzing the application's bytecode, such as Dexpler, will analyze the invalid bytecode since it does not know if this class is used or not at runtime. If Dexpler does not find a valid typing for a method it replaces the code by instructions throwing an exception.

3.5.2 Invalid Dalvik Bytecode Bypassing the VM Verification

Bremer [24] has shown that is is possible to write Dalvik bytecode which is not checked by the DVM. The proof of concept exploits a bug in the DVM which does not check the bytecode if the class has a particular flag turned on. This kind of bytecode may be invalid in respect to typing constraints normally enforced by the DVM. This bug impacts all Android versions up to 4.3. The consequence is that it may be impossible to correctly type this kind of bytecode. During our experiments we checked the particular class flag used in the bug and did not find any application using this bug.

3.5.3 Hidden Bytecode

Outside the Bytecode. A Dalvik program can dynamically load Dalvik bytecode at runtime. An attacker could then hide the bytecode in a file (e.g. picture) and load it at runtime. Since this bytecode is not present in the *dex* file finding where convert it. Dexpler only converts the portion of code responsible for loading the hidden bytecode. Statically finding where the hidden bytecode is located is out of scope.

[4] even if these instructions are semantically wrong, it seems that the Dalvik Virtual Machine executes them anyway. We thus updated Soot to correct the wrong instructions.

Within the Bytecode. Schulz [117] presents a technique to hide Dalvik bytecode in a Dalvik bytecode array. This may confuse linear bytecode parsers which may not correctly interpret the array content as valid Dalvik bytecode. We check for every branch instruction that the destination is a valid instruction and not data in an array. We have never found a case where a branch instructions pointed to data in an array: this kind of obfuscation has not been used in any of the applications used to evaluate Dexpler. Currently Dexpler does not handle this obfuscated bytecode.

3.6 Conclusion

Dalvik bytecode cannot be analyzed by existing Java tools. In this chapter, we have presented Dexpler [5], a Soot modification with converts Dalvik bytecode to Jimple and allows to statically analyze Android applications. Dexpler leverages a model of Dalvik exceptions for constructing precise control flow graphs of methods' code as well as an algorithm to resolve variables with ambiguous types. The algorithm starts by identifying variables with an ambiguous type and then deduce their actual type by looking at use of these variables in the code. Dexpler has been evaluated on 135,289,314 methods from 27,846 Android applications and fully types variables for 99.9% of the analyzed methods.

[5] Dexpler webpage: `http://www.abartel.net/dexpler/`

Algorithm 1: Correctly Type *null* and `0` value.

```
1  int nullOrZero(Method m)
2  begin
3      defs = getZeroDefs(m)
4      for d ∈ defs do
5          types = Set()
6          forEveryUse(d);
7          if ! areTypesConsistent(types) then
8              return -1
9          end
10         correctDef(def, types)
11     end
12     return 0
13 end

14 forEveryUse(Def d)
15 begin
16     uses = getUses(d)
17     l = getLocalFromDef(d)
18     for use ∈ uses do
19         type = isObjectOrNot_Use(use)
20         if type != UnknownType then
21             types.add(type)
22         else if use is AliasStatement then
23             alias = getAlias(use)
24             forEveryUse(alias)
25         else if use is IfStmt then
26             if ifCondition contains other local then
27                 other = other local
28                 collectDefinitionsWithAliases(other)
29             end
30         else
31             types.add(UnknownType)
32         end
33         collectDefinitionsWithAliases(use, l)
34     end
35 end

36 collectDefinitionsWithAliases(local)
37 begin
38     defs = getDefs(local)
39     for def ∈ defs do
40         type = isOjbectOrNot_Def(def)
41         if type != Unknown then
42             types.add(type)
43         else if def is AliasStatement then
44             alias = getAlias(def)
45             collectDefinitionsWithAliases(alias)
46         else if def in ConstantZeroDef then
47             // do nothing
48         else
49             types.add(UnknownType)
50         end
51         forEveryuse(def)
52     end
53 end
```

Chapter 4

Finding and Removing Permission Gaps to Reduce the Attack Surface of Android Applications

The objective of this chapter is to check that developers do not give too many permissions to the Android applications they develop. Reducing the number of permission reduces the attack surface of an malicious user exploiting an application. We leverage Dexpler to analyze the code of applications to check which permissions they really require. This requires to deeply analyze the Android framework to extract a mapping between API methods (that Android application call) and required permissions. We present an Andersen-like field-sensitive approach using novel domain-specific optimizations to extract the mapping from the Android framework.

This chapter is based on work that has been published in the following papers:

• **Alexandre Bartel**, Jacques Klein, Martin Monperrus, and Yves Le Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. In *Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, 2012. Short paper.

• **Alexandre Bartel**, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. In *IEEE Transactions on Software Engineering (TSE)*, 2014.

4.1 Introduction

The security architecture of the mobile operating systems Android and Blackberry as well as other systems such as the Google Chrome browser extension system, use a similar security model called the permission-based security model [11]. A permission-based security model can be loosely defined as a model in which 1) each application is associated with a set of permissions

that allows accessing certain resources[1]; 2) permissions are explicitly accepted by users during the installation process and 3) permissions are checked at runtime when resources are requested.

In Android, the permission model is embedded into the "Android framework". The framework exposes an Application Programming Interface (API) that contains classes and methods for developers to interact with the system resources. For instance, the API contains a method `getGPSLocation`[2] which gives the current GPS location of the smartphone, if available. This API method, and many others, are sensitive with respect to security or privacy. Consequently, in response to a call to `getGPSLocation`, the framework checks that the caller has been explicitly granted the `GPS` permission.

This permission model has an impact on the development process of applications. To write an application, developers must identify, for each API method they use, the permissions that must be declared for the application to work correctly. They need a mapping between the API methods and the required permissions.

In the case of Android, the mapping is given by the official documentation. However, the documentation is not always up-to-date or clear and, consequently, question-and-answers website are full of questions regarding the use of permissions[3]. This results in that developers often either under- or over-estimate the required permissions. Missing a permission causes the application to crash. Adding too many of them is not secure. In the latter case, injected malware can use those declared, yet unused permissions, to achieve malicious goals. We call those unused permissions, "permission gap". Any permission gap results in insecure, suspicious or unreliable applications.

To sum up, having a a clear and precise mapping that links API methods and required permissions is of great value in a permission-based system such as Android. It enables developers to easily declare the permissions they actually need: not more, not less.

To extract this map, we explore in this chapter the use of statical analysis to extract the permission checks. On a framework of the scale and sophistication of Android, naive approaches using off-the-shelf static analysis fail miserably. This chapter discusses the building blocks that must be put together to extract a valuable mapping between API methods and permissions with two kinds of analysis: based on class hierarchy (CHA) and a field-sensitive, Andersen [2] like one called Spark [81]. Technically, we describe five components required for extracting permission checks in Android. The first one is a generic *String analysis*, yet essential for Android where permissions are not static constants but dynamic strings. The remaining ones are specific to Android. Of those four, the last two components specifically target Spark. *Service Redirection* redirects call to services to a properly initialized service (Android-specific). *Service Identity Inversion* avoids analyzing irrelevant system calls to services (Android-specific). *Service Initialization* properly initializes services for overcoming null values (Spark specific). *Entry Points Initialization* initializes all entry point methods and their parameters (Spark specific). The main difficulty of this research is that, due to the scale and complexity of Android, no building-block yields acceptable result in isolation. Eventually, we show that Spark can produce a good map-

[1] Contrary to the traditional Unix permission system where permissions are at the level of users, not applications.

[2] simplified view of the API

[3] e.g. `http://stackoverflow.com/questions/2378607/what-permission-do-i-need-to-access-internet-from-an-android-application/2378619`

ping of API methods to permissions, and we compare it against the related work [53, 6].

To sum up, the contributions of this chapter are:

- the empirical demonstration that off-the-shelf static analysis does not address the extraction of permission checks for a framework of the caliber of Android;
- three static analysis components (generic and Android-specific) to be put together in order to use Class Hierarchy Analysis (CHA) on Android;
- two static analysis components that allows one to use field-sensitive static analysis (Spark [81]) for analyzing Android's permissions;
- a comparison of our results against PScout [6], a static analysis designed in parallel as ours and against Felt et al.'s results based on dynamic analysis [53];
- an application of the extracted mapping on two sets of 1421 real Android applications showing that 129 (9%) applications suffer from a permission gap, i.e., they have more permissions that necessary.

The reminder of this chapter is organized as follows. In Section 4.2 we explain why reducing the attack surface is important and present a short study supporting our intuition. In Section 4.3 we propose a formalization for permission-based software. In Section 4.4 we describe the Android system and its access control mechanisms. Then, in Section 4.5 we extract the permission map from the Android system using static analysis. Experiments we conducted and results are presented and discussed in Section 4.6. In Section 4.7 we propose a generic methodology for deriving correct application permission sets. Finally we conclude the chapter and discuss open research challenges in Section 4.8.

4.2 The Permission Gap Problem

Let us now detail the permission gap problem introduced in Section 1. We also present short empirical facts showing that this problem actually happens in practice.

Possible Consequence of a Permission Gap Let us consider app_{wrong}, an Android application which is able to communicate with external servers since it is granted the `INTERNET` permission. Moreover, app_{wrong} has declared permission `CAMERA` while it does not use any code related to the camera. The `CAMERA` permission allows the application to take pictures without user intervention, i.e., the permission gap consists of a single permission: `CAMERA`. Unfortunately, app_{wrong} uses a native library on which a buffer-overflow exploit has recently been discovered.

As a result, an attacker can execute the code of its choice in the process of app_{wrong} by exploiting the buffer-overflow vulnerability. The code executed by the attacker in app_{wrong} is granted all permissions defined in app_{wrong}, INTERNET but also CAMERA. This effectively increases the attacker's privileges. In this particular example the attacker would be able to (1) write code to use the camera, take a picture and send the picture to a remote host on the Internet

and (2) execute this code in the target application by exploiting the buffer overflow vulnerability. This kind of attack is described in detail by Davi et al. [36].

On the contrary, if app_{wrong} does not declare `CAMERA`, this attack would not have been possible, and the consequences of the buffer-overflow exploit would have been mitigated. As noted by Manadhata [85], reducing the attack surface does not mean no risks, but less risks. In order to show that this example of misconfigured application is not artificial, we now discuss a short empirical study on the declaration of two permissions on 1,000+ Android applications.

Declaration and Usage of Permissions "camera" and "record audio" We conducted a short empirical study on 1000+ Android applications downloaded from the Freewarelovers application market[4]. For permissions `CAMERA` and `RECORD_AUDIO`, we grepped the source code of the Android framework to approximate the set of methods requiring one of them. These two sets of methods are noted M_{CAM} and $M_{\text{REC\_AUDIO}}$. Then, we computed the list A of all the applications which declare `CAMERA` or `RECORD_AUDIO`. Next, we took each application $app \in A$ individually and we checked whether the application uses at least one method of M_{CAM} and $M_{\text{REC\_AUDIO}}$ by analyzing the application's bytecode. If not, it means that app is not using the corresponding permission. When this happened, we modified the application manifest that declares the permission and run the application again to make sure that our grepping approximation did not yield false positives.

There are 7/82 applications that declare `CAMERA` while not using it. Similarly, 3/35 applications declare but do not use `RECORD_AUDIO` . Those results confirm our intuition: declared permission lists are not always required, and permission gaps indeed exist. Developers would benefit from a tool that automatically infers the set of required permissions and approximates permission gaps.

4.3 Definitions

Permission-based software is conceptually divided in three layers: 1) the core platform (the operating system) which is able to access all system resources (e.g., change the network policy); 2) a middleware responsible for providing a clean application programming interface (API) to the OS resources and for checking that applications have the right permissions when they want accessing them; 3) applications built on top of the middleware. They have to explicitly declare the permissions they require. Layers #2 and #3 motivate the generic label "permission-based software". Since the middleware also hides the OS complexity and provides an API, it is sometimes called, as in the case of Android, a "framework". Let us now define those terms.

Framework A framework $\mathcal{F}$ is a layer that enables applications to access resources available on the platform. We model it as a bi-partite graph where each node in the set of API method nodes connects a node in the set of resource nodes (this set also contains a 'no resource' node). *Example*: In Figure 4.1 the framework is composed of nine methods (four of them being public). Applications access the framework through four API methods. In the case of Android, $\mathcal{F}$ is the Android 4.0.1 Java Framework composed of 4,071 classes and 126,660 methods. To access a

[4] `http://www.freewarelovers.com/android/`

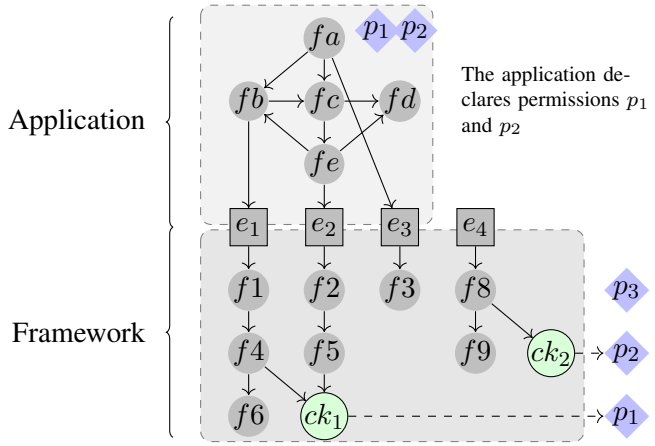


Figure 4.1: A Bird's Eye View of An Application Written on Top of a Permission-based Framework. (e_n are entry points, f_n are functions and methods, ck_n represent checks of permissions p_n)

resource, an Android application has to make a method call that goes through $\mathcal{F}$.

Permission A permission is a token that an application needs to access a specific resource.
Example: In Figure 4.1, the application declares two permissions. The framework defines three permissions but only checks two. We make no assumptions on permissions, and we consider them as independent (neither grouped, nor hierarchical).

Permission-based system A permission-based system is composed of at least one framework, a list of permissions and a list of protected resources. Each protected resource is associated with a fixed list of permissions.

Entry point An entry point of a framework is a method that an application can use (e.g., public or documented). Constructors are also considered as entry points. We denote $Entry_{\mathcal{F}}$ as the set of all entry points of $\mathcal{F}$.
Example: In Figure 4.1, there are four entry points (e_1 to e_4). An application can call any public method of the framework. Some methods accessing system resources (like an account) are protected by one or more permissions. In the case of Android 4.0.1, there are 50,029 entry points.

Declared permission A declared permission for an application app is a permission which is in the permission list of app. The set of all declared permission for an application app is noted $P_d(app)$.
Example: In Figure 4.1, the application declares p_1 and p_2. In the case of Android, the permissions of an application are declared in a file called *manifest*.

Required permission A required permission for an application app is a permission associated with a resource that app uses at least once. The set of all required permissions for an application app is noted $P_{req}(app)$.
Example: In Figure 4.1, the application requires permission p_1.

Inferred permission An inferred permission for an application app is a permission that an analysis technique found to be required for app.
Depending on the analysis technique used, the inferred permission list may be either an over- or an under- approximation of the required permission list. When developers write manifests, they write $P_d(app)$ by trying to guess $P_{req}(app)$ based on documentation and trial-and-errors. In Section 4.7, we propose to automatically infer a permission list $P_{ifrd}(app)$ in order to avoid this manual and error-prone activity.

4.4 Overview of Android

This section gives an overview of the architecture of Android. It briefly reminds the reader of Section 2.1 and then focuses on the parts related to permissions.

4.4.1 Software Stack

Android is a system with different layers. It consists of a modified Linux kernel, C/C++ libraries, a virtual machine called Dalvik, a Java framework compiled to Dalvik bytecode, and a set of applications. Applications for Android are written in Java and compiled into Dalvik bytecode. Dalvik bytecode is optimized to run on devices where memory and processing power are scarce. An Android application is packaged into an Android package file which contains the Dalvik bytecode, data (pictures, sounds ...) and a metadata file called the "manifest".

4.4.2 Android Permissions

Application vendors define a set of permissions for each application. For installing an application, the user has to approve as a whole all the permissions the application's developer has declared in the application manifest. If all permissions are approved, the application is installed and receives group memberships. The group memberships are used to check the permissions at runtime. For instance, an application *Foo* is given two group memberships `net_bt` and `inet` when installed with permissions `BLUETOOTH` and `INTERNET`, respectively. In other terms, the standard Unix ACL is used as an implementation means for checking permissions.

Android 2.2 defines 134 permissions in the `android.Manifest$permission` system inner class, whereas Android 4.0.1 defines 166 permissions. This gives us an upper-bound on the number of permissions which can be checked in the Android framework.

Android has two kinds of permissions: high level and low level permissions. High-level permissions are only checked at the framework level (that is, in the Java code of the Android SDK). Android 2.2 declares eight low-level permissions which are either checked in C/C++ native services (`RECORD AUDIO` for instance) or in the kernel (e.g., when creating a socket).

In this chapter, we focus on the high-level permissions that are only checked in the Android Java framework.

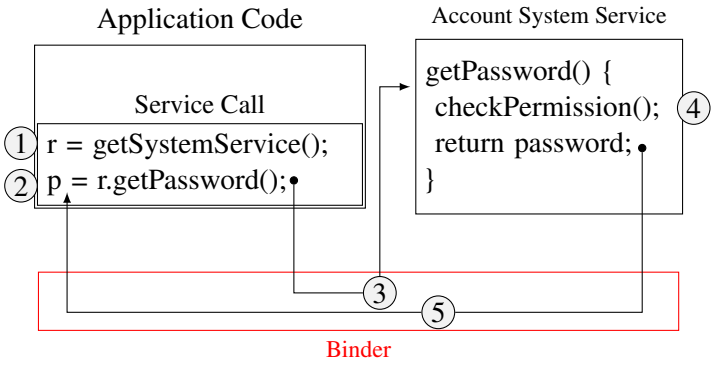


Figure 4.2: A Simplified Illustration of the Communication between an Android Application and a Permission Protected Service through the so-called "Binder".

4.4.3 Services and Permissions

An Android application is made of *components* which can be: an *Activity* that is a user interface; a *Service* that runs in background; a *BroadcastReceiver* (or *Receiver*) that listens for "intents" (a kind of message for inter process communication); a *ContentProvider* which is a kind of database used to store and share data. Most permissions are checked at the service level.

Android applications communicate with the operating system using a special kind of service called *system service*. System services are specific services running in a specific scope (called the "system server") and allow applications to access system resources (ex: GPS coordinates). Those resources may be protected by Android permissions to prevent access by unauthorized applications. Permission checks associated to services are mostly implemented in Java. Hence, the scope of the chapter consists of analyzing *Android permissions that are enforced in services in the Java framework*. The impact of this focus is discussed in Section 4.6.

It is important to understand the inner working of system services to devise good static analyses (that will be presented later in Section 4.5.2). We now describe how the applications communicate with system services. Applications synchronously communicate with system services through a mechanism called *Binder* as presented in Figure 4.2. The first step to communicate with a remote service is to dynamically get a reference (interface) to the service by calling `Context.getSystemService()` (step 1 in Figure 4.2). The next step is to call a method (method `getPassword` from the AccountManager Service in Figure 4.2) from the interface on the object reference *r* (step 2 in Figure 4.2). A special component, called "binder" is responsible for intercepting and redirecting that service calls to the remote service that performs the actual computation (steps 3 in Figure 4.2). The system service is responsible for enforcing permission checks (step 4 in Figure 4.2). To check that the caller's application declares the permission in its manifest (Section 4.4.1), the service calls one of the methods listed in appendix (Table 4.1) with the permission to be checked as parameter (not shown in the Figure). This specific point in the program is called Permission Enforcement Point or PEP. In Figure 4.2, if the application has the correct permission, the password is returned to the calling application (step 5).

4.4.4 Android Boot Process

It is important to know how to initialize system services when performing precise static analysis with Spark (Section 4.5.3). If services are not properly initialized, the analysis may be incomplete.

Let us briefly recall Section 2.1.3 and detail further the Android boot process. The first program to run on the device is the bootloader which provides support for loading, recovering or updating system images. The early startup code for loading the Linux kernel is very hardware dependent: it first initializes the environment and only then starts the architecture-independent Linux Kernel C code by jumping to the `start_kernel()` function. Then, high-level kernel subsystems are initialized (scheduler, system calls, process and thread operations ...) the root filesystem is mounted and the init process is started.

The init process creates mountpoints and mount filesystems, sets up filesystem permissions and starts daemons such as the network daemon, the zygote or the service manager. The zygote is a core process from which new Android processes are forked. The initialization of zygote starts the system server which in turn initializes system services and managers. System services include the input manager service and the wifi service. Managers include the activity manager which handles user interfaces (activities).

Android's boot process indicates that system services and managers are instantiated and initialized at boot time.

4.4.5 Android Communication

As presented in more detail in Section 2.1.2, components communicate with one another through the binder, the Android-specific Inter Process Communication (IPC) mechanism, and Remote Method Invocation (RMI) system. Components do not communicate with the binder directly but instead rely on three high-level abstractions of communication called *intent*, *query* and *proxy*. Figure 4.3 focuses on those communications at the Java level of the Android framework. It shows that an application communicate with the system server (and thus system services) through proxies and stubs (abstraction on top of the binder).

Intent Intents describe operations to be performed. They are used to start a new user interface screen (Activity), trigger a component which listens to intents (BroadcastReceiver) or communicate with services.

Query/Uri Queries are used to communicate with content provider components (which share data for instance through a database). Queries use Uniform Resource Identifier (URI) to indicate the target provider component on which the query must be performed.

Proxy/Stub System services extend stub classes which describe methods they must implement. System services are mainly used by application to access system resources. They are accessed by other components through their public interface called proxy. System services are running in the system server and are registered to the service manager. An application can get

`int`	`checkPermission (String, int, int)`
`int`	`checkCallingPermission (String)`
`int`	`checkCallingOrSelfPermission (String)`
`void`	`enforcePermission (String, int, int, String)`
`void`	`enforceCallingPermission (String, String)`
`void`	`enforceCallingOrSelfPermission (String, String)`

Table 4.1: List of Permission Check Methods of the `android.content.Context` Class (since Android 1.0 / API Level 1)

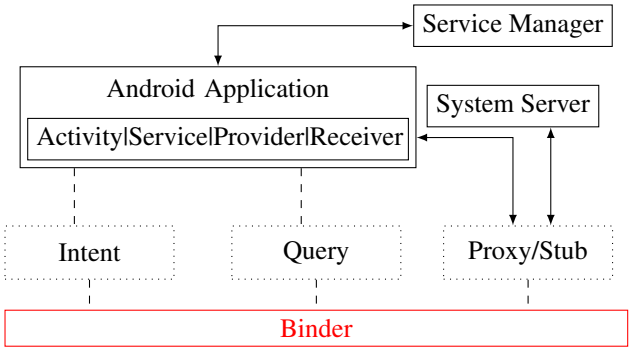


Figure 4.3: Android Communication Overview

a reference to a registered service through the service manager and can then communicate with the service through its proxy (which uses the binder).

4.5 Static Analyses for the Android Framework

Our goal is to define static analyses for extracting permission checks. In essence, each analysis constructs a call graph from the bytecode, finds permission check methods and extracts permission names.

Obtaining a meaningful call graph is challenging. We ran the default Soot's CHA-Naive (Class Hierarchy Analysis) on $50,029$ entry points methods of Android 4.0.1. It takes more than one week and outputs $31,458$ (64%) methods with no permissions, one method with a single permission[5] and $18,381$ (36%) entry points (methods) that each needs more than 100 high-level permissions. This is not meaningful. The reason is that Android has been implemented using the object-oriented paradigm and there are many subclasses of the core classes (e.g., of Service[6] , Activity[7], etc.). By construction, CHA outputs that all clients of those classes call all their subclasses. This results in an explosion of edges in the call graph and consequently

[5] this is the `INTERNET` permission checked in class `android.webkit.WebSettings`
[6] `https://developer.android.com/reference/android/app/Service.html`
[7] `https://developer.android.com/reference/android/app/activity.html`

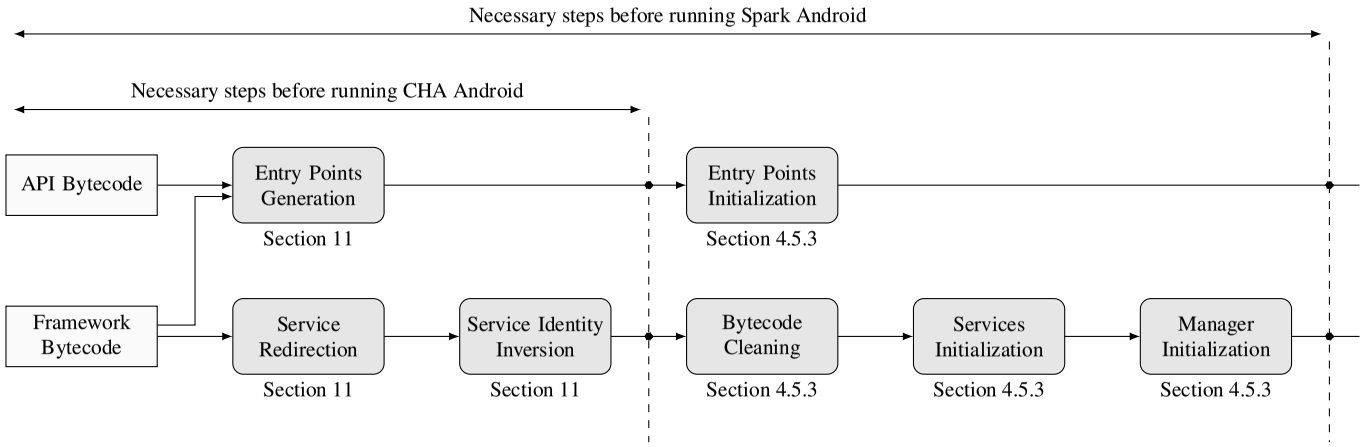


Figure 4.4: Bytecode Processing Before CHA-Android/Spark-Android Analyses. Entry points are generated using methods from the Android SDK API bytecode. Bytecode from the framework is transformed to redirect call to services to actual service classes, bypassing the ICC glue code. CHA-Android requires entry point generation, service redirection and service identity. Spark-Android is more precise and thus require proper entry points, services, and managers initialization.

an explosion of required permissions. **The main challenge for defining static analyses for extracting permission checks is to get a precise call graph.**

We still aim at using CHA, but we need to customize it for Android. We also aim at using Soot's Spark [81], an Andersen-like points-to analysis. Our motivations for running CHA are as follows. First, it enables us to identify key Android-specific analysis components. Those components can be reused with benefits in more sophisticated analyses such as Spark. Second, it gives us a baseline for assessing the improvements given by Spark. Third, it gives a list of API methods with no permission which do not require to be analyzed by Spark. Eventually, the best-of-breed of Android-specific analysis components and Spark enable us to obtain a precise permission map.

Figure 4.4 represents Android-specific components that manipulate the framework bytecode, and generate and initialize entry points. CHA-Android, the customized version of CHA for Android, requires generation of the entry point, presented in Section 11, service redirection, described in Section 11, and service identity inversion, detailed in Section 11. In addition to those components, Spark-Android, the customized version of Spark for Android, requires proper entry point initialization as well as services and managers initialization. Those components are described in Section 4.5.3.

In our experiments, the call graphs are generated from the 50,029 entry points found in the Android API version 4.0.1. All the analyses use Soot [76], a widely used framework for the static analysis of Java programs. The experiments run on a Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz running GNU/Linux Debian 3.11; the Java virtual machine 1.7.0 is given 4 Gb of heap memory. The Android version used in the experiments is 4.0.1 unless otherwise specified.

Section 4.5.1 presents the different components to modify the bytecode and to extract permissions from the call graph. Section 4.5.2 describes the CHA-Android analysis and Section 4.5.3 the Spark-Android analysis.

4.5.1 Common Components for CHA and Spark

In this section we present three techniques that are required for both CHA and Spark. String analysis is used to extract the permission names from the call graph. Service redirection enables the call graph construction algorithm to link the service caller to the service itself by bypassing the ICC glue code. Finally, service identity inversion removes code from the call graph which is executed as a system service itself and thus is not relevant from the entry point caller's point of view.

String Analysis for Extracting Permissions from Permission Enforcement Points

A basic call graph can only give the number of permission checks but not the actual names of the checked permissions because of the lack of string analysis to extract permission names from the bytecode. As explained in Section 4.4.3, Permission Enforcement Points (PEPs) are method calls to 6 methods of classes `Context` and `ContextWrapper` (see Table 4.1, in appendix, for a list of PEPs). Those method calls can be resolved statically. However, the actual permission(s) that are checked are dynamically set by a String parameter or sometimes, an array of strings. Thus, when a check permission method is found in the call graph, a basic analysis is only able to

Algorithm 2: The Algorithm that Extracts The Concrete Permissions Names (String Analysis)

Input: Method Call Stack, Target Method, Target Method Parameter
Result: Set of Permission Strings

```
1  stack ← Method Call Stack;
2  tm ← Target Method;
3  tp ← Target Parameter;
4  pSet ← set ();
5  pSet ← findPermission (tm, tp);
6  if pSet is empty then
7      tp ← getCurrentMethodParameter ();
8      N ← size(stack) − 1;
9      r ← StringAnalysis (stack[1...N], stack[N], tp);
10     pSet ← pSet ∪ r;
11 return pSet;
```

tell that a permission check occurs, but not which precise permission is checked because a call graph does not handle literal and variable resolution by itself.

To overcome this issue, we have implemented a String analysis as a Soot plugin whose pseudo code is shown in Algorithm 2. Once PEPs are found, it extracts the corresponding permission(s) (line 5). This plugin performs an intra-method analysis and manages the following scenarios: either (1) the permission is directly given as a literal parameter, or (2) the permission value is initialized in a variable which is given as a parameter, or (3) an array is initialized with several permissions and is given as a parameter. In every case we do a backward analysis of the method's bytecode using Soot's unit graphs which describe relations among statements of a method. In the case where only a single permission is given to the method, statements in the unit graph containing a reference to a valid Android permission String is extracted and the permission added to the list of the permissions needed by the method under analysis. In case of an array, all permissions of references to Android permission Strings are added to the list.

It can happen that the permission string cannot be found in the current method $\texttt{M}_i$'s body. This happens when it is referenced from a local variable initialized with one of the current method's parameter `P`. The solution is for the analysis to go one method down in the method call-stack (lines 6-10). At this point the analysis goes through the statements of $\texttt{M}_{i-1}$ looking for a call to `M`. When a call is found the parameter `P` is extracted and the string analysis starts again from there.

Service Redirection: Handling Binder-based Communication

Permission Size Explosion A call to a service method usually goes through a `manager` which gets a reference to a system service called `proxy`. It is always a method call on a `proxy` which results in data marshaling from the proxy through the `binder` to the `stub` on top of which lays the real system service method. All data transfers between the proxy and stub go through the `transact()` method which calls the `onTransact()` method. This method calls

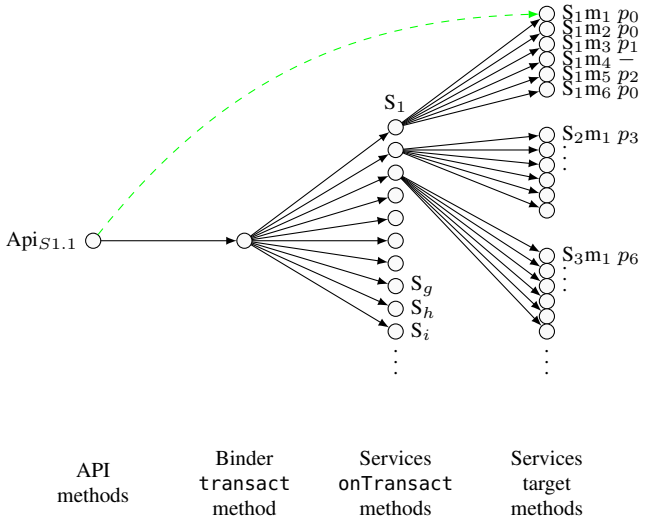


Figure 4.5: The number of edges explodes when an API method reaches the `transact` method of the `Binder` class. This node leads to an explosion in the number of permissions since it reaches all services' `onTransact` methods and each of those reaches all methods of their service. Those methods check for different permissions. Solving this problem boils down to short-circuit the low level `transact` and `onTransact` methods to directly reach the method of interest. The solution is represented by the dashed arrow which directly links an API method to its corresponding method in the right service. Thus, the API method is not mapped to permissions $\{p_0, p_1, p_2, p_3, p_6, \dots\}$ but only to permission p_0.

the right method on the system service side according to an integer value. This integer value is not determined when doing a static analysis. Thus, as illustrated in Figure 4.5, all methods of system services are added as edges in the call graph. Moreover, as all system services implement a `stub`, when constructing the call graph using CHA, all system services stubs' `onTransact()` methods are potential method calls from every method call on a proxy object and are thus added to the graph. A consequence of this is the explosion of the permission set size we observe when running CHA. In short, when doing a naive analysis from the point of view of services, any system service method call does have edges to all methods of every system service.

Service Redirection Figure 4.2 illustrates a communication between an application and a service. The communication is done through the binder. As explained in the previous paragraph, the problem is that analyzing binder based communications leads to an explosion in the number of permissions. The solution, illustrated Figure 4.5, is to bypass the binder (proxy/stub) mechanism by directly connecting a call to a service method to the corresponding method within the remote service. In Figure 4.2 edges from method `r.getPassword()` to the binder and from the binder to service method `getPassword()` are removed. Only the direct edge from the calling method to the called method (not shown in the Figure) is kept. As presented in Figure 4.4 this is

the first transformation done on the bytecode of the Android framework.

Service Identity Inversion

In Android, services can call other services either with the identity of the initial caller (by default) or with the identity of the service itself. In the later case, remote calls are within `clearIdentity()` and `restoreIdentity()` method calls. When using the service's own identity, permission checks are not done against the caller's declared permissions, but against the service's declared permissions. Since our goal is to compute the permission gap of an application (and not of system services), we can safely discard all permission checks that occur between calls to `clearIdentity()` and `restoreIdentity()`.

For instance, let us assume that service S requires and declares permission θ which is not declared by application A. If A calls S, the code of S is executed with the identity of A itself which would require A to declare θ. To avoid this, the portion of code requiring θ is executed with S's identity. When we encounter calls to `clearIdentity()` or `restoreIdentity()`, we use an intra-procedural flow-sensitive analysis to discard permission checks that occur between those calls.

Figure 4.4 shows that the Service Identity Inversion step is done after the Service Redirection transformation.

Entry Points Handling for CHA

In the case of an API (such as the Android API), the problem is that there is no "main" but N classes totalizing M entry point methods. Our solution is to build one call graph per public method of the Android API by creating one fake method $\mathtt{m}_{class_i}$ ($i \in (1, \ldots, N)$) per public class of the framework (for Android, `android.*` and `com.android.*`). The role of method $\mathtt{m}_{class_i}$ is to create an instance `o` of $class_i$ and to call all methods of $class_i$ on `o`. We also build a unique artificial main calling all $\mathtt{m}_{class_i}$ methods. This main method is the unique start point of the analysis. As presented in Figure 4.4, entry points are constructed using methods from the Android API.

Section 4.5.2 presents CHA-Android which leverages the service redirection, service identity inversion and entry point construction components.

4.5.2 CHA-Android

We perform the map construction with CHA because it enables us to identify key Android-specific analysis components that can be reused with benefits in more sophisticated analyses such as Spark, it gives us a baseline for assessing the improvements given by Spark and, more importantly, it gives a list of more than 30k API methods with no permission which do not require to be analyzed by Spark.

CHA-Android is a CHA-based static analysis for extracting permission checks on the Android framework. It uses the string analysis presented in Section 4.5.1, the service redirection (Binder) of Section 11, and the service identity inversion explained in Section 11. We enrich it with an optimization that we now describe.

Algorithm 3: The Algorithm that Extracts and Propagates the Permissions

Input: Call Graph
Result: Set of Methods with their Permission Sets
1 g1 ← Call Graph;
2 DephtFirstSearchAndPermissionExtraction (g1);
3 SCC ← TarjanFindSCC (g1);
4 g2 ← ReplaceSCC (g1, SCC);
5 PropagatePermissions (g2);

Call Graph Search Optimization

Section 4.5.1 describes how to extract permission names. This Section explains how permission names are propagated through the graph from PEPs. Algorithm 3 propagates permission sets through the graph. It proceeds in three steps. The first step (line 2) traverses the graph using depth first search and keeps track of the methods already visited. During the traversal it finds where permissions are checked and extracts the permission names (see string analysis above). This first step makes the analysis much faster than the naive approach since no method is analyzed more than once. Steps two and three make sure that permissions of already analyzed method are propagated in the graph. During the second step (lines 3-4) we use Tarjan's algorithm [**tarjan:connectedComponents**] to replace Strongly Connected Components (SCC) from the graph by a single node. This essentially removes loops from the graph and simplifies the propagation of permission names. During this step one has to be careful not to remove essential parts of the graph such as methods that check permissions since permissions are not propagated at this stage. Concretely, if a check permission method is part of an SCC it must not be removed from it otherwise permissions mapped to this method would not be propagated and thus be lost. The third and last step (line 5) propagates permissions throughout the graph.

This algorithm has a linear complexity in the number of nodes and edges. During the first step the graph is searched using depth-first search and methods are never analyzed twice: this step is bound linear in the number of edges and nodes. Tarjan's algorithm is bound linear in the number of nodes and edges. The last step propagates permissions through a depth first search of the graph where SCCs are replaced.

Empirical Results

Permission Strings Resolution Let us now analyze the efficiency of the string analysis. The distribution of the results of string analysis is presented in Table 4.2. We observe that 91.89% of the permission string analyses only check a single permission and that 83.25% of the analysis the permission string can directly be determined as a literal parameter. Hence, it is a common practice in the Java codebase of Android to (1) protect a method with only one or two permissions and (2) to make reference to permission strings and call the check permission method in the same method body. Those results show that for *99.08%* of permission checks the permission string is found using a string analysis.

Sometimes (0.92%), it is not possible to resolve permission strings: in 12 cases permissions

Total # analyses	1,516 (100.00%)
String found	
total	1,502 (99.08%)
with 1 permissions	1,393 (91.89%)
with 2 permissions	109 (7.19%)
with only direct strings	1,262 (83.25%)
with flow analysis	183 (12.07%)
with strings in array	57 (3.76%)
String not found	
total	14 (0.92%)
with URI read perm.	6 (0.40%)
with URI write perm.	6 (0.40%)
with read from parcel	2 (0.13%)

Table 4.2: The Kinds of Permission Specification as Found by Our String Extraction Analysis

are related to URIs; in two cases permissions are read from the Binder (Parcel).

Execution time On Android, CHA-Android analyzes 50,029 entry points in 4 minutes user time or 10 minutes real time. This shows that CHA-Android is able to scale on a large scale real world Framework.

Permission Set	# entry points
with 0 permissions	32,924 (65.8%)
with 1 permissions	39 (0.08%)
with 2 permissions	55 (0.12%)
with > 65 permissions	17,011 (34.0%)
	50,029 (100%)

Table 4.3: CHA-Android Permission Sets

Entry Point Permission Sets Running CHA-Android yields Table 4.3 which shows the permission set size for the entry points. As CHA-Android correctly models system service communications, the number of entry points requiring no permissions increases from 64% to 65.1% (31,458 to 32,429) (some service methods are not protected by permissions) and the number of entry points with one and two permissions increases from less than 0.01% to 0.08% (1 to 39) and from 0% to 0.12% (0 to 55) respectively (service method redirection avoids explosion in the number of edges in the call graph and thus the number of permissions).

Nevertheless, 34% (17,011) of entry points still have an over-approximated permission set. This is caused by the imprecision of the points-to set of CHA. This results in an explosion

in the number of permissions. An improvement would be to develop other domain-specific optimizations: handling other Android-specific points (e.g. content providers, handlers and messages) is similar to handling service communications and would not have an impact on the contributions of this chapter.

The following Section 4.5.3 presents the Spark based analysis. The analysis tackles Spark specific issues such as entry point initialization or Android-specific issues such as service initialization.

4.5.3 Spark-Android

We run Spark in context-insensitive, path-insensitive, flow-insensitive, field-sensitive mode to generate the call graph. Recall from Section 2.2 that in context-insensitive mode, every call to a same method is merged to a single edge independently of the context (receiver and parameters values). A path-insensitive analysis ignores conditional branching hence takes into account all paths of method bodies. The call graph construction is flow-insensitive since it does not consider the order of executions of instructions. It is also field-sensitive because it uses and propagates initialization data (e.g., constructor calls) to reduce the number of edges.

We first run a naive version of Spark-Android in Section 4.5.3 to illustrate the need to correctly initializing objects on which API methods are called as well as method's parameters.

Section 4.5.3 describes how we initialize entry points. It also explain another Spark subtlety: why and how system services must be initialized.

Naive Usage of Spark

As for CHA, we "naively" run off-the-shelf Spark to get a first understanding of the main problems that occur when analyzing the Android API. This gives us a key insight, Spark discards 96% of the API methods to be analyzed. The reason is that Spark is, field-sensitive, it only processes static methods and does not process the methods called on uninitialized references (e.g., initialized by default with *null*). This means it is not possible to run a Spark based analysis without correctly initializing entry points. Even with key Android-specific static analyses of CHA, a naive usage of Spark completely fails. Consequently, we need Spark specific analysis components.

Spark Specific Analysis Components

Processing Time Our first experiments show that Spark does not scale to the size of the Android framework. As we experience that Spark is time consuming when processing some entry points, we empty specific methods of certain classes to be able to compute permissions sets in a realistic amount of time (i.e., less than one day).

Analyzing time consuming entry points always leads to the windowing system classes. The windowing system is at the heart of Android components such as activities. It is responsible for the GUI (Graphical User Interface) management, and has relationships with numerous GUI abstractions such buttons or text fields and methods to start Android components such as other activities. When the call graphs hits a component of the windowing system it can grow in such

huge proportion, because of the imprecision in the point-to-sets, that the search in it triggers a timeout.

We make the hypothesis that classes responsible for GUI rendering and the windowing system management do not link to any permission check. Thus, we remove code of their methods and launch the experiments again. Removing the code means that (1) Spark does not construct the call graph for this code and thus that (2) the traversal of the call graph is much faster. With those modifications, the computation time of the permission map is much faster, terminates in less than 11 hours and does not trigger any timeout.

Entry Points Handling for Spark Spark-Android leverages artificial methods generated for CHA (see Section 11). However, it must initialize parameters of the 50,029 entry point methods of the Android API. Each receiver object `o` on which to call Android API methods as well as every method parameter `p` are initialized by calling $\texttt{generate}_o\texttt{()}$ and $\texttt{generate}_p\texttt{()}$, respectively. This tailor made method generates all possible instances of type `P` (i.e., over-approximation). Parameter initialization is necessary since one does not know a priori the effect of parameters on permission checks. Since Spark is field-sensitive, non-initialized parameters result in missing edges in the call graph.

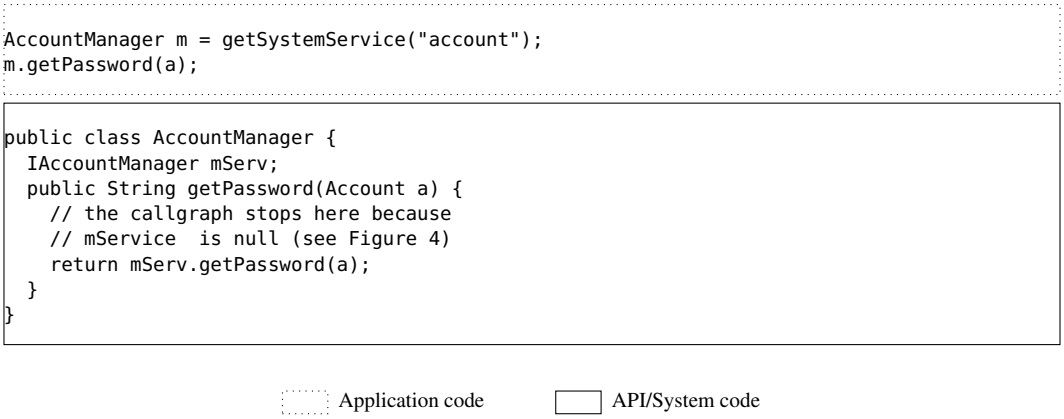


Figure 4.6: How Spark Discards Call Graph Edges Because of "null" Objects.

Importance of Service Initialization for Spark A Spark based approach does require proper initialization of the analyzed modules of the Android framework. The reason is that, as presented in Figures 4.6 and 4.7, skipping the initialization phase may result in important fields, containing references to system services for instance, to only point-to *null*. Spark does not generate edges for method calls on references which can only point to *null*.

Figure 4.6 represents a code snippet which retrieves an `AccountManager` object and calls method `getPassword()` on it. At this point `AccountManager`'s service reference `mServ` can only point to *null*. Thus, `mServ.getPassword()` cannot be executed and would not be represented in a field-sensitive call graph. In other words, Spark generates an edge for the `Ac-`

```
AccountManager m = getSystemService("account");
```

```
public class ContextImpl {
  public Object getSystemService(String ts) {
    if (ts.equals("account") {
      return getAccountManager();
    } else ...
  }
  private AccountManager getAccountManager() {
    IBinder b; IAccountManager mServ;
    // returns null
    b = ServiceManager.getService("account");
    // returns null because b is null
    mServ = IAccountManager.Stub.asInterface(b);
    // is null
    return new AccountManager(this, mServ);
  }
}

public class ServiceManager {
   // sCache initialized at boot time
  HashMap<String, IBinder> sCache;
  public static IBinder getService(String name) {
    // statically, getService() returns null
    return sCache.get(name);
  }
}
```

Figure 4.7: How Spark Propagates "null" Due to Initialization that is not Statically Visible.

`countManager` object but not for the service method call within it because the service reference (`mServ`) points to *null*.

This `AccountManager` object is created by the `Context` class as described in Figure 4.7. To simplify, only `AccountManager` objects are created in method `getSystemService()`. To create an `AccountManager` object a reference to the AccountManagerService is required. This reference is fetched through a call to `getService()`. However, since `ServiceManager` has not been initialized, `ServiceManager`'s sCache map is empty. So, `getService()` always returns *null*.

Service Initialization for Static Analysis As detailed in Appendix, system services are initialized in the `SystemServer` class. Methods from this class are not present in the call graph generated from entry points of the Android API since they are only called at system boot time.

To simulate system services initialization we create a static object and an initialization method for each concrete system service. Those objects are initialized by adding edges to the service initialization methods to the call graph. Moreover, the original bytecode is modified to replace calls to `getService` by a reference to the newly created static objects.

Manager Initialization for Static Analysis Android applications have two possibilities to communicate with system services

- The first possibility is to directly get a reference to the service[8] through the service manager and then to call remote procedures of the service
- The other possibility is to use another interface called `Manager`. The manager is created from the system `Context` class and has itself a reference to the service to directly communicate with it and acts as a proxy for the application (as show in Figure 4.6).

Managers are wrappers to ease communication with system services. We redirect calls to `getSystemService(String s)` to our own methods. To be able to do that, we used string analysis to compute a mapping between strings given to `getSystemService` and the code which initializes the corresponding manager. Each call to `getSystemService` is analyzed to extract the string parameter to know to which method it must be redirected. To each string corresponds one Manager and thus one method whose role is to initialize the manager.

We also provide our own `getService()` method that returns properly initialized services as presented in Section 4.5.3. All calls to the original `getService()` are redirected to our own methods. Method `getSystemService` returns a manager whereas method `getService()` returns an interface to a service.

The original bytecode of the Android framework is modified to reflect services and managers initialization. The resulting bytecode can be analyzed by any static analysis tool and is not specific to Soot.

Empirical Results

Spark-Android runs in 11 hours. Permission set sizes for entry points when running Spark-Android are described in Table 4.4. The number of entry points with a single permission is 471. Furthermore, 48 entry points have a permission set of two, 10 of 3 and three have more than three permissions. The total number of entry points is less than the one for CHA since abstract classes cannot be initialized with Spark. No method associated with those classes is represented in the set of entry point methods.

Permission Set	# entry points
with 0 permissions	42,895 (98.77%)
with 1 permissions	471 (1.08%)
with 2 permissions	48 (0.11%)
with 3 permissions	10 (0.02%)
with > 3 permissions	3 (< 0.01%)
	43,427 (100%)

Table 4.4: Spark-Android Permission Sets

[8] also called a *binder* to the service

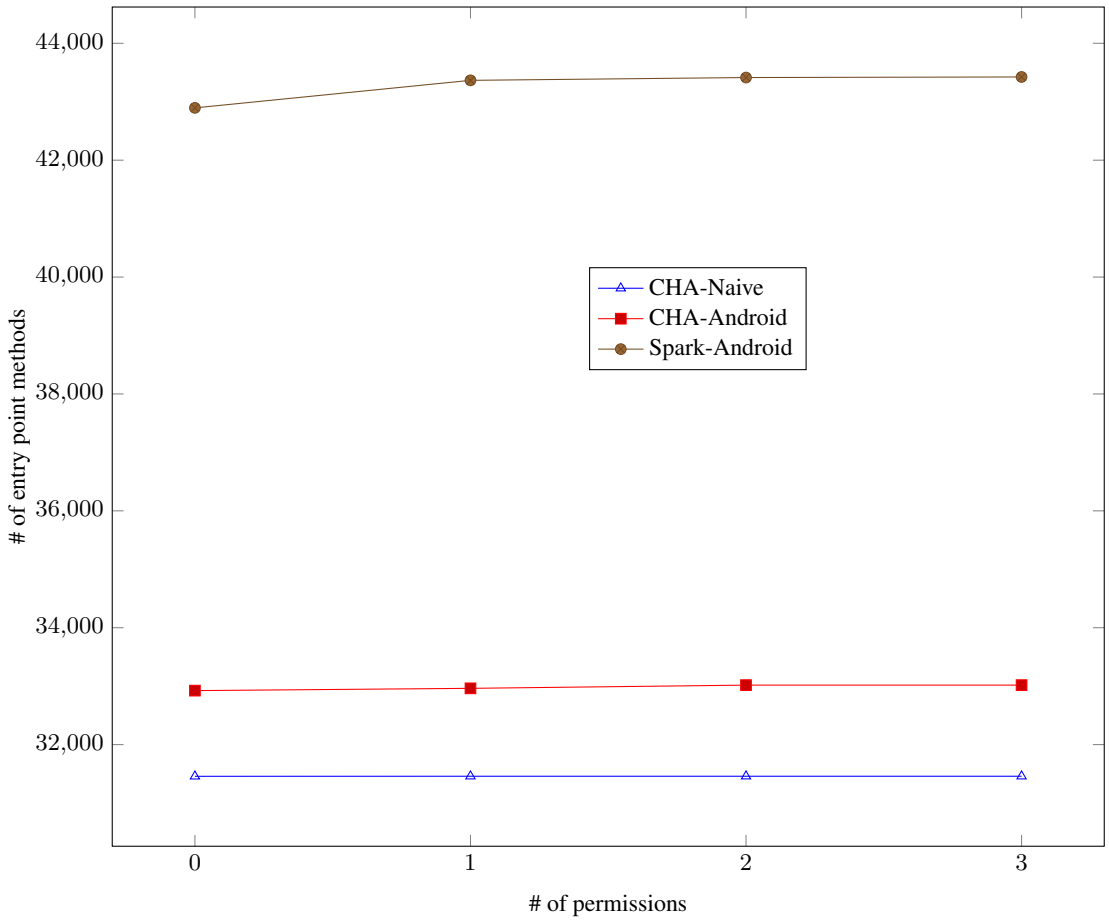


Figure 4.8: Cumulative Plot of the Number of Methods per Permission Set Size (The higher, the better).

4.5.4 Recapitulation

We have presented the core technical issues we encountered while implementing our approach. We think that those problems may arise in other permission-based platforms than Android, and that identifying them and their solutions can be of great help for future work. Last not but not least, those points are crucial for replication of our results.

Section 4.6 evaluates the CHA and Spark based analyses.

4.6 Discussion

How do the 6 analyses presented in section 4.5 perform compared to others? What are their limitations? This section answers those questions.

Permission set	Number of Methods
#API Methods in PScout	593
#API Methods in Spark and PScout	468 (100%)
Identical	289 (61.75%)
we find more precise permission checks	176 (37.60%)
we find more permission checks	3 (0.64%)

Table 4.5: Comparison between Our Results (Spark-based analysis) and Pscout's ones [6] (CHA-based analysis) using Android 4.0.1.

4.6.1 CHA versus Spark

Figure 4.8 is a cumulative plot of the number of entry point methods in function of their permission set size. By cumulative we mean that at each permission set size the number of methods is added to the number of methods at the previous permission set size. It first shows that the more precise an analysis is, the bigger the set of entry points with zero permission will be. This result reflects the fact that with precision, "false positive" edges are removed from the graph. Then, the plot (Spark-Android) highlights that, when only system services communication are handled, Spark yields the best results as it finds more methods with a permission set of one, two or three than all other analyses. Moreover, Spark never finds an entry point with more permission than CHA. It finds the same permission set (with one or more permission) than with CHA for 91 entry points. Spark finds a smaller permission set for 428 entry points.

4.6.2 Comparison with PScout

PScout [6] relies on a CHA based approach and generates a permission list for classes in the Android framework. We only consider classes of the Android 4.0.1 API. There are 593 methods in the results of PSCout that have more than one permission and 468 methods that are both in PSCout and Spark. Among those 468 methods, 289 (61.75%) have the same permission size in both PSCout and Spark and 176 (37.60%) have a smaller permission set size with our approach.

For instance, for method `KeyguardManager.exitKeyguardSecurely()`, PScout finds five permissions whereas Spark only one, `DISABLE_KEYGUARD`. The official documentation confirms that only one permission is required as well as the runtime data from Felt [53] . Spark also misses a permission for method `AudioManager.setMicrophoneMute(boolean)`. It is because we do not handle C/C++ native code where this permission check is done. Table 4.5 summarizes the results of this comparison. *Our analysis yields more precise results than a pure CHA-based approach.*

Interestingly we also find three methods (0.64%) for which our Spark approach finds more methods than PSCout's approach. We manually checked the `Vibrator` class where the involved methods are defined and there is a path to a method checking permission `WAKE_LOCK`. PScout probably did not correctly link those specific entry point methods to all methods they can reach, thus missing the `WAKE_LOCK` permission.

Permission set	Number of Methods
#Methods analyzed in [53]	1282
#Methods with HL perm. only	673
Identical	552 (82.3%)
we find more permission checks	119 (17.7%)
one more	118 (17.6%)
two more	1 (0.1%)
we find less permission checks	0 (0%)

Table 4.6: Comparison between Our Results and Felt et al.'s ones [53] (Based on Testing) using Android 2.2. Only methods with high-level permissions are considered.

4.6.3 Comparison with Felt et al.

Let us now compare our results obtained with static analysis [15] with the results of Felt et al.' obtained through testing [53]. Both extract a list of required permissions for each method of the Android 2.2 framework. Android 2.2 features 134 permissions, eight of them being low-level permissions that we do not analyze. Felt et al.'s results contain 673 methods mapped to high-level permissions. We analyze only 671 methods because 2 methods are related with application-specific objects provided in Felt's approach that are not available in our static analysis approach.

For a given method, we either find the same permission set, or a larger one. Our method never misses a permission that Felt et al. describe, this is piece of evidence of the soundness of our approach.

More precisely, we infer the same permission set per method signature for 552 methods (82.3% of commonly analyzed methods). There is one additional permissions for 119 methods (1 additional permission for 118 methods, 2 for 1 method). There is no method for which we miss a permission, Table 4.6 summarizes those results. Let us now discuss the discrepancy between our results.

The additional permissions are due to either analyzing irrelevant code or to missing input data in Felt et al.'s approach. In the latter case, we are able to find permissions that are checked within specific contexts that were not taken into account by the generated tests of Felt et al. For instance, `MOUNT_UNMOUNT_FILESYSTEMS` is only checked for method `MountService.shutdown()` if the media (storage device) is "*present not mounted and shared via USB mass storage*" (from the API documentation). Another permission, `READ_PHONE_STATE` is needed for method `CallerInfo.getCallerId()` only if the phone number passed in parameter is the voice mail number. Those test cases were not generated by Felt's testing approach. In real applications, test generation techniques cannot guarantee a comprehensive exploration of the input space.

To us, these findings are typical when comparing a static analysis approach against a testing one: static analysis sometimes suffers from analyzing all code (including debugging and dead code, or code run in specific runtime environments), but is strong at abstracting over input data. On the other hand, testing must simulate as close as possible the production environment, but is cursed to always miss very specific usage scenarios.

Those results highlight the complementarity between static analysis and testing in the context of permission inference. We think that the static analysis approach is complementary to the testing approach. Indeed, the testing approach yields an under-approximation which misses

some permission checks whereas the static analysis approach yields an over-approximation in which those missing permission checks are found. Using both approaches in conjunction would enable developers to obtain a lower and an upper bound of the permission gap. In particular, for a given Android applications, if both testing and static analysis approaches yield the same list of permissions, this strongly suggests that this list is the "correct" list of required permissions. As testing could miss permissions and static analysis may not model all Android specificities this cannot be a strong claim.

4.6.4 Soundness

We have shown in this chapter that the Android framework has many specificities that may threaten the soundness of static analysis. In this context, soundness refers to having no false negatives (no missed permission checks). Furthermore, the concept of soundness refers to a specific scope: in our cases, checks of high-level permissions inside Android services.

For CHA and Spark-based analysis, such as PScout, CHA-Android or Spark-Android, the manipulation of the call graph based on domain-specific knowledge (such as the bytecode redirection, and windowing system methods emptying) is sound if and only if all cases are envisioned. Given the complexity and scale of a framework such as Android, this completeness is hard to prove.

For Spark-based analysis, the analysis is sound if and only if the object and static fields are correctly initialized. Hence the analysis may be sound for some entry-points and unsound for others. For a framework such as Android, there is no oracle for formally answering those questions. However, for those entry points when the CHA-based results and the Spark-based results are identical it is a strong piece of evidence of soundness. For the rest, comparison with documentation or runtime data is required.

Finally, our results hold as far as there is no serious bug in the implementation of any part of the static analyses (e.g., entry point initialization and bytecode redirection), as well as in the glue and measurement code we wrote.

4.6.5 The Impact of Service Identity Inversion

A legitimate question to ask is whether or not service identity inversion has an impact on the resulting permission set. To answer that question, we ran Spark-Android with and without activating service identity inversion. Within the set of entry points that did not time out, two have a bigger permission set when service identity inversion is turned off. For instance, method `android.net.ConnectivityManager boolean requestRouteToHost(int,int)` has one more permission when service inversion is disabled `CONNECTIVITY_INTERNAL`. This permission is not required for the entry point according to the official documentation[9] which validates the usefulness of the service identity inversion building block.

Service inversion may only impact a few entry points but not taking it into account leads to wrong permission sets.

[9] `http://developer.android.com/reference/android/net/ConnectivityManager.html`

4.6.6 Limitations

Native Code

The Android framework is a real-world large-scale framework, featuring heterogeneous layers written in different languages. For Android 2.2 most Android permissions (126/134) are checked in the Android Java framework only. Our approach is complete for these 126 permissions, but incomplete for the eight permissions checked in native C/C++ code. These eight permissions are: BLUETOOTH_ADMIN, BLUETOOTH, INTERNET, CAMERA, READ_LOGS, WRITE_EXTERNAL_STORAGE, ACCESS_CACHE_FILESYSTEM and DIAGNOSTIC.

Reflection in the Framework

If the framework uses reflection, then the call graph construction is incomplete by construction. Fortunately, the Android framework uses reflection in only 7 classes. We manually analyzed their source code. Five of those classes are debugging classes. The `View` class uses reflection for handling animations. Finally, the `VCardComposer` uses reflection in a branch that is only executed for testing purpose. In all cases, the code is not related to system resources hence no permission checks are done at all. This does not impact the static analysis of the Android framework.

Dynamic Class Loading

The Java language has the possibility to load classes dynamically. Static analysis cannot deal with this since the loaded classes are only known at runtime. We found that eight classes of the Android system are using the `loadClass` method. After manual check, six of them are system management classes and are either not linked to permission checks (ex: instrumenting an application) or have to be accessed through a service. Two are related to the `webkit` package. They are used in the `LoadFile` and `PluginManager` classes. In both cases, permissions are checked *before* loading classes, and not inside the loaded classes. Thus, there is no missed permission enforcement point either.

Spark

Our model of the Android framework focuses on services and missed the initialization of other Android components (e.g., content providers). In other words Spark is sounds with regards with our model of Android components.

4.7 Computing Permission Gaps

We now have static analyses to compute the mapping between Android API methods and their required permissions. This section first presents a method to efficiently compute the required permission set and the corresponding permission gap (permissions declared but not used), if any. Then we present the results of an empirical study that show the existence of permission gaps in the wild.

4.7.1 A Calculus for Permission Analysis

This section describes the permission gap inference as a calculus on top of a boolean matrix algebra. Permission inference is at heart a reachability analysis (does the application reach a permission check?), the goal of this calculus is to "factorize" the static analysis, so as to be much more efficient.

Let app be an application. The *access vector* for app is a boolean vector AV_{app} representing the entry points of the framework under study. Thus, the length of vector AV is the number of entry points of framework $\mathcal{F}$. An element of the vector is set to *true* if the corresponding entry point is called by the application. Otherwise it is set to *false*. Let us consider a framework with four entry points (e_1, e_2, e_3, e_4), and an application foo that reached e_1, e_2 and e_3 but not e_4. AV_{app} reads:

$$AV_{foo} = (1, 1, 1, 0)$$

We define the *permission access matrix* M as a boolean matrix which represents the relation between entry points of the framework and permissions. The rows represent entry points of the framework and the columns represent permissions. A cell $M_{i,j}$ is set to *true* if the corresponding entry point (at row i) accesses a resource protected by the permission represented by column j. Otherwise it is set to *false*. For a framework with four entry points (e_1, e_2, e_3 and e_4) and three permissions (p_1, p_2 and p_3), the permission access matrix reads:

$$M = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

... meaning that e_1 and e_2 require permission p_1, e_3 requires no permission and e_4 requires permission p_2.

Let app and $\mathcal{F}$ be an application and a framework respectively. The inferred permissions vector, IP_{app}, is a boolean vector representing the set of inferred permissions for application app. By using the boolean operators AND and OR instead of arithmetic multiplication and addition in the matrix calculus, we have:

$$IP_{app} = AV_{app} \times M$$

A cell $IP_{app}(k)$ equals to *true* means that the permission at index k is required by app. Using AV_{app} and M from the previous examples, the inferred permissions vector for app is:

$$\begin{aligned} IP_{app} &= \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \\ IP_{app} &= \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \end{aligned}$$

... meaning that the application should declare and only declare permissions p_1.

4.7.2 Extraction of M and AV

The permission access matrix M is based on a static analysis of framework $\mathcal{F}$. As shown in Section 4.5, we first compute a call graph for every entry point of the framework and then to detect whether or not permission checks are present in the call graph. A permission enforcement point (PEP) is a vertex of a call graph whose signature corresponds to a system method that checks permission(s). Each PEP is associated with a list of required permissions $perms_{PEP}$. Matrix M is constructed as follows: it is a matrix of size (|entry points| $\times$ |high level permissions|); all elements of M are initialized to false; for each e_i that reaches one or more PEP, and for each permission j in $perms_{PEP}$, $M(i, j) = true$. In other terms, ***M is a condensed version of the reachability information that is latent in call graphs.***

Let's take the example of Figure 4.1 in Section 4.3. It shows a framework with four entry points (e_1, e_2, e_3, e_4), and three permissions (p_1, p_2, p_3). For every of those entry points a call graph is constructed. Three of those call graphs have a PEP node: e_1 and e_2 have PEP ck_1 which checks permission p_1 and e_4 has PEP ck_2 which checks permission p_2. On the figure a dashed arrow connects each PEP to the permission(s) it checks. The framework matrix is then matrix M presented above (see Section 4.7.1).

Extracting AV simply means listing the list of entry points of a framework $\mathcal{F}$ called by an application app. The application example in Figure 4.1 uses a single entry point, and $AV_{ex} = (1, 1, 1, 0)$.

4.7.3 Computing the Permission Gap

The permission gap is the difference between the permissions extracted from IP_{app} and the declared permissions $P_d(app)$. In Figure 4.1, using matrix M_{ex} and vector AV_{ex} of the example framework and application, we obtain a list of inferred permissions only containing p_1. If the application declares p_1 and p_2, the permission gap is $\{p_2\}$.

We ran our tool on two datasets of Android applications. The first comes from an alternative Android Market[10] and contains 1329 Android applications. For the second one, we consider the top 50 downloaded applications of all 34 top-level categories of the Official Android Market, as well as the top 500 of all applications and the top 500 of new applications (on February, 23^{rd} 2012). After removal of duplicates (the applications appearing in several rankings), the second dataset contains 2057 applications.

Alternative Android Market: We discard 587 applications that use reflection and/or class loading. Of the 742 remaining applications, 94 are declaring one or more permissions which they do not use. Consequently, *we identify a permission gap for 94 Android applications*. We define the "area of the attack surface" with respect to permission gaps, as the number of unnecessary permission. In all, among applications suffering from a permission gap, 76.6% have an attack surface of 1 permission, 19.2% have an attack surface of 2 permissions, 2,1% of 3 permissions and also 2,1% of 4 permissions.

Official Android Market: We discard 1378 applications that use reflection and/or class loading. On the 679 remaining applications, 124 are declaring one or more permissions which

[10] www.freewarelovers.com/android

they do not use. In all, among applications suffering from a permission gap, 64.5% have an attack surface of 1 permission, 23.4% have an attack surface of 2 permissions, 12.1% of 3 or more permissions.

To sum up, those results show that permission gaps exists, and that our approach allows developers to fix the declared permission list in order to reduce the attack surface of permission-based software.

4.8 Conclusion

In this chapter, we have used static analysis to extract permissions from the Android framework. At least three static analysis components must be put together in order to use Class Hierachy Analysis (CHA) and field-sensitive static analysis (Spark) for analyzing Android ' s permissions. Those are (1) a string analysis, (2) service identity inversion and (3) entry point and service initialization for Spark.

The approach has been fully implemented for Android, a permission-based platform for mobile devices. Our prototype implementation is able to automatically find 9562 Android framework entry points which check permissions. Concurrent work such as PScout [6] and Felt [53] confirm our results.

The approach has been fully implemented for Android, a permission-based platform for mobile devices. For end-user applications, our evaluation revealed that 94/742 and 35/679 applications crawled from Android application stores indeed suffer from permission gaps.

The security architecture of permission-based software in general and Android in particular is complex. In this chapter, we abstracted over several characteristics of the platform such as low-level permissions.

Chapter 5

Data Leakage in Android Applications

We have seen in the previous chapter that permissions protect sensitive data. Nevertheless, applications having the right permission(s) to access the data could leak the data. This is for instance the case with malware or application packaged with aggressive advertisement libraries. The objective of this chapter is to statically analyze Android applications to detect such leaks. Android applications are different from traditional Java applications. One of the most important differences is that Android applications are made of components. Analyzing Android applications to find leaks requires to link components that communicate together and to model every component. We developed IccTA to detect privacy leaks. It connects components at the code level to perform inter-component and inter-application data-flow analysis.

This chapter is based on work published in the following technical report:

• Li Li, **Alexandre Bartel**, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau and Patrick McDaniel: I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis, ISBN 978-2-87971-129-4, 2014.

5.1 Introduction

With the growing popularity of Android, thousands of applications (also called apps) emerge every day on the official Android market (Google Play) as well as on some alternative markets. As of May 2013, 48 billion apps have been installed from the Google Play store, and as of September 3, 2013, 1 billion Android devices have been activated [3]. Researchers have shown that Android apps frequently send the user's private data outside the device without the user's prior consent [141]. Those applications are said to *leak* private data. Android applications are made of different components; most of the privacy leaks are simple and operate within a single component. More recently, cross-component and also cross-app privacy leaks have been reported [136]. Analyzing components separately is not enough to detect such leaks. Therefore, it is necessary to perform an inter-component analysis of applications. Android app analysts could leverage such a tool to identify malicious apps that leak private data. For the tool to be useful, it has to be highly precise and minimize the false positive rate when reporting applications

leaking private data.

Privacy leaks. In this chapter, we use a static taint analysis technique to find privacy leaks, i.e., paths from sensitive data, called *source*s, to statements sending the data outside the application or device, called *sink*s. A path may be within a single component or cross multiple components and/or applications.

State-of-the-art approaches relying on static analysis to detect privacy leaks on Android apps mainly focus on detecting intra-component sensitive data leaks. CHEX [84], for example, uses static analysis to detect component hijacking vulnerabilities by tracking taints between sensitive sources and sinks. DroidChecker [29] uses inter-procedural Control-Flow Graph (CFG) searching and static taint checking to detect exploitable data paths in an Android application. FlowDroid [5] also performs taint analysis within single components of Android applications but with a better precision. In this chapter, we not only focus on intra-component leaks, but we also consider Inter-Component Communication (ICC) based privacy leaks, including Inter-Application Communication (IAC) leaks.

Other approaches use dynamic tracking to find privacy leaks. For instance, TaintDroid [45] leverages Android's virtualized execution environment to monitor Android apps at runtime in which it tracks how application leaks private information. CopperDroid [105] dynamically observes interactions between the Android components and the underlying Linux system to reconstruct higher-level behavior.

A dynamic approach must send input data to the app at runtime to trigger code execution. The input data may be incomplete and thus not execute all parts of the code. Furthermore, some code may only be executed if precise conditions are met at runtime such as a data. In this chapter, we focus on static analysis to avoid these drawbacks. The counterpart of static analysis is that it may yield an over-approximation since it analyzes all code even the one that could never be executed.

Static taint analysis for Android is difficult. Despite the fact that Android applications are mainly programmed in Java, off-the-shelf static taint analysis tools for Java do not work on Android applications. The tools need to be adapted mainly for three reasons. The first reason is that, as already mentioned, Android applications are made of components. Communications between components involve two main artifacts: *Intent Filter* and *Intent*. An *Intent Filter* is attached to a component and "filters" *Intents* that can reach the component. An *Intent* is used to start a new component by first dynamically creating an *Intent* instance, and then by calling a specific method (e.g. *startActivity*, *startService*) with the *intent* previously created as parameter. The *intent* is used either explicitly by specifying the new component to call, or implicitly by for instance only specifying the action[1] to perform. The launch of a component is performed by the Android system which "resolves" the matching between *Intent* and *Intent Filter* at runtime. This dynamic resolution done by the Android system induces a discontinuity in the control-flow of Android applications. This specificity makes static taint analysis challenging by requiring pre-processing of the code to resolve links between components.

The second reason is related to the user-centric nature of Android applications, in which a user can interact a lot through the touch screen. The management of user inputs is mainly done by handling specific callback methods such as the *onClick* method which is called when the user

[1] Such as `android.intent.action.VIEW` or `.CALL` or `.EDIT`

clicks on a button. Static analysis requires a precise model that stimulates users' behavior.

The third and last reason is related to the lifecycle management of the components. There is no *main* method as in a traditional Java program. Instead, the Android system switches between states of a component's lifecycle by calling callback methods such as *onStart*, *onResume* or *onCreate*. However, these lifecycle methods are not directly connected in the code. Modeling the Android system allows to connect callback methods to the rest of the code.

Our Proposal. The above challenges will unavoidably cause some discontinuities in the control-flow graph. To overcome these issues, we present an Inter-component communication Taint Analysis tool named IccTA[2]. IccTA allows a sound and precise detection of ICC and IAC links. This approach is generic and can be used for any data-flow analysis. In this chapter we focus on using IccTA to detect privacy leaks.

IccTA is based on three software artifacts: Epicc-IccTA, FlowDroid-IccTA and ApkCombiner.

Epicc-IccTA extends Epicc [92] which computes ICC links between Android components. Epicc-IccTA leverages Epicc to incrementally store the computed ICC links to a database for conveniently analyzing a large set of apps. FlowDroid-IccTA extends FlowDroid [5]. FlowDroid only finds privacy leaks within single components of Android applications but not between components.

FlowDroid-IccTA uses ICC links computed by Epicc to improve FlowDroid. Based on these computed links, FlowDroid-IccTA modifies Android applications' code to directly connect components to enable data-flow analysis between components. By doing this, we build a complete control-flow graph of the whole Android application. This allows propagating the context between Android components and yielding a highly precise data-flow analysis. To the best of our knowledge, this is the first approach that precisely connects components for data-flow analysis.

Finally, ApkCombiner helps analyzing multiple Android applications by combining multiple apps into one when there exist data-flows between these apps. This results in having a complete control-flow graph of the combined apps. This allows to propagate the context not only between components of a single app but also between components of different apps.

To verify our approach, we run IccTA on 3000 real-world Android applications and on 26 apps containing ICC-based privacy leaks that we developed. We have added these 26 applications to DroidBench [43], an open test suite for evaluating the effectiveness and accuracy of taint analysis tools specifically for Android apps. The 26 apps cover the top 8 used ICC methods illustrated in Table 5.1.

Contributions. To summarize, we present the following original contributions in this chapter:

- A novel methodology to resolve the ICC problem by directly connecting the discontinuities of Android apps at the code level.
- IccTA, a tool for inter-component data-flow analysis.
- An improved version of DroidBench with 26 new apps to evaluate tools detecting ICC-based privacy leaks.

[2]Our experimental results and IccTA itself are available at https://sites.google.com/site/icctawebpage.

Table 5.1: The top 8 used ICC methods†

ICC Method	Counts(#.)	Used Apps(#.)
startActivity	55802 (61.44%)	2765 (92.2%)
startActivityForResult	11095 (12.21%)	1980 (66.0%)
query	6606 (7.27%)	1601 (53.4%)
startService	3942 (4.34%)	1077 (35.9%)
sendBroadcast	3472 (3.82%)	790 (26.3%)
insert	2100 (2.31%)	615 (20.5%)
bindService	1515 (1.67%)	644 (21.5%)
delete	1238 (1.36%)	350 (11.7%)
Other ICC Methods	5058 (5.57%)	-
Total	90828 (100%)	-

† Methods with higher counts are selected when overload methods exist

- An empirical study to evaluate IccTA over an augmented version of the DroidBench test suite (available online[3]) and 3000 real-world Android applications.

The rest of this chapter is organized as as follows. Section 5.2 explains the necessary background on Android security. Section 5.3 gives a motivating example and Section 5.4 introduces some essential definitions. In Section 5.5, the paper discusses the implementation details of IccTA, while Section 5.6 evaluates IccTA. The limitations of IccTA are described in Section 5.7. Finally, Section 5.8 concludes the chapter.

5.2 Background

5.2.1 Android ICC Methods

As explained in Section 2.1, an Android application is made of basic units, called components, described in a special file, called *Manifest*, stored in the application. There are four types of components: a) Activities that represent the user interface and are the visible part of Android applications; b) Services which execute tasks in background; c) Broadcast Receivers that receive messages from other components or the system, such as incoming calls or text messages; and d) Content Providers which act as the standard interface to share structured data between applications.

Some specific Android system methods are used to trigger inter-component communication. We call them Inter-Component Communication (ICC) methods. Those methods take as parameter a special kind of object, called *Intent*, which specifies the target component(s). We perform a short study to compute the usage rate of ICC methods. We analyzed 3000 Android applications randomly selected from Google Play and other third party markets. Table 5.1 shows the top 8 most used ICC methods. The third column represents the number of apps using at least once the corresponding ICC method. The most used ICC method is `startActivity`, used to launch a new Activity component, which accounts for 59.2% of the total detected ICC methods.

[3] github.com/secure-software-engineering/DroidBench

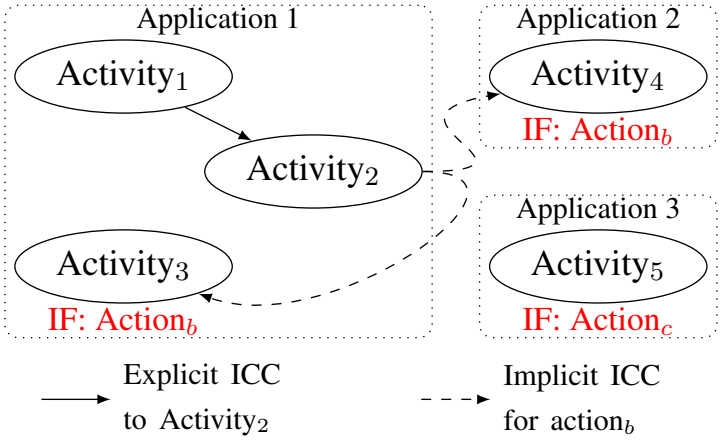


Figure 5.1: Explicit and Implicit ICC between Components of Android Applications.

All ICC methods[4] take at least one *Intent* in their parameters to specify the target component(s). There are two ways to specify ICC method's target components. The first one is by explicitly specifying them by setting the name of the target components through an *Intent*. The second one is by implicitly specifying them by setting the *action*, *category* and *data* fields of an *Intent*. In order to receive implicit *Intents*, target components need to specify an *Intent Filter* in their application's manifest file. Note that *Intents* can transfer data between components.

Again, we performed a short study on the 3000 apps to compute the ratio between explicit and implicit *Intents* for the `startActivity` ICC method. Among the 55,802 `startActivity` method calls, 27978 use explicit *intents* and 27824 use implicit *Intents*.

Figure 5.1 represents three Android apps made of Activity components. There is an explicit ICC from `Activity`$_1$ to `Activity`$_2$ in `Application 1`. There are two implicit ICCs from `Activity`$_2$ to `Activity`$_3$ in `Application 1` and from `Activity`$_2$ to `Activity`$_4$ between `Application 1` and `Application 2`. Note that the target components of implicit ICC, `Activity`$_3$ and `Activity`$_4$, have an *Intent Filter* with the same action and category value as the *Intent* used in `Activity`$_2$. Each time there is an ICC, there may be a flow of data between components and potentially a privacy leak.

5.2.2 FlowDroid

FlowDroid [5] is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications. The context-, flow-, field-, object-sensitives of FlowDroid are guaranteed by the precise call graph of Soot [75] and the IFDS [106] based data-flow analysis of Heros [22]. The sources and sinks used by FlowDroid are provided by SuSi [4], also an open sourced tool used to fully automatically classify and categorize Android sources and sinks.

[4]Except `Content Provider` related methods such as `query` or `insert`

Precise Modeling of Lifecycle

Android applications' components can be started independently and run in parallel. FlowDroid is path-insensitive (see Section 2.2.1) and assumes that components within an application can run in an any sequential order. A special main method, which considers all combinations of lifecycles (e.g., method `onPause()` which is executed when the corresponding Activity component is paused), callbacks (e.g., method `onClick()` which is executed when a button is clicked by the user) and entry points (e.g., method `onCreate()` which is executed when an Activity component is launched) of Android components is generated to model data-flows within the application.

IFDS Problem

FlowDroid does a taint-analysis and uses the IFDS framework. The analysis starts at statements assigning the result of a source method (e.g., `getDeviceId()`) in a variable. This variable is tainted since it contains data from the source method. The analysis then uses an idea introduces by Andomeda [129] and goes backward to find aliases of the tainted variable. During the forward analysis, if aliases reach the original assignemnt with the source, they are also tainted. Finally, if a tainted variable reaches a sink method, a leak is detected. As presented in Section 2.2.2, the analysis relies on flow functions applied on statements to compute the data-flow facts. In this analysis data-flow facts are the set of variables that are tainted at each program statement. For instance, the normal flow functions applied on statement `x = y` kills x if it was tainted and y is not tainted or progagate the taint of y to x if y was tainted before the statement.

Experimental Results

FlowDroid achieves 93% recall and 86% precision when detecting data leaks on DroidBench. FlowDroid has been mainly used to analyze data leaks within single components. However, with slight modifications, FlowDroid could also be used when multiple components are involved, i.e., for ICC analyses. Indeed, it is possible to use FlowDroid to compute paths for all individual components and then combines all those paths together, whether there is a real link or not between these components. A major drawback of this approach is that it yields many false positives. The next Section presents Epicc, a tool which statically resolves ICC links.

5.2.3 Epicc

Epicc [92] is a static analysis tool, also based on Soot and Heros, which computes ICC (Inter-Component Communication) links. In other words, it finds links from ICC methods to their target components.

IDE Problem

ICC methods take an object called *Intent* as parameter. This *Intent* object describes the destination component. Epicc statically analyze applications' code to reconstructs these objects at every statement calling an ICC method. Epicc reduces the discovery of ICC in Android to an

instance of the Inter-procedural Distributive Environment (IDE) problem [111]. An IDE problem propagates environments. In the case of Epicc, an environment can be seen as the mapping between one variable representing an object that is being reconstructed and the current value of that reconstructed object. For instance, when a new *Intent* object is created, the variable referencing this new object will be mapped to an empty Intent. During the analysis, environment transformers (on every edge of the supergraph) will update environments. For example, the environment transformer for statement `i.setAction("A1")` changes the enviroment of variable i by putting the string "A1" as the action value of the corresponding reconstructed object. Other objects used within *Intent* objects, such as *ComponentName* or *Bundle* objects, are reconstructed using the same process.

Experimental Results

Experiments show that when applied on a set of 1,200 Android applications, Epicc identifies 93% of all ICC links and finds ICC vulnerabilities with far fewer false positives than the next best tool. In this chapter we use the links generated by Epicc to improve the precision of FlowDroid ICC analyses. The following Section motivates our approach to find ICC leaks.

5.3 Motivating Example

This section motivates our approach and illustrates the problem we solve through a concrete example. This example is detailed in Figure 5.2, which presents code of `Application 1` introduced in Figure 5.1. The app has three Activity components represented by $\texttt{Activity}_1$, $\texttt{Activity}_2$ and $\texttt{Activity}_3$ classes. It also features `ButtonOnClickListener` a listener class used to handle button click events. $\texttt{Activity}_1$ registers a button listener for the *to2* button (lines 6-11) and $\texttt{Activity}_2$ registers one for the *to3* button (line 15).

When button *to2* and *to3* are clicked, the `onClick` method is executed and the user interface will change to $\texttt{Activity}_2$ and to $\texttt{Activity}_3$, respectively. In both cases, an `Intent` containing the device ID (lines 7 and 32), considered as sensitive data, is sent between two components by first attaching the data to the intent with the `putExtra` method (lines 9, and 35) and then by invoking either `startActivity` or `startActivityForResult` (lines 10 and 36). Note that Figure 5.2 exemplifies both the use of explicit and implicit intents. At line 8, the intent is created by explicitly specifying the target class ($\texttt{Activity}_2$). At line 34, only the intent action is specified with no explicit reference to the target.

In this example, `sendTextMessage` is directly executed when $\texttt{Activity}_2$ or $\texttt{Activity}_3$ is loaded since `onCreate` is the first method in the lifecycle of an `Activity`. It sends the data retrieved from the `Intent` as a SMS to the specified phone number.

In this code, two privacy leaks occur: one when button *to2* is clicked, the other when button *to3* is clicked. When *to2* is clicked, the device ID is transferred from $\texttt{Activity}_1$ to $\texttt{Activity}_2$ (line 10) and then $\texttt{Activity}_2$ sends it outside the application (line 18).

When button *to3* is clicked, the device ID is transferred from $\texttt{Activity}_2$ to $\texttt{Activity}_3$[5] (line 36). Actually, the device ID (the source) is retrieved in class `ButtonOnClickListener`

[5] As illustrated in Figure 5.1, $\texttt{Activity}_3$ has the appropriate *Intent Filter* to catch the implicit *Intent*

```
//TelephonyManager telMnger; (default)
//SmsManager sms; (default)
class Activity1 extends Activity {
 void onCreate(Bundle state) {
  Button to2 = (Button) findViewById(to2a);
  to2.setOnClickListener(new OnClickListener(){
   String id = telMnger.getDeviceId();
   Intent i = new Intent(Activity1.this,Activity2.class);
   i.putExtra("sensitive", id);
   Activity1.this.startActivity(i);
  });}}
class Activity2 extends Activity {
 void onCreate(Bundle state) {
  Button to3 = (Button) findViewById(to3a);
  to3.setOnClickListener(new ButtonOnClickListener(this));
  Intent i = getIntent();
  String s = i.getStringExtra("sensitive");
  sms.sendTextMessage(number,null,s,null,null);
 }
 void onActivityResult(int,int,Intent){
  //log all the Extras of Intent
 }}
class Activity3 extends Activity {
 void onCreate(Bundle state) {
  Intent i = getIntent();
  String s = i.getStringExtra("sensitive");
  sms.sendTextMessage(number,null,s,null,null);
 }}
class ButtonOnClickListener extends OnClickListener{
 //Activity act; (construct)
 void onClick(View view) {
  String id = telMnger.getDeviceId();
  Intent i = new Intent();
  i.setAction("test.ACTION"); //Action b
  i.putExtra("sensitive", id);
  act.startActivityForResult(i, 1);
 }}
```

Figure 5.2: A Motivating Example Code

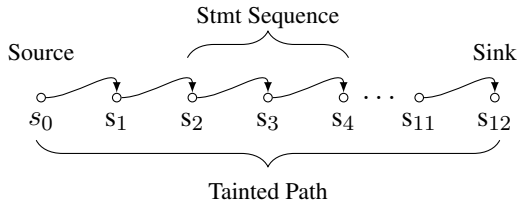


Figure 5.3: Representation of Statements, Source, Sink, Statement Sequence and Tainted Path.

instantiated by $\texttt{Activity}_2$. Finally, $\texttt{Activity}_3$ sends the device ID outside the application (line 27).

The sensitive data leaks described above crosses two components: they cannot directly be detected since there is no real code connection between `startActivity` and `onCreate` (lines 10 and 13) or between `startActivityForResult` and `onCreate` (lines 36 and 24). Section 5.5 describes our approach to connect components to analyze paths between components and even between applications.

5.4 Definitions

In order to better describe our approach, some Android and taint analysis related concepts need to be defined.

Control-Flow Graph (CFG) We detect data leaks by analyzing control-flow graphs of Android applications. An application CFG consists of a collection of method CFGs linked together according to how they call one another.

Source Method. A source method returns data considered as private from the user's point of view into the application code. For example, method `getDeviceId` (line 7[6]) is a source method returning the device ID.

Sink Method. A sink method sends data out of the application. For example, method `sendTextMessage` (line 27) is a sink method sending data to another phone using SMS. We use sources and sinks computed for Android by the SuSi tool [4].

ICC Method. An ICC method is used to trigger communication between two components. For example, method `startActivity` (line 10) is an ICC method which triggers component communication from $\texttt{Activity}_1$ to $\texttt{Activity}_2$.

Tainted Stmt. A tainted statement contains at least one tainted piece of data. For example, `i.putExtra("sensitive ",id)` (line 9) is a statement containing the tainted data `id`.

Tainted Stmt Sequence. A tainted stmt sequence is a flow-sensitive sequence of tainted stmt. For instance statements at line 9 and 10 form a tainted statement sequence.

Tainted Path. A tainted path is a tainted stmt sequence where 1) More than one stmt exist in the tainted path; 2) The first stmt contains a source method; 3) The last stmt contains a sink method. Tainted Stmt, Tainted Stmt Sequence and Tainted Path are illustrated in Figure 5.3.

[6]All the line numbers described in this section is referring to Listing 5.2

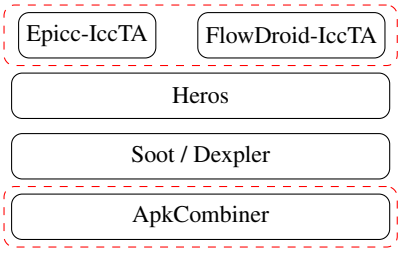


Figure 5.4: The architecture of IccTA

There are three types of tainted statement paths in Android: Intra-Component Communication, Inter-Component Communication (ICC) and Inter-Application Communication (IAC) based tainted paths.

Intra-Component Tainted Path. An intra-component tainted path is a tainted path within a component. In our motivating example, there is no intra-component tainted path. But if the `startActivity` call was replaced with a call to `sendTextMessage` which sends the device id out of the application, there would be an intra-component tainted path (line 7-10).

ICC-based Tainted Path. An ICC-based tainted path is a tainted path among two or more components, i.e., there is at least one ICC method in the path. In our motivating example, there is an ICC-based tainted path from source method `getDeviceId` in $\texttt{Activity}_1$ to sink method `sendTextMessage` in $\texttt{Activity}_2$ through the `startActivity` ICC method (line 10).

IAC based Tainted Path. An IAC based tainted path is a tainted path between two or among more applications, i.e., it has at least one ICC method between two components of different applications. There is no IAC based tainted path in our motivating example. But if the $\texttt{Activity}_4$ in Figure 5.1 sends the device id transferred from $\texttt{Activity}_2$ out of the application, then there is an IAC based tainted path from `Application 1` to `Application 2`.

Privacy Leaks. If a tainted path is detected, it means that a privacy leak has been found. In other words, some private data obtained from a *source* method can flow through the tainted path to a *sink* method.

5.5 IccTA

In this Section we describe IccTA, our tool to detect privacy leaks in Android applications. It uses static taint analysis to detect privacy leaks. The main challenge for this is to solve the discontinuities problem introduced by the Android system.

We present the architecture of IccTA in Figure 5.4 where new or modified component are surrounded by a dashed line. IccTA is the combination of Epicc-IccTA and FlowDroid-IccTA. Epicc-IccTA relies on Epicc to incrementally compute ICC links from Android apps. Both FlowDroid and Epicc are based on Soot [75] and Heros [22]. Soot is a framework to analyze Java-based applications. It uses the Dexpler [14] plugin to convert Android Dalvik byte code to Soot's internal representation called Jimple and relies on Spark [82] to build accurate call graphs. Heros is a scalable implementation of IFDS [107] and IDE [111], two frameworks to

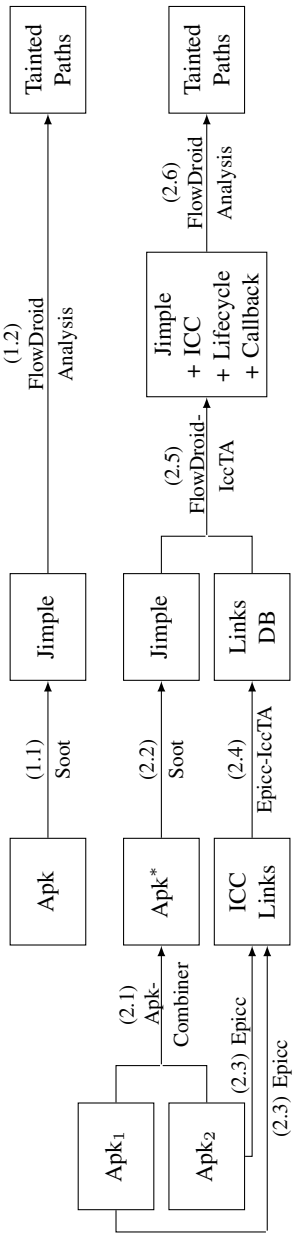


Figure 5.5: Overview of IccTA (down) and FlowDroid (up).

perform data-flow analysis. Analyzing multiple applications is done using ApkCombiner. It combines multiple apps to a single one to ease the analysis of IccTA.

Figure 5.5 is a comparison between IccTA and FlowDroid. FlowDroid (5.5 up) first converts the Android bytecode to Jimple in step (1.1). Then, in step (1.2), it analyzes the Jimple code to detect tainted paths in single Android components.

IccTA (5.5 down) can analyze one or multiple Android applications. If more than one application is analyzed, it uses ApkCombiner to merge the Android applications in a single application in step (2.1). The Android application's bytecode is then converted to Jimple in step (2.2). In parallel, Epicc-IccTA analyzes all the input applications (Apk_1 and Apk_2 in the Figure) to generate ICC Links in step (2.3) and stores the results to a database in step (2.4). IccTA uses ICC links generated by Epicc-IccTA to connect Android components in the Jimple code in step (2.5). Steps (2.2) and (2.6) correspond to FlowDroid's steps (1.1) and (1.2): the Jimple code is updated to take into account lifecycles and callbacks of components and the taint analysis is launched to generate a list of tainted paths.

5.5.1 FlowDroid-IccTA: Reducing the ICC problem to an Intra-Component Problem

Since there is no direct code connection between two Android components, FlowDroid cannot detect ICC-based privacy leaks with precision. In this section, we describe how FlowDroid-IccTA reduces the ICC problem to an intra-component problem on which FlowDroid can perform an highly precise data-flow analysis. Our approach instruments the *Jimple* code of Android applications to connect components directly in the code.

As mentioned in the introduction, there are three types of discontinuities in Android: (1) ICC methods, (2) lifecycle methods and (3) callback methods. We first describe how FlowDroid-IccTA tackles ICC methods in Section 5.5.1. Then, we detail how FlowDroid-IccTA resolves lifecycle and callback methods in Section 5.5.1. Finally, using our motivating example of Listing 5.2, we illustrate the code instrumentation process in Section 5.5.1.

ICC Methods

(A)
```
// modifications of Activity1
~~Activity1.this.startActivity(i);~~
IpcSC.redirect0(i);
```

(B)
```
// creation of a helper class
class IpcSC {
 static void redirect0(Intent i) {
  Activity2 a2 = new Activity2(i);
  a2.dummyMain();
 }
}
```

(C)
```
// modifications in Activity2
public Activity2(Intent i) {
  this.intent_for_ipc = i;
}
public Intent getIntent() {
 return this.intent_for_ipc;
}
public void dummyMain() {
 // lifecycle and callbacks
 // are called here
}
```

Figure 5.6: Code Modifications to Handle ICC Communication between $Activity_1$ and $Activity_2$. The `startActivity` ICC method is replaced (A) by a call to code that instantiates and calls the "main" method of $Activity_2$ (B). The target component class is updated to handle Intent objects directly, by modeling the Android system behavior (C).

As shown in Figure 5.5, the ICC problem is solved at step 2.5. This is where the *Jimple* code is updated by FlowDroid-IccTA to connect components. This code modification is required for all ICC methods (listed in Table 5.1). We detail these modifications for the two most used ICC methods: `startActivity` and `startActivityForResult`. We handle ICC methods for *Service*s and *Broadcast Receiver*s in a similar way.

StartActivity. Figure 5.6 shows the code transformation done by FlowDroid-IccTA for the ICC link between $\texttt{Activity}_1$ and $\texttt{Activity}_2$ of our motivating example. FlowDroid-IccTA first creates a helper class named `IpcSC` (B in Figure 5.6) which acts as a bridge connecting the source and destination components. Then, the `startActivity` ICC method is removed and replaced by a statement calling the generated helper method (`redirect0`) (A).

In (C), FlowDroid-IccTA generates a constructor method taking an `Intent` as parameter, a `dummyMain` method to call all related methods of the component (i.e., lifecycle and callback methods) and overrides the `getIntent` method. An Intent is transferred by the Android system from the caller component to the callee component. We model the behavior of the Android system by explicitly transferring the Intent to the destination component using a customized constructor method, $\texttt{Activity}_2\texttt{(Intent i)}$, which takes an `Intent` as its parameter and stores the Intent to a newly generated field `intent_for_ipc`. The original `getIntent` method asks the Android system for the incoming Intent object. The new `getIntent` method models the Android system behavior by returning the Intent object given as parameter to the new constructor method.

The helper method `redirect0` constructs an object of type $\texttt{Activity}_2$ (the target component) and initializes the new object with the `Intent` given as parameter to the helper method. Then, it calls the `dummyMain` method of $\texttt{Activity}_2$.

To resolve the target component, i.e., to automatically infer what is the type that has to be used in the method `redirect0` (in our example, to infer $\texttt{Activity}_2$), Flowdroid-IccTA uses the ICC links computed by Epicc-IccTA. Epicc-IccTA resolve the target component not only for explicit *intents*, but also for implicit *intents*. Therefore, there is no difference for Flowdroid-IccTA to handle explicit or implicit *intent* based ICCs.

StartActivityForResult. There are some special ICC methods in Android, such as `startActivityForResult`. A component C_1 can use this method to start a component C_2. Once C_2 finishes running, C_1 runs again with some result data returned from C_2. The control-flow mechanism of `startActivityForResult` is shown in Figure 5.7. There are two discontinuities: one from (1) to (2), similar to the discontinuity of the `startActivity` method, and the other from (3) to (4).

The `startActivityForResult` ICC method has a more complex semantic compared to common ICC methods that only trigger one-way communication between components (e.g., `startActivity`). Figure 5.8 shows how the code is instrumented to handle the `startActivityForResult` method in our motivating example. To stay consistent with common ICC methods, we do not instrument the `finish` method of $\texttt{Activity}_3$ to call `onActivityResult` method. Instead, we generate a field `intent_for_ar` to store the *Intent* which will be transferred back to $\texttt{Activity}_2$. The *Intent* that will be transfered back is set by the `setResult` method. We override the `setResult` method to store the value of *Intent* to `intent_for_ar`. The helper method `IpcSC.redirect0` does two modifications to link these two components

directly. First, it calls the `dummyMain` method of destination component. Then, it calls the `onActivityResult` method of the source component.

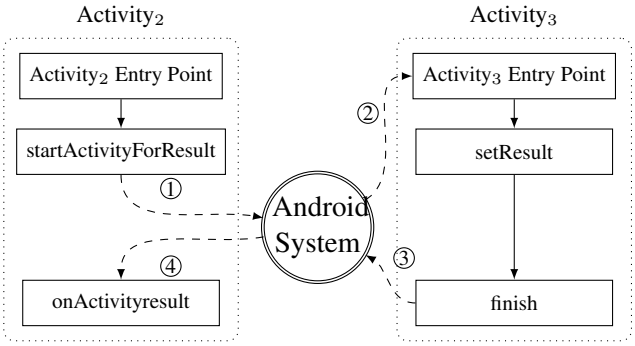


Figure 5.7: The control-flow mechanism of `startActivityForResult`

Lifecycle and Callback Methods

```
(A) act.startActivityForResult(i);
    IpcSC.redirect0(act, i);

    void setResult(Intent i) {
     this.intent_for_ar = i;
     a2.dummyMain();
(C) }
    public Intent getIntentFAR() {
     return this.intent_for_ar;
    }
```

```
class IpcSC {
 static void redirect0(Activity a2,
                       Intent i) {
  Activity3 a3 = new Activity3(i);
(B) a3.dummyMain();
  Intent retI = a3.getIntentFAR();
  a2.onActivityResult(retI);
 }
}
```

Figure 5.8: An Example about running FlowDroid-IccTA to `startActivityForResult` ICC method. (A) represents the modified code of `ButtonOnClickListener` and (C) the modified code of `Activity`$_3$. (B) is the glue code connecting `ButtonOnClickListener` and `Activity`$_3$`.` Some method parameters are not represented to simplify the code.

One challenge when analyzing Android applications is to tackle the callback methods and the lifecycle methods of components. There is no direct call among those methods in the code of applications since the Android system handles lifecycles and callbacks. For callback methods, we need to take care of not only the methods triggered by the User Interface (UI) events (e.g., `onClick`) but also of callbacks triggered by Java or the Android system (e.g., the `onCreate` method). In Android, every component has its own lifecycle methods. To solve this problem, IccTA generates a `dummyMain` method for each component in which we model all the methods mentioned above so that our CFG based approach is aware of them. Note that FlowDroid also generates a `dummyMain` method, but it is generated for the whole app instead of for each component like we do.

The CFG of instrumented motivating example

Figure 5.9 represents the CFG of the instrumented motivating example presented in Listing 5.2. In the CFG, `getDeviceId` is a *source* method in the anonymous `OnClickListener` class (line

6) called by $\texttt{Activity}_1$. Method `sendTextMessage` is a *sink* in $\texttt{Activity}_2$. There is an intra-component tainted statement path from the *source* method to *sink* method (represented by edges 1 to 12).

Figure 5.9 also shows that IccTA builds a precise cross-component control-flow graph. Since we use an technique instrumenting the code to build the CFG, the context of a static analysis is kept between components. This enables IccTA to analyze data-flows between components and thereby enables IccTA to have a better precision than existing approaches.

5.5.2 ApkCombiner: Reducing an IAC problem to an ICC problem

In Android, Inter-Application Communication (IAC) is similar to Inter-Component Communication (ICC). Indeed, IAC also relies on component communication, except that the source component and the destination component belong to different applications. If we can connect applications, an *IAC Problem* becomes a standard *ICC Problem*.

Analyzing Multiple Applications. As shown in Figure 5.5, FlowDroid can only analyse one application at a time. Therefore, we develop a tool, *ApkCombiner*, to combine multiple apps into one. ApkCombiner combines all the parts of Android apps including bytecodes, assets, manifest and all the resources. Then, we use IccTA to analyze the combined app to compute IAC based privacy leaks. As FlowDroid-IccTA handles the combined application as a single applications, it only detects ICC-based privacy leaks. To distinguish ICC leaks from IAC leaks, IccTA checks if all statements of the tainted path belong to the same application or not.

Reducing the Number of Combined Apps to Analyze. In practice, when increasing the number of applications to analyze, and if all those applications are combined with ApkCombiner, the processing time and memory requirement of FlowDroid-IccTA also grows. To solve this problem, we need to decrease the number of Android apps to combine. Our solution is to build an IAC graph, where a node is an application and an edge a link, to represent the dependencies between applications. The idea behind being that if there is no link between two applications there is no need to combine them.

The IAC graph is made up of small independent IAC (sIAC) graphs (connected components). Given a sIAC graph, ApkCombiner combines all the nodes (apps) in it into one app, then IccTA extracts leaks from the resulting app. However, in some case, if a sIAC graph still contains a lot of nodes. This will also limit our approach to be scalable. Our solution is to limit the length (how many apps are involved) of an IAC leak[7]. For example, If a sIAC graph contains 10 nodes (where $\texttt{A}_i$ is connected to $\texttt{A}_{i+1}$, $i \in \{1, 9\}$) and the length limitation is set to five. Then, the sIAC graph is split into five sIACs (e.g., one sIAC is from `A2` to `A6`) that IccTA can analyze. The trade-off limitation length enables our approach to become scalable.

Another good point of building an IAC graph is that new applications can be added to the graph in an iterative and incremental manner. When new apps are involved, we only run them against Epicc-IccTA and add them to the existing IAC graph. We do not need to run the previously computed apps again when adding the new apps to the IAC graph.

In short, by building an IAC graph, the original set of Android applications is split into multiple small sets that IccTA can analyze.

[7] In practice we have not seen a leak going through more than 2 apps.

5.6 Evaluation

Our evaluation addresses the following research questions:

RQ1 How does IccTA compare to commercial taint-analysis tools for Android and FlowDroid in terms of precision and recall?

RQ2 Can IccTA find leaks in real-world applications and how fast is it?

RQ3 How do IccTA compare to other academic ICC leak detection approaches?

5.6.1 RQ1: IccTA vs FlowDroid and Commercial Tool

We evaluate and compare IccTA with FlowDroid and IBM AppScan Source 9.0 on DroidBench to test for ICC and IAC leaks. Unfortunately, we were unable to compare IccTA to other static analysis tools as their authors did not make them available.

DroidBench. DroidBench [43] is a set of hand crafted Android applications for which all leaks are known in advance. The fact of knowing all leaks in the applications is called the *ground truth* and is used to evaluate how well static and dynamic security tools find data leaks. DroidBench version 1.2 contains 64 different test cases with different privacy leaks. However, all the leaks in DroidBench are intra-component privacy leaks. Thus, we developed 26 apps and 23 test cases to extend DroidBench with ICC and IAC leaks. A test case is applied on one application to test for ICC and on two applications to test for IAC. In total, 18 apps contain inter-component privacy leaks and 6 apps contain inter-app privacy leaks. The new set of test cases covers each of the top 8 ICC methods in Table 5.1. Moreover, among the 26 new apps, two of them do not contain any privacy leaks. If a tool detects privacy leaks on these two apps, the detected leaks are false alarms. Finally, for each test case application we add an unreachable component containing a sink. These unreachable components are used to flag tools that do not properly construct links between components.

The 23 test cases are listed in the first column of Table 5.2.

IccTA. We run IccTA on all the 23 test cases. The results are shown in Table 5.2. IccTA successfully passes 18 test cases, with 17 test cases containing 19 privacy leaks and one test case (`startActivity5`) with no leak.

Among the detected privacy leaks, three of them are IAC based privacy leaks and the remaining ones are ICC-based privacy leaks. In the `startActivity5` test case, the source component uses an implicit intent with data type *text/plain* to start another activity. However, no other activity in this test case declares that it can receive an intent with data type *text/plain*. That means there is no connection among the components in `startActivity5` test case. As IccTA takes into consideration the data type of an intent it does not report any privacy leak for this test case.

The `startActivity4` test case does not contain any leaks. However, IccTA does report a false warning. The reason is that the source component uses an implicit intent with an URI to start another activity. Since IccTA relies on Epicc which does over-approximate URIs links, it reports a false leak.

The current version does not take into account `Content Providers`. This is why IccTA misses leaks for the `insert1`, `delete1`, `update1`, and `query1` test cases. All the four test

cases are related to `Content Provider`.

FlowDroid. FlowDroid has been evaluated on the first version of DroidBench in [5]. In table 5.2, we present the results of FlowDroid on the new 23 test cases. As already explained, FlowDroid has been initially proposed to detect leak in single Android component. However, we can use FlowDroid in a way that it computes paths for all individual components and then combines all those paths together (whatever there is a real link or not). As a result, we expect that FlowDroid detects most of the leaks but yields several false positives. Results of Table 5.2 confirm this expectation: FlowDroid has a high recall (69.6%) and a low precision (23.9%). FlowDroid misses three more leaks than IccTA in `bindService{2,3,4}`. After investigation, we discover that FlowDroid does not consider some callback methods for service components.

AppScan. AppScan Source 9.0 requires a lot of manual initialization work since it has no default sources/sinks configuration file and is unable to analyze Android applications without specifying the entry points of every components. We define the `getDeviceId` and `log` methods, that we always use in DroidBench for ICC and IAC leaks, as source and sink, respectively. We also add all components' entry point methods (such as `onCreate` for activities) as callback methods so AppScan knows where to start the analysis. AppScan is natively unable to detect inter-component data-flows and only detects intra-component flows. AppScan has the same drawbacks as FlowDroid and should have a high recall and low precision on DroidBench. We use an additional script to combine the flows between components. As expected AppScan's recall is high (56.5%) and its precision low (21.0%). Compared to FlowDroid, AppScan does worse. Indeed, AppScan does not correctly handle `startActivityForResult` and thus misses leaks going through methods receiving results from the called activities in `startForResult{2,3,4}`.

Conclusion. IccTA outperforms both the commercial taint-analysis tool AppScan 9.0 and FlowDroid in terms of precision and recall.

5.6.2 RQ2: IccTA and Real-World Apps

We run the experiments on a Core i7 CPU running a Java VM with 8 Gb of heap. To evaluate our approach, we use IccTA to analyze 3000 Android apps downloaded from the Google Play market as well as some third-party markets (e.g., wandoujia). IccTA process 3000 apps in about 100 hours. IccTA does not detect any leak for 2575 (85.83%) applications. IccTA reports 425 applications containing privacy leaks. Among the 425 apps, 411 apps only contain intra-component leaks and 14 apps contain at least one ICC leak. From those 14 apps, 13 contain both intra-component leaks and ICC leaks. IccTA detects 6989 IAC links. Among those IccTA detects one IAC leak. This result indicates that components do communicate and share data, but it is rare that an inter-application leak occurs.

For intra-app leaks, IccTA detects 5986 leaks in the 425 apps. Among the detected leaks, 147 (2.5%) are ICC privacy leaks. We manually check the 147 reported ICC leaks and found out that 17 (11.6%) are false positives. In other words, IccTA achieves a precision of 88.4% on real-word apps. The false positives comes from Epicc that generates false positives for links between components.

We summarize the frequently used *source* methods and *sink* types (Java classes) in Table 5.3 from the 425 apps having at least one leak. Note that we only count such *source* and *sink*

Table 5.2: DroidBench test results

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported
† C/A: # of Components / # of Applications

Test Case (C/A)†	**FlowDroid**	**AppScan**	**IccTA**
Inter-Component Communication			
startActivity1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
startActivity2 (4/1)	⊛ (4 ⋆)	⊛ (4 ⋆)	⊛
startActivity3 (6/1)	⊛ (32 ⋆)	⊛ (32 ⋆)	⊛
startActivity4 (3/1)	⋆ ⋆	⋆ ⋆	⋆
startActivity5 (3/1)	⋆ ⋆	⋆ ⋆	
startForResult1 (3/1)	⊛	⊛	⊛
startForResult2 (3/1)	⊛	○	⊛
startForResult3 (3/1)	⊛ ⋆	○	⊛
startForResult4 (3/1)	⊛ ⊛ ⋆	⊛ ○	⊛ ⊛
startService1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
startService2 (3/1)	⊛ ⋆	⊛ ⋆	⊛
bindService1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
bindService2 (3/1)	○	○	⊛
bindService3 (3/1)	○	○	⊛
bindService4 (3/1)	⊛ ⋆ ○	⊛ ⋆ ○	⊛ ⊛
sendBroadcast1 (3/1)	⊛ ⋆	⊛ ⋆	⊛
insert1 (3/1)	○	○	○
delete1 (3/1)	○	○	○
update1 (3/1)	○	○	○
query1 (3/1)	○	○	○
Inter-App Communication			
startActivity1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
startService1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
sendBroadcast1 (4/2)	⊛ ⋆	⊛ ⋆	⊛
Sum, Precision, Recall and F_1			
⊛, higher is better	16	13	19
⋆, lower is better	51	49	1
○, lower is better	7	10	4
Precision $\circledast/(\circledast + \star)$	23.9%	21.0%	95.0%
Recall $\circledast/(\circledast + \bigcirc)$	69.6%	56.5%	82.6%
F_1 $2\circledast/(2\circledast + \star + \bigcirc)$	0.36	0.31	0.88

methods that appear in the detected leaks. The most used *source* method is `openConnection` and it is used 601 times in 169 apps. The most used *sink* types is `Log` and it is used 2755 times in 261 apps. The reason why we study *sink* types instead of *sink* methods is that there are a lot of *sink* methods in a same *sink* type. Take the Log *sink* type as an example, there are eight *sink* methods which log the private data to disk.

Let us describe in details three leaks, one for each type of leak.

Intra-component leak: `bz.prana.myphonelocator.` IccTA detects an intra-component privacy leak starting from the `getLongitude` source method in method `onLocationChanged`

Table 5.3: The top 5 used source methods and sink types

Method/Type	Counts(#.)	Detail
Source Methods		
openConnection	601	http connection
getLongitude	514	longitude
getLastKnownLocation	448	Location
getDeviceId	403	IMEI or ESN
getCountry	265	country code
Sink Types		
Log	2755	error or warn
URL	821	execute
SharedPreferences	717	putInt, putString
Message	339	sendTextMessage
File	9	write(string)

of class `.SMSReceiver$MyLocationListener`[8]. The location is sent out of the app through SMS by the `sendTextMessage` *sink* method in method `smsReply` of class `.SMSReceiver`. The app is designed to send the location outside the device through SMS. However, to distinguish the intention of detected privacy leaks is out of scope of this chapter. We take it as our further work.

ICC leak: `com.dikkar.ifind.` An ICC-based privacy leak is detected by IccTA on this application. In method `onLocationChanged` of class `.iFindPlaces`, the `getLongitude` *source* method is called and returns the location of the Android phone. Then, the location is transferred to another component, `.PlaceDetail`, where method `b` of class `j` is called. In method `b`, a *sink* method `Log.d` logs the location into disk with `ServiceHandler` tag name. To verify the detected leaks, we developed an Android application named LogParser. By giving the permission `android.permission.READ_LOGS` [9], LogParser reports all the locations logged by `Find Places`.

IAC leak: `com.bi.mutabaah.id to jp.benishouga.clipstore.` An IAC leak is reported by IccTA between application `com.bi.mutabaah.id` and application `jp.benishouga.clipstore`. The *source* method `findViewById` is called in component `com.bi.mutabaah.id.activity.Statistic`, where the data of a `TextView` is obtained. Then, the data is stored into an intent with two extras named `extra.SUBJECT` and `extra.TEXT`. After that, the `startActivity` ICC method is called to send the data to the `jp.benishouga.clipstore` application, which extracts the data from the intent with the same extra names and writes all the data to a file.

Conclusion. IccTA finds leaks in real-world apps in a reasonable amount of time. Nevertheless, IccTA only detects a single IAC leak. This is an indication that inter-application leaks are rare.

[8]The package name is omitted when the class name starts with the package name

[9]Starting from Android 4.1 it is no more granted to regular apps, but it can still be granted to either vendor apps or apps running on rooted phones.

5.6.3 RQ3: Compare with Other academic Tools

We identify two academic tools able to deal with ICC leaks: SCanDroid [56] and SEFA [136]. However, ScanDroid fails to report any leaks and SEFA is not available. As a result, we were not able to evaluate them on DroidBench.

To answer the research question, we focus and discuss some key aspects of the various approaches. SCanDroid and SEFA both use a *path matching* approach, which computes paths for all individual components and then combines some paths together, the decision of combining two paths or not is given by a matching algorithm. A *path matching* approach presents at least two main drawbacks.

First, even if the taint analysis is done for each component, the context of the analysis is lost when SCanDroid and SEFA combine the taint paths, since the analysis is performed before the combination of the paths. IccTA does not present this problem because it connects the components at the code level and then performs the analysis. Thus, it keeps the data-flow between two components. Losing the context decreases the precision of the tool. Indeed, an `Intent` can carry data, i.e., it may contain a lot of extras key/value pairs but only part of them are sensitive. A precise tool needs to distinguish them to avoid false positive. For a path matching approach, it is not easy to distinguish them because they do not keep the state of `Intent` when matching two available paths.

Second, some specific ICC methods such as `startActivityForResult` are difficult to handle with a matching algorithm. It will become even worse when the special ICC methods exist in a class which is invoked by multiple components. Suppose a component $\texttt{Activity}_4$ also uses the class `ButtonOnClickListener` shown in Listing 5.2 to communicate with other components. We present this scenario in Figure 5.10. A path matching approach would first find a path from the `startActivityForResult` ICC method to $\texttt{Activity}_3$. After the `finish` method of $\texttt{Activity}_3$ is called, the `onActivityResult` method of the source component is invoked by the Android system. The problem is that it is difficult to know which component ($\texttt{Activity}_2$ or $\texttt{Activity}_4$) is the source because they both use the same class `ButtonOnClickListener` where the Intent is created. In fact, It is very difficult to statically resolve this problem since it is caused by the mechanism of dynamic binding of Android (or Java). In our approach, IccTA resolves this problem by explicitly calling the appropriate `onActivityResult` method (see Figures 5.7 and 5.8) of the source component ($\texttt{Activity}_2$ or $\texttt{Activity}_4$) thanks to the helper class `IpcSC`.

Conclusion. Even if we were not able to evaluate state-of-the-art tools detecting ICC leaks (SCanDroid and SEFA), IccTA seems to be more precise mainly because it keeps the context between components unlike *path matching* approaches.

5.7 Limitations

In this section, we discuss the limitations of IccTA.

FlowDroid. IccTA is based on FlowDroid to perform static taint analysis and thereby shares the same limitations of FlowDroid. IccTA resolves reflective calls only if their arguments are string constants. It is also oblivious to multi-threading. We experienced that FlowDroid cannot

properly analyze some apps (too much memory consumption or hangs). We start by analyzing a set of 5000 and keep only 3000 apps that work with FlowDroid. Running IccTA on a big server could significantly decrease the number of falling analysis. Moreover, we are very confident that the next release of FlowDroid will resolve this problem.

Epicc. IccTA relies on Epicc to compute links between components. Since Epicc does not handle URIs, it fails to find ICC links for `ContentProvider` and yields false positives for the other three types of components when they communicate using URIs. In practice the number of links is huge due to the false positives. We check the links (intents and intent filters) and only keep the ones not using URIs.

IccTA. At the moment IccTA does not handle some rarely used ICC methods such as `sendActivities` or `sendOrderedBroadcastAsUser`. Data send between component with an intent, is represented as key/value pairs. When a tainted data is put in the intent, IccTA taints all key/value pairs. This could result in false positives if a tainted data is put in an intent and, in the receiving component, a non-tainted data is retrieved from the intent and flows to a sink.

Native Code. Some Android application are packaged with native code. IccTA only analyzes the dex file containing the Dalvik bytecode.

5.8 Conclusion

This chapter addresses the major challenge of performing data-flow analysis across multiple components or multiple applications. We have presented IccTA[10], an ICC-based taint analysis tool able to perform such analysis. In particular, we demonstrate that IccTA can detect ICC-based privacy leaks by providing a highly precise control-flow graph through instrumentation of the code of applications. Unlike previous approaches, IccTA enables a data-flow analysis between two components and adequately models the lifecycle and callback methods to detect ICC-based privacy leaks. When running IccTA on DroidBench, it reaches a precision of 95.0%. When running IccTA on three thousands applications randomly selected from the Google Play store as well other third-party markets, it detects 130 inter-component based privacy leaks in 12 applications.

[10]Our experimental results and IccTA itself are available at https://sites.google.com/site/icctawebpage.

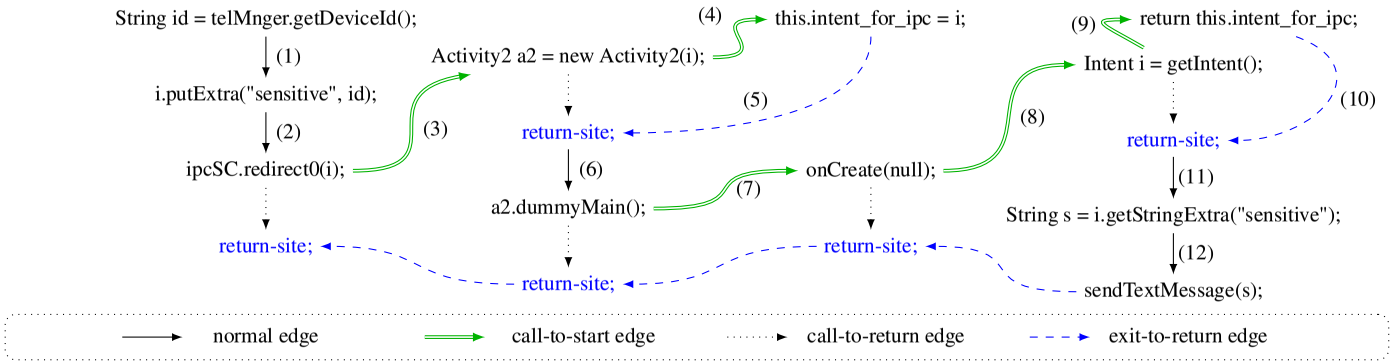


Figure 5.9: The control-flow graph of the instrumented motivating example

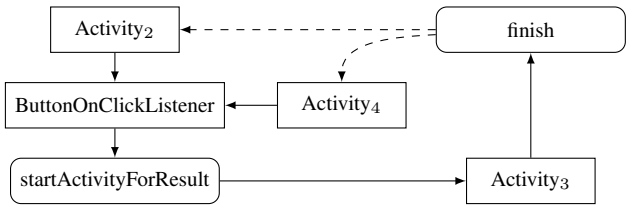


Figure 5.10: The problem of using path matching approach for `startActivityForResult`

Chapter 6

In Vivo: Dynamic Approaches for Security and Privacy

Analyzing Android applications statically enables to find security issues such as the GPS coordinates leaking out of the device. However, static analyses do not run directly on users' devices and thus do not take the device's context into account. The objective of this chapter is to have an insight of how dynamic approaches can complement static analyses. We were the first[1] to present a tool-chain to dynamically instrument Android applications in vivo, i.e. directly on the device. We present two use cases instrumenting applications to show that dynamic approaches are feasible, that they can leverage results from static analyses, and that they are beneficial for the user from the point of view of security or privacy. One of the use case is a fine-grained permission system prototype enabling the user to disable or enable application permissions at will.

This chapter is based on work published in the following technical report:

• **Alexandre Bartel**, Jacques Klein, Martin Monperrus, Kevin Allix and Yves Le Traon: Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation, ISBN 978-2-87971-111-9, 2012.

6.1 Introduction

On the official market of Google (Google Play, formerly AndroidMarket), more than 10 000 new applications are available every month.[2] For the end user, downloading an application on her smartphone is similar to choosing an apple on an apple tree: she only sees the surface and has no evidence that there is no worm in it. Unfortunately there are many worms of different kinds waiting to infect smartphones such as malware leaking private data and adware calling premium-rate numbers.

[1] Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation, Tech. Report, ISBN 978-2-87971-111-9, May 22, 2012

[2] http://www.appbrain.com/stats/number-of-android-apps

In this chapter we claim that an efficient and readily applicable means to improve privacy of Android applications is to perform runtime monitoring and interception of the application interactions with the Android stack by instrumenting the application bytecode directly on the smartphone (in vivo). Before further introducing our contribution let us defend our key claim.

Why performing runtime monitoring and interception? We want to allow or disallow behaviors of an application at runtime. We use runtime monitoring as it consists of observing the behavior of an application during execution. It collects certain metrics or intercepts all exchanges at the interface between the application and the rest of the system. In this chapter, we discuss two case-studies involving runtime monitoring and interception, including an implementation of a fine-grained permission model on top of the Android software stack as proposed in [123].

Why performing bytecode instrumentation? There are at least two ways to perform runtime monitoring and interception: modification of the Android software stack or bytecode instrumentation. Modification of the software execution stack consists in altering the operating system or the core libraries to intercept the required information. On Android, it means changing the underlying kernel, the Dalvik virtual machine or the Android framework. Unless convincing the Android consortium, this is rather limited in deployment since normal end-users have neither the rights (jailed phones) nor the ability to do so. Also, this solution would require users to change their firmware which is a non-trivial task, further complicated by the so called *fragmentation problem* of the Android system as there is not a single Android system but many different Android systems each customized to run on a specific device (tablet, smartphone, ...). If the operating system is modified, one would need to create a custom instrumented version for every possible Android version which is not easily doable in practice. Bytecode instrumentation however, is one of the lightest way to perform runtime monitoring on top of execution platform that cannot be modified. In the context of a fine-grained policy enforcement for improving privacy, we are able – thanks to bytecode instrumentation – to enforce a fine-grained permission model of already deployed applications on Android smartphones without any modification of the Android software stack.

Why performing in vivo instrumentation directly on smartphones? Bytecode instrumentation could be done outside the device for instance using a remote service on the Internet. However, many countries forbid distributing binaries to third-party services (e.g. France). Also, terms of service of several markets (e.g. Google Play for Android) do not allow this. Instrumenting applications directly on the device keeps the application within the device.

To sum up, we believe that the most efficient and practical way for ensuring security and privacy on mobile devices is to instrument the application bytecode directly on the smartphone (in vivo), the instrumentation being tailored for a given security or privacy concern. Our main contributions are that:

- We have built a toolchain to automatically repackage Android applications directly on an Android device;
- We have built a toolchain to automatically analyze Android applications directly on an Android device;
- The toolchain has been tested by implementing two prototypes which increase the end-

user privacy. One removes advertisement and the other gives the user total control over the applications' runtime permissions.

- The feasibility of such a tool chain has been evaluated. Limitations and challenges have been pinpointed.

To the best of our knowledge, we were the first[3] to present a tool chain to automatically transform Android applications directly on a device.

The chapter is organized as follows: Section 6.2 provides the reader with two scenarios motivating the need of bytecode instrumentation of Android applications. Section 6.3 describes a tool chain for instrumenting Android applications directly on Android devices (smartphones, tablets, ...). Section 6.4 presents the design and implementation of valuable bytecode instrumentations for the security and privacy of smartphones. Section 6.5 demonstrates the feasibility of running the whole tool chain in a reasonable amount of time. Finally, Section 4.8 concludes the chapter.

6.2 Motivation for Bytecode Instrumentation

There are different scenarios in which it would be beneficial to manipulate and analyze Android applications' bytecode directly on smartphone devices (in vivo). In this Section we present two valuable use cases: *AdRemover* and *BetterPermissions*.

Both of them improve the privacy for the user. *AdRemover* hinders advertisement libraries to work and thus, at the same time, prevents them from sending private information related to localization (GPS coordinates,...) or of the device itself such as the IMEI (International Mobile Equipment Identity). *BetterPermissions* gives users the power to enable or disable applications' permissions. In an extreme case where the user would like no application to have access to her contact list, she would remove the contact permission from all applications on the phone. The result is a better privacy for the user.

6.2.1 Advertisement Removal

Nearly half of the Android applications embeds third-party code to handle in-app advertisement [101]. A significant proportion of ad-supported apps include at least two advertising libraries [120].

Furthermore, Android applications are distributed as self-sufficient packages, bundling together both specifically developed code and the third-party libraries they may need, such as binary-only advertisement modules.

Android enforces a per-application policy-based security model: either all parts of an application benefit from a given permission, or none of its parts. It means that when a user grants permissions to an application, she actually grants permissions to components potentially written by different entities, including the ad libraries.

For example, a newspaper app may be allowed to send its location back to the app publisher so that she is presented with local news. However, from a privacy perspective the embedded

[3] we published a technical report in May 2012 [13]

advertisement library should not be allowed to send the location data to the ad companies. Currently, the user faces a dilemma: she either has to reduce her privacy level expectation, or refrain from using an otherwise valuable application.

A workaround of this limitation of the platform is to disable the use of the ad library in vivo.

This may have positive side-effects, since advertisement libraries also have a significant impact on the battery usage. According to a recent study [100], third-party advertisement modules can be held responsible for up to 65%-75% of energy spent in free applications .

6.2.2 Fine-Grained Permission Policy

The Android framework relies on a permission-based model and follows an *"all or nothing"* policy. At installation time, users must either accept or reject all permissions requested by the application. An application is installed only if all the requested permissions are accepted. There is no way to accept only some permissions (such as accessing the localization data) and not others (such as connecting to the Internet). Users are doomed to completely trust the application developers who write the list of permission. Enck et al. [48] have pointed out that an application with several sensitive permissions is a real security threat. For instance if an application requests the permission to send SMS and a permission to read the contact list, the contact list could potentially be sent to a remote phone by sending it through SMS.

A fine-grained permission model consists in giving users the ability to specify their own set of permissions to applications, according to their own usage. All sets of permissions for all applications on the device constitutes the security policy. The underlying permission-based system would then enforce this user-defined policy.

Running such user-level security policy is impossible on a unmodified Android platform with unmodified application code. However, as we show later, it is indeed possible by manipulating the application bytecode at installation time, in vivo.

6.3 Toolchain for In vivo Bytecode Instrumentation

This section presents our proposal for performing bytecode instrumentation of Android applications in vivo, i.e. directly on smartphones. The reader may refer to Section 2.1 for more detail on the Android stack and Android applications.

6.3.1 Requirements

Instrumenting and repackaging a fully-runnable Android application is not straightforward. It consists of extracting the executable code from the application code, analyzing and instrumenting it, rebuilding a new working Android application and signing it again, since the OS requires applications to be signed.

Our toolchain has the following requirements:

1. The Android OS must be unmodified (for the sake of a broad applicability as presented in Section 6.1);

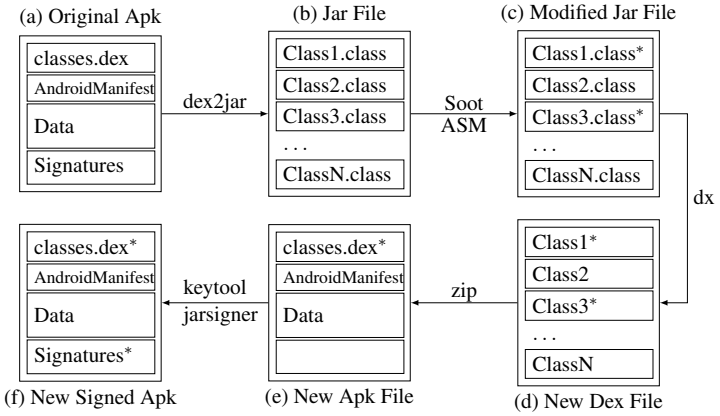


Figure 6.1: Our Process to Instrument Android Applications

2. The Dalvik virtual machine that runs Android applications must be unmodified, in particular in terms of configuration values such as the maximum heap size (for the sake of a broad applicability, see Section 6.1);
3. The hardware that is used to instrument bytecode must be representative of common smartphones on the market.

6.3.2 Toolchain

The bytecode instrumentation process features the following steps: 1) Extract code from Android application apk files; 2) Modify the extracted code with bytecode manipulation tools; 3) Rebuild a new Android application containing the modified code.

Those three steps can be broken down into five elementary steps, as shown in Figure 6.1: i) Extracting and converting the Dalvik bytecode into Java bytecode (step a-b), ii) Manipulating the bytecode (steps b-c), iii) Translating this representation back to Dalvik bytecode (step c-d), iv) Rebuilding a new apk file (step d-e) and v) Finally signing all files with a new private key (step e-f). Let us now discuss the tools that are used in each step.

i) Extracting the Dalvik Bytecode The first step, as shown in Fig. 6.1.(a-b), is to extract the `classes.dex` file from the apk file and convert it to Java bytecode classes which can be analyzed with standard unmodified Java bytecode analysis toolkits. For this step, we use the tool `dex2jar`[4].

ii) Instrumenting the Bytecode In this step, we experiment with two different tools which manipulate bytecode. Recall that bytecode manipulation is the step from (b) to (c) at illustrated in Figure 6.1. Using different tools gives us the opportunity to measure the difference in terms

[4]available at `http://code.google.com/p/dex2jar/`

of execution time and memory consumption between them and decide which one is more appropriate to manipulate bytecode in a memory constrained system.

ii.a) Soot. Classes are transformed to Jimple with the Soot analysis toolkit. Soot [75] is an open-source analysis toolkit for Java programs. It operates either on Java source code or bytecode. It allows developers to analyze and transform programs. For instance, an intra-procedural flow analysis could determine if a variable can be *null* at some point in the code. Soot can also perform different call graph analyzes, useful for specific bytecode instrumentation. Most analyses and transformations in Soot use an internal representation called Jimple. Jimple is a simple stack-less representation of Java bytecode. We ported Soot on the Android system by converting its Java bytecode to Dalvik and creating a wrapper Android application. To our knowledge there is no previous work which represent Android bytecode as an abstraction on which on could perform static analysis directly on the smartphone.

ii.b) ASM. We experienced that Soot is sometimes slow and requires a lot of resources (especially memory). Thus, we also run ASM for bytecode instrumentation. ASM [25] is a Java bytecode engineering library. One of its characteristics is that it is lightweight hence more suitable for running on systems constrained in term of memory or processing resource. It is primarily designed to manipulate and transform bytecode although it can also be used to perform some program analysis. It features a core API to perform simple transformations as well as a tree API to perform more complex bytecode transformations (which requires more CPU processing and memory space).

iii) Translating the Modified Bytecode back to Dalvik Bytecode Once the classes are analyzed and modified by the analysis toolkit, they are transformed back into Dalvik bytecode using `dx`[5] which generates the `classes.dex` file from Java class files. This step is illustrated in Fig. 6.1 as the edge c-d.

iv) Rebuilding Application As presented in Fig. 6.1.(d-e), after the fourth step, a new Andoid application is built. The newly generated `classes.dex`, the data and the Android manifest from the original application are all inserted in a new zip[6] file.

v) Signing the Modified Application Android requires applications to be cryptographically signed. Hence, all files of the generated zip file are signed using a newly created couple of public/private keys (not represented on the figure), The new public key is added to the zip (not represented on the figure). We used the `keytool` and `jarsigner` Java programs to sign applications (Fig. 6.1.(e-f)).

Signing applications with new keys may cause compatibility problems between applications. For instance two or more applications signed with the same key can share the same process. In order for this feature to continue working a one-to-one mapping between old keys and new ones needs to be maintained in order to sign two transformed applications (originally signed with the same keys) with the same new generated keys. Maintaining this mapping and handling such compatibility between applications is out of scope of this chapter.

[5] using `com.android.dx.command.Main` from the Android SDK

[6] using the `java.util.zip` library

We have devised a bytecode manipulation process on Android using standard tools. The following presents the design and implementation of two concrete bytecode instrumentation prototypes.

6.4 Use-case Design and Implementation

Any use-case leveraging the toolchain presented in Section 6.3 analyzes or modifies the bytecode of an application. Analyzing or modifying the bytecode is represented by step (b-c) in Figure 6.1. We now present how we have implemented and evaluated the two use-cases of Section 6.2. Thy both modify the bytecode of applications. AdRemover modifies the bytecode to remove advertisement. BetterPermissions modifies the bytecode to enable a fine-grained permission policy system for the user.

6.4.1 Implementation of AdRemover

We focus on two widely used Android advertisement modules: AdMob and AdSense. Advertisement is not part of the Android system but is present in the application's bytecode. Thus, applications do not share ad library code. However, they each have a copy of the library code. Disabling advertisement requires to instrument every application containing an ad library.

Advertisement requires I/O operations for fetching the ad data. An Android application developer using an ad library do not want her app to crash because of the ad library. This is the reason why developers of ad libraries take special care of exceptions when designing the ad library. They expect I/O operations to fail on a regular basis, depending on unpredictable contexts. For example, an exception can be thrown if the device has no network coverage anymore.

Building on this observation, we make the assumption that I/O code has been placed by ad developers inside a Try/Catch block to recover for exceptions raised by I/O failures. Our tool leverages this assumption and inhibits every Try/Catch section of the ad packages of the application. For every Try/Catch block it encounters, our tool extracts the type of the handled I/O exception, creates such an exception object, and inserts an instruction that throws this exception at the very beginning of the try block.

For this, we collected the Java package names used by these libraries and we configured AdRemover to operate only on classes that are part of those packages. We wrote two implementations of AdRemover: One using Soot and one using ASM.

6.4.2 BetterPermissions: A Fine-grained Permission Policy Management

In this context a fine-grained policy is a file in which the user specifies which permissions are granted to applications. In the real world users are only familiar with permissions and applications, so it makes perfect sense to limit policies at the level of applications and not a lower level (such as Android component or Java methods). However, for explanatory purposes the policies in this Section contain a mapping between Java methods and permissions.

For a user-centric policy to exist, we need to instrument the bytecode of every application one wishes to control. Recall from Section 2.1 that Android applications communicate to the

Android system through the Android API. The instrumentation detects all API calls protected by one or more permissions and redirected every of those calls to a *policy service*. The policy service is a Android service component part of independent Android application. Base on the user defined policy it authorizes or not the application to call the protected method.

When the instrumented application runs, the user-defined policy is enforced by the policy service. Indeed, for every instrumented method, the running instrumented application calls the policy service and the policy is checked. If the policy allows the original API method call, the API call is performed. Otherwise, a fake implementation is executed and returns a fake default value.

Our prototype tool enforces a user-defined policy at the user level (also called application level). It allows users who previously could not modify the system policy to enforce their own policy for a set of applications. Modifying code to insert security check is known as Inline Reference Monitoring (IRM) and has been first introduced by Erlingsson et al. and Evans et al. [50, 51, 52].

Instrumenting the Application To control or limit an application's permission it's bytecode has to be instrumented. This is illustrated in Figure 6.2 where application $NewsReader$ is represented as a graph of method calls starting from node s. All method calls that require one or more permissions [**bartel2012automatically**, 53] are wrapped with code which in order:

1. asks the policy service if the application is authorized to call the method
2. according to the answer from the policy service either invokes the original method or the fake method.

For instance, the `getLocation(p1)` method invocation of node *7* (which requires permission `GPS`) has been wrapped in the figure by a call to the *policy* service. If the policy approves this call, the original `getLocation(p1)` is executed, otherwise a fake method is invoked, returning a fake default value.

In total, there are N instrumentations where N is the number of API calls under consideration present in the application bytecode.

Defining the Policy The next step, as shown in Figure 6.3, is to define the policy regarding the instrumented applications. The user defines a set of allowed permissions for each application. Behind the scene, the policy generates a list of all Java methods which require the enabled permissions. Those methods are set as authorized. In Figure 6.3, only method `getLocation()` is allowed for application Instrumented NewsReader.

Note that this step could be performed first to instrument only method calls which are not authorized by the policy. However, instrumenting every API method calls which requires one or more permissions makes it possible to change the policy at runtime.

Policy Service Finally, when the instrumented application runs, the policy is enforced by the Policy service as shown in Figure 6.4. For every instrumented method (here the original/instrumented method is `getLocation` and its associated permission `GPS`) the running application

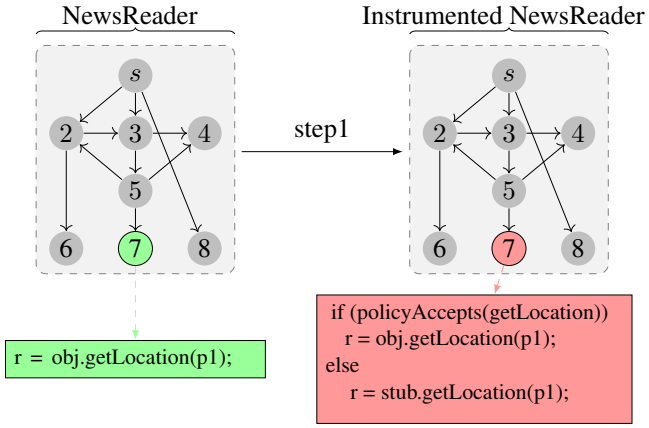


Figure 6.2:]
Step 1: Wrapping and Redirection of Android API Calls For Fine-Grained Permission Management

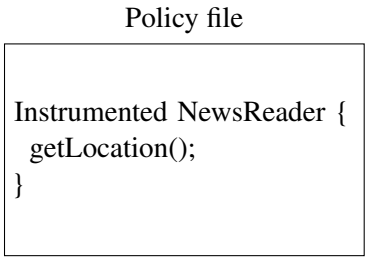


Figure 6.3: Step 2: The Policy File Defines that InstrumentedNewsReader is Allowed to Use API Method getLocation

calls `policyAccepts()` (step *A*) and the policy is checked by calling `policyHas()` (step *B*). Method `policyAccepts()` returns `true` if the policy allows the original method or false if it does not. If the original method is allowed in the policy, the original method is called (this is the case in Figure 6.4, since step *C* returns *true*). Otherwise, the stub method corresponding to the original method is executed. Here, the stub handling method `getLocation` is not executed. We implemented the policy service as an Android service and the instrumentation code as a plugin for the static analysis tool Soot.

6.4.3 Evaluation

We now check whether our use-case implementations work. For both of them, we run the instrumentation against a real-world application and runs the resulting modified application.

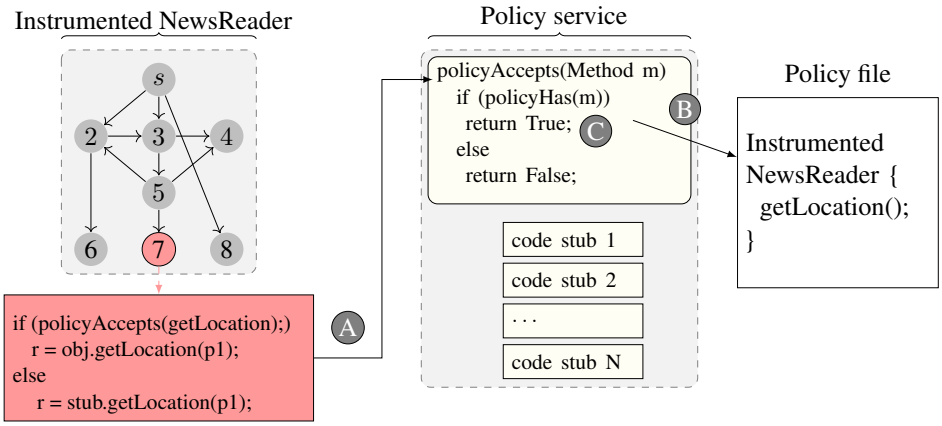


Figure 6.4: Step 3: The Policy Monitor Enforces the Fine-Grained Permissions by Returning Default Values for Unauthorized API Calls

AdRemover We test that our tool is functional by selecting a random application on the Android Market. We make sure that the test application uses one of the two advertisement modules currently handled by AdRemover.

First we run the unmodified test application on an Android devices, and make sure that it is a working application, and that it actually displays advertisements.

We then send this application to our toolchain (with the Soot implementation) running on a PC. The modified application is still functional, and no more advertisements are displayed. We monitor the network connection during the test and found out that it the application does not send any ad request anymore.

Finally, we process the unmodified application again, this time running the bytecode manipulation directly on the smartphone. Running the modified application yielded the same results as with the application modified on a standard PC.

BetterPermissions For evaluating the fine-grain policy, we select another random application and instrument it to wrap every permission sensitive API call related to the GPS. The application is instrumented and then repackaged into a new signed application. We run the instrumented application on an Android device, and test it with different policies. The user-defined policy is enforced as expected.

To sum up, the two bytecode transformations result in applications that correctly runs. Those first results are important as the two use cases illustrate what can be achieved using the bytecode instrumentation toolchain. What also matters for us is to know whether the toolchain under consideration can be run in vivo on a large dataset of Android applications given the memory and CPU limitations of current smartphones. The next section answers to this questions by measuring execution time and memory consumption of in vivo instrumentation.

Name	Processor	Memory	Android	Heap Size
smartphone1	ARM 800MHz, 1 core	512MiB	2.2	24MiB
smarthpone2	ARM 1.2GHz, 2 cores	768MiB	2.3.4	32MiB
tablet1	ARM 1.4GHz, 4 cores	1GiB	4.0.3	48MiB

Table 6.1: The Hardware used in our Experiment

6.5 Performance of In Vivo Instrumentation

In this section we present the results of applying the instrumentation process presented in Section 6.3 and summarized in Fig. 6.1. The goal is to know: 1) whether it is possible to manipulate bytecode on smartphones given the restricted resources of the hardware. 2) whether it takes a reasonable amount of time.

6.5.1 Measures

We measure the execution time of the five steps of the instrumentation process on a set of 130 Android applications. This set is described in Section 6.5.3. We run the instrumentation process on three different Android smartphones whose configurations are presented in Section 6.5.2.

The feasibility of the whole process is measured by the time to pass every step of the toolchain (1: $dex2jar$, 2: $Soot/ASM$, 3: dx, 4: $customZip$, 5: $signature$). The time to run each step and the number of applications that successfully go through each step are measured as well.

For the second step of the process (Step: Instrumenting the bytecode), we evaluate both ASM and $Soot$. For ASM, we measure the time required to instrument Java bytecode on the AdRemover case study. The AdRemover transformation leverages the ASM tree API to perform the try/catch block manipulation described in 6.4.1. $Soot$ is evaluated by measuring the time required to generate Java classes for both AdRemover and BetterPermissions case studies (AdRemover is implemented with ASM and Soot).

6.5.2 Experimental Material

We conduct the experiment on three Android-based smartphone devices. Their configuration is detailed in Table 6.1. The main differences are the processor clock speed (0.8, 1.2 and 1.4 GHz), the total amount of main memory (512, 768 and 1024 MiB), the Android version (2.2, 2.3.4 and 4.0.3) and the maximum heap size of the Dalvik virtual machine (24, 32 and 48). Since the heap size controls the maximum memory that can be allocated by a single process it also controls the maximum number of objects that can be allocated simultaneously.

The number of cores also differs. However, we do not take advantage of multiple cores during the experiments. This hardware complies with requirement #3 mentioned in 6.3.1.

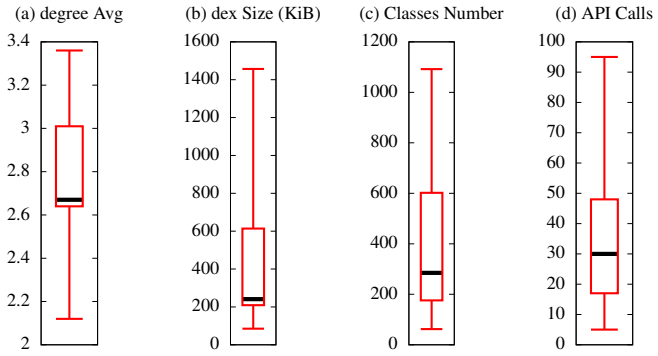


Figure 6.5: Descriptive Statistics of the 130 Applications of our Dataset

6.5.3 Dataset

We apply the whole experimental protocol on a set of 130 Android applications randomly selected among the top 500 applications from the Android market[7]. They span various domains such as finance, games, communications, multimedia, system or news. This dataset is not artificial as it only consists of real world applications.

To give a better overview on these applications, Figure 6.5 shows the key application metrics as boxplots. They indicate that most (75%) of Android applications have less than 614 KiB of Dalvik bytecode, less than 602 classes, an average method degree smaller than 3. Haft of the applications have more than 30 calls to a method of the Android API which require a permission.

6.5.4 Dalvik to Java Bytecode Conversion

The conversion time from the Dalvik executable code to Java bytecode using *dex2jar* is shown in Fig. 6.6.

Observation 1 The time to convert dex files to jar does not exceed 60 seconds on *smartphone2* and *tablet1* for 75% of the applications. The conversion time does not exceed 250 seconds on our dataset of Android applications.

Observation 2 The application with the biggest Dalvik bytecode file (4000 KiB) is successfully converted both on *smartphone2* and on *tablet1*.

Observation 3 We notice that the conversion time is linear with the size of the dex file (of the form $a \cdot X + b$) for a Dalvik bytecode size less than 4000 KiB. Using linear regression, we find that for *smartphone2* a equals 0.069 and b equals 0.3. For *tablet1* we have, 0.049 and -0.4. The linear relation between the conversion time and the size of the Dalvik bytecode enables us to theoretically predict the necessary amount of time to convert any size of Dalvik bytecode (if we extrapolate for size bigger than 4000 KiB). For instance, the time to process the Android

[7] `http://play.google.com`

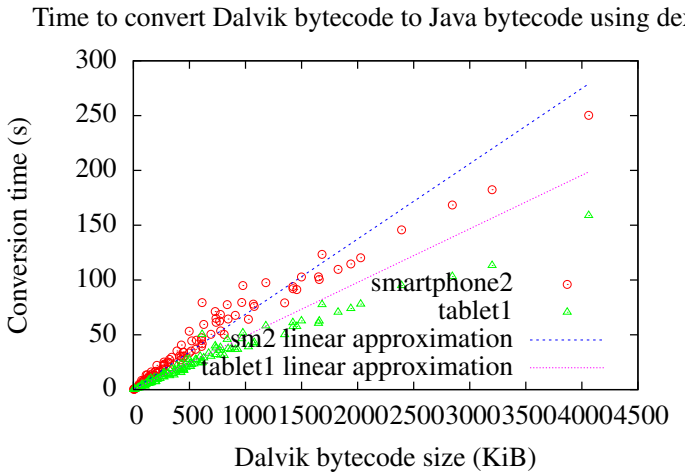


Figure 6.6: Performance of In Vivo Dalvik to Java Bytecode Conversion.

application with 10 MiB of Dalvik bytecode would be 700 seconds for *smartphone2* and 500 seconds for *tablet1*.

Conclusion 1 Converting Dalvik bytecode to Java bytecode in vivo is feasible within minutes.

Limitations: smartphone1 is unable to process any dex file. Also, when using *smartphone2* and *tablet1*, 26 and 11 dex files, respectively, cause the conversion Android application dex2jar to crash. This crash is either an `OutOfMemory` or a `StackOverflow` exception.

Result of *smartphone1* is explained by the hard-coded maximum heap size of Android (32 MiB or 48 MiB). For the two other devices, crashes are to be attributed to the default 8 KiB stack size. In total, 104 (80%) Android applications were successfully converted to a jar file on *smartphone2* and 119 (91%) on *tablet1*.

However, since Android devices become more and more powerful the default heap size of the Android system grows. Indeed, in Android 2.2 the heap size is 24 MiB, in Android 2.3.4 32 MiB and in Android 3.0 48 MiB. This continued growth would allow our tool chain to convert Android applications which have bigger Dalvik bytecode size.

Also, some applications may be obfuscated to prevent Dex2jar to convert Dalvik bytecode to Java classes. We did not encounter any obfuscation during the experiment. Our toolchain relies on independent components. Thus, if Dex2jar cannot handle some obfuscation techniques it could easily be replaced by an equivalent component which handles them.

6.5.5 Performance of Bytecode Manipulation

This section presents our performance measures of in vivo bytecode manipulation using two different instrumentation libraries: ASM and Soot.

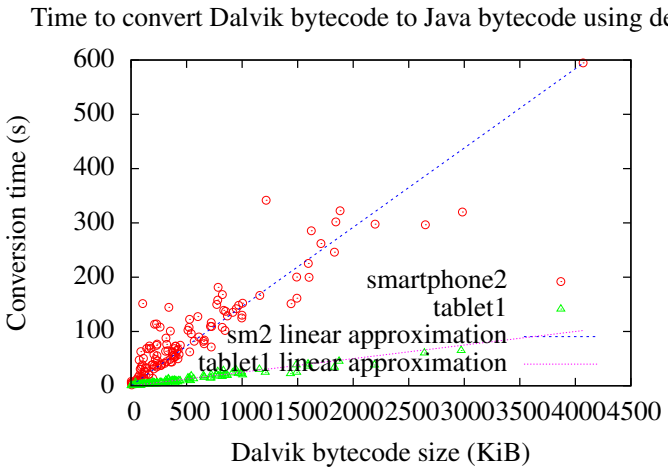


Figure 6.7: Transformation Time of In Vivo Java Bytecode Manipulation with ASM

Manipulation With ASM

Transformation time of Java bytecode using ASM is represented in Figure 6.7. In this experiment the AdRemover transformation described in 6.2.1 is implemented using ASM.

Observation 4 All 104 applications successfully transformed with dex2jar on *smartphone2* are successfully processed by ASM in vivo. It processes every jar (up to 4MiB in size) in less than 600 seconds.

Observation 5 We notice that the transformation time is linear with the size of the jar files (of the form $a \cdot X + b$) for a Dalvik bytecode size less than 4000 KiB. Using linear regression, we find that for *smartphone2* a equals 0.146. For *tablet1* we have, 0.025.

Conclusion 2 Manipulating bytecode on smartphones using ASM is feasible. Given our transformation and our dataset, ASM does not have specific memory or CPU requirements that are incompatible with smartphone resources.

Manipulation With Soot

We now consider the Soot implementation of the AdRemover transformation. Out of the 130 Android applications, only 3/130 are correctly processed on *smartphone2* and 19/130 are correctly processed on *tablet1*.

Observation 6 Only the smallest applications (in terms of Dalvik bytecode) can be converted. For instance, it takes less than 30 seconds to convert any jar which size is less or equal to 20 KiB on *smartphone2*. However, larger, yet small applications (in the 25% quartile), take up to 18 minutes for being instrumented with Soot.

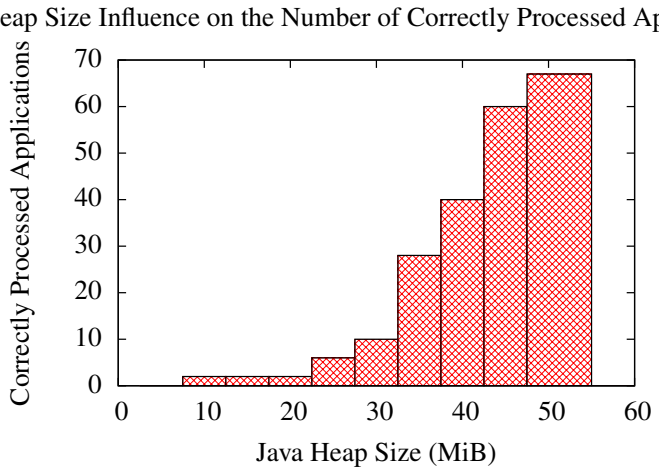


Figure 6.8: Influence of the Heap Size on Jimple Transformation

Conclusion 3 Using Soot in vivo is feasible only for the smallest applications. We assume that the heap size is the main blocking factor of using Soot in vivo. To check this assumption, we conducted an experiment on a desktop computer consisting of analyzing our dataset of Android applications with different maximal heap sizes (from 5 Mib to 50 Mib by steps of 5 Mib). Results are presented Fig. 6.8. Soot was able to process 67 applications with a heap size of 50 Mib. Those results clearly indicate that maximum half of the Android applications could be processed with a heap size of 50 MiB. Under the assumption that the heap usage (hence the maximum required size) is similar on the Java and Dalvik virtual machines, it means that the memory is actually the main blocking factor of using Soot on Android.

6.5.6 Java Bytecode to Dalvik Conversion

Once an application has been instrumented at the Java bytecode level, it has to be transformed back into Dalvik bytecode. Conversion time from Java classes to the dex file using the *dx* tool is shown in Fig. 6.9.

Observation 7 Java bytecode of 33/130 on *smartphone2* and and 39/130 applications on *tablet1*, respectively, have been successfully converted to Dalvik bytecode.

Observation 8 Conversion time for jar files ranging from 20 to 400 KiB does not exceed 80 seconds.

Conclusion 4 The Dx tool is a bottleneck of the tool chain. It can only correctly process 25 to 30% of the applications. The reason is that it puts every Java class in memory and suffers from the memory limitation of in vivo processing, similarly to Soot. This tool is used off the shelf and could be optimized to run on devices where resources are limited, by processing class after class to limit memory consumption. .

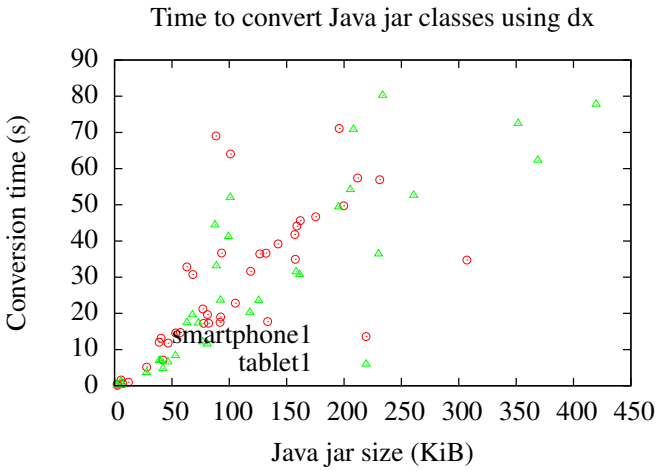


Figure 6.9: Conversion Time of In Vivo Java Bytecode to Dalvik Translation.

6.5.7 Creating a New `apk` File

The time taken to create an `apk` file from the instrumented Dalvik bytecode is shown in Fig. 6.10. Note that for this step, the input set is not the output of the previous step. We only have 39/130 applications that have been correctly processed in the previous steps. At every step, some applications failed. For the remaining 91/130 applications where the final instrumented Dalvik bytecode could not be computed, we take as input the original Dalvik dex file of the application. In this way, the problems of the previous step do not interfere with the results of this fourth step.

Observation 9 121/130 inputs were successfully processed. There is no clear relation between the size of the previous `apk` file and the creation time of the new `apk`. Only 9/130 applications generate an exception because their size is too big and can thus not be processed by the zip utility.

Observation 10 For 95% of the applications it takes less than five seconds regardless of the device and of the size of the original `apk` file.

Conclusion 5 It is feasible to create `apk` files on smartphones. The time to create a new `apk` file is negligible compared to the time to convert bytecode or to manipulate bytecode with Soot.

There is no linear relation with the Dalvik size as it is the case in Fig. 6.6 and 6.9. This is probably due to the fact that when generating apk files, others factors than the bytecode size come into play, such as handling the media files (images, sound, etc.), which sometimes dominate the Dalvik bytecode size.

6.5.8 Signing the Generated `apk` File

Signing time of applications is represented in Figure 6.11.

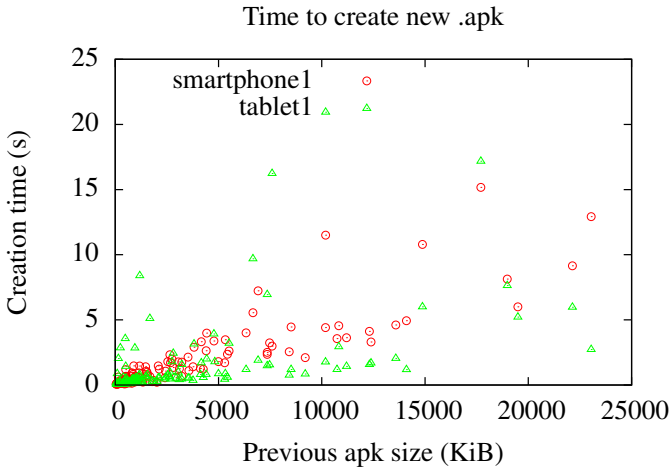


Figure 6.10: In Vivo Creation Time of a New `apk` File

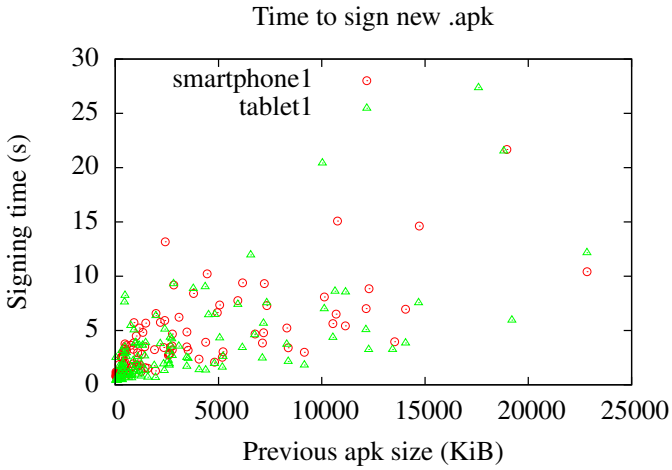


Figure 6.11: Performance of In Vivo Signature of the Instrumented Apk File

Observation 11 120/130 Android applications were successfully signed on *tablet1*. There is no clear relation between the size of the `apk` file and the signature time of the `apk` file. 14/130 applications generate an exception because their size is too big and can thus not be processed (14 on *smartphone2* and 10 on *tablet1*).

Observation 12 For 95% of the applications a maximum of 12 seconds is required to sign the application regardless of the device and the size of the `apk` file.

Conclusion 6 It is feasible to sign `apk` files on smartphones. Similarly to the `apk` file creation step, the computation time is negligible. The difference observed between *smartphone1* and *smartphone2* reflects the difference in their CPU clock frequencies.

6.5.9 Conclusion

We now recapitulate the results of our experiments of in vivo modification of Android applications.

Feasibility

Table 6.3 summarizes all the experiments for *smartphone2* and highlights the feasibility of the whole approach.

Total execution times for all steps of the toolchain are computed for the Soot and ASM version. For an ASM-based instrumentation it takes a median time of 120 seconds, that is 2 minutes, to process an application. We think that users would agree with waiting 2 minutes before starting using an application, if they are provided more guarantees with this instrumentation process enabling better privacy. During those 2 minutes the phone is still usable since only once core is used (most smartphones feature multi-core CPUs) and only the maximum amount of heap memory allowed by the virtual machine can be used (and not all memory).

Those experiments show that it is feasible to manipulate bytecode directly on Android devices. The most expensive steps of the process are the conversion of Dalvik to Java bytecode and vice versa, and the Soot bytecode manipulation step.

hopsasa

How to Improve Performance of In Vivo Instrumentation?

According to our analysis, the main blocking factor is the memory. The maximum heap size required to analyze and transform applications is an issue for many transformation steps. We think that this issue can easily be solved by 1) the next generation of more powerful hardware and 2) the upcoming versions of the Android OS and virtual machines which will likely have a significantly higher maximum heap size (e.g. Android 4 heap size is set to 48 MiB).

Dalvik to Java conversion and Java to Dalvik conversion are two very time expensive steps. They use unmodified versions of Dex2jar and dx. There are two ways to overcome those resource-hungry tools.

First, those tools were never optimized to run on platforms with limited resources. We believe that there are many optimization opportunities in terms of CPU and memory consumption.

Step Name	Min. Time (s)	Avg. Time (s)	Median Time (s)	Max. Time (s)	App.	Feasibility
Conversion .dex to .jar (a-b)	0.22	43.76	28.9	250.2	104/130 (80%)	⋆⋆⋆
Analyzing .jar with Soot(b-c)	25.8	76	26	187.7	3/130 (2.3%)	
Analyzing .jar with ASM(b-c)	1.55	90.45	65.1	594.67	129/130 (99.2%)	⋆⋆⋆⋆
Conversion class to dex (c-d)	0.09	28.07	22.8	71	39/130 (30%)	⋆⋆
Creating new .apk (d-e)	0.06	1.89	0.87	15.1	119/130 (91.5%)	⋆⋆⋆⋆
Signing new .apk (e-f)	0.71	3.85	3.0	21.67	116/130 (89.2%)	⋆⋆⋆⋆
All Steps with Soot (a-b-c-d-e-f)	26.88	153.57	81.57	545.67	3/130 (2.3%)	⋆
All Steps with ASM (a-b-c-d-e-f)	2.63	168.02	120.67	952.64	39/130 (30%)	⋆⋆⋆

Table 6.2: Summary Metrics of Our In Vivo Instrumentation Process for Smartphone2. There are problematic steps but the overall process is feasible.

Step Name	Min. Time (s)	Avg. Time (s)	Median Time (s)	Max. Time (s)	App.	Feasibility
Conversion .dex to .jar (a-b)	0.19	25.6	17.85	158.9	119/130 (91.5%)	⋆⋆⋆
Analyzing .jar with Soot(b-c)	24.2	76	352	1054	19/130 (14.6%)	⋆
Analyzing .jar with ASM(b-c)	1.55	11.3	7.06	65.5	119/130 (91.5%)	⋆⋆⋆⋆
Conversion class to dex (c-d)	0.09	29.5	20.2	80.2	33/130 (25.3%)	⋆⋆
Creating new .apk (d-e)	0.03	1.6	0.5	20.9	121/130 (93.1%)	⋆⋆⋆⋆
Signing new .apk (e-f)	0.4	3.4	1.91	27.3	120/130 (92.3%)	⋆⋆⋆⋆
All Steps with Soot (a-b-c-d-e-f)	24.91	136.1	392.46	1341.3	19/130 (14.6%)	⋆
All Steps with ASM (a-b-c-d-e-f)	2.26	71.4	47.52	352.8	33/130 (25.3%)	⋆⋆⋆

Table 6.3: Summary Metrics of Our In Vivo Instrumentation Process for Tablet1

Second, one could replace those tools by better alternatives. For instance, an ASM-like library for manipulating Dalvik bytecode would allow to skip Dalvik-to-Java and Java-to-Dalvik conversion. Such tools are emerging such as ASMdex[8]. Another solution would consist of performing bi-directional transformations directly from Dalvik bytecode to Jimple bytecode which are both register based. We are indeed working on a Dalvik to Jimple translation prototype called Dexpler [12].

To sum up, our results show that we can reasonably imagine to manipulate the bytecode on 100% of our dataset applications within at most 5 minutes.

Threats to Validity

Let us now discuss the threats to validity of our experimental results.

Implementation Bug: Our results hold as far as there is no serious bug in the implementation of any of the five programs involved in the five steps, as well as in the glue and measurement code we wrote.

Dataset Generalizability: Our dataset may not be representative of the Android applications used in the real-world.

Linear Extrapolation: The linear relations we establish for the Dalvik to Java and the Java to Dalvik conversions holds for bytecode size less or equal to 300 KiB. It may not hold for bytecode whose size is bigger. In the presence of non-linear singularities, it may not be possible to analyze large applications.

Bytecode Manipulation Time: Our results on the bytecode manipulation time were obtained with relatively simple transformations. It may be the case that complex transformations are not of the same order of magnitude and consume much more memory. However, for the use cases presented in Section 6.2, the instrumentation only consists in monitoring and proxying Java methods.

6.6 Conclusion

The toolchain we propose and evaluate in this chapter is a milestone that respond to the recent claim of Stravou et al. [123] about the urgent need for bytecode analysis to perform in vivo security checks on mobile phones. We have 1) proposed a tool chain allowing the manipulation, instrumentation and analysis of Android bytecode and 2) shown that it is possible to run the tool chain in a reasonable amount of time directly on unmodified smartphones with unmodified Android software stack. Concretely, our experiment shows that with ASM, 39 (30%) applications of our dataset can be instrumented in less than 952 seconds (with a median time of 120s). Moreover, we discuss specific limitations that we observed, such as the hard-coded heap size of Android systems.

We believe that those various limitations could be quickly overcome, at least for two main reasons. First, we used off-the-shelf Java tools that are not optimized to run on environments

[8] See `http://asm.ow2.org/asmdex-index.html`

where resources (memory/CPU) are limited, and there may be possibilities of significant optimization. Second, the hardware and OS evolution of smartphones will make it possible to process ever bigger Android applications (for instance, on Android 4, the default size of the heap is twice as large as in the previous version).

We are currently working on other use cases. In particular, we are implementing a behavioral malware detection approach that is set up and run on the smartphone. This approach involves instrumenting the bytecode to redirect API method calls to stubs responsible for detecting malicious behavior.

Chapter 7

Related Work

In the last four years, there has been a steep increase in research about the Android stack and its applications. This could be explained by the fact that the Android stack is popular, open-source, which eases analysis and modification of the system, and that millions of applications are available to analyze.

The rest of this chapter is organized as follows. Section 7.1 describes research related to Dalvik bytecode parsing and typing. Section 7.2 focuses on alternative techniques to extract the permission map from the Android system. Section 7.3 summarizes the techniques used to analyze inter-component and inter-application communications as well as techniques to find leaks in Android applications. Finally, Section 7.4 describes research related to instrumentation of Android applications directly on a device.

7.1 Local Typing for Dalvik

In practice the Java source code or Java bytecode of an Android application is not available, only the Dalvik bytecode is. The Java language appeared in 1995 and since them tools have been developed to analyze Java source code and bytecode. This is the reason why there has been an interest in tools to convert Dalvik bytecode back to Java bytecode so that existing tools could be used to analyze Android applications. In Section 7.1.1, we discuss tools to convert Dalvik bytecode to Java bytecode and in Section 7.1.2, we discuss tools that disassemble and/or assemble Dalvik bytecode using an intermediate representation.

7.1.1 Dalvik to Java Bytecode Converter

Ded [47] and Dare [91] are Dalvik bytecode to Java bytecode converters. Once the Java bytecode is generated, Soot is used to optimize the code. Dex2jar [99] also generates Java bytecode from Dalvik bytecode but does not use any external tool to optimize the resulting Java bytecode. Undx [115] was also a Dalvik to Java bytecode converter but seems to be unavailable.

Our approach, on the other hand, does not directly generate Java bytecode but Jimple code. To our knowledge no existing tool directly converts Dalvik bytecode to Jimple code. From the Jimple code, and since the Jimple code is Soot's internal representation of code, we can generate

Java bytecode as well. Analyzing an Android application with our approach requires only one step (i.e., Dalvik bytecode to Jimple) and not two steps as for all the existing approaches (i.e., Dalvik bytecode to Java bytecode and Java bytecode to Jimple).

7.1.2 Dalvik Assembler/Disassembler

Radare [96], Dedexer [95], Smali [62] are Dalvik disassemblers. They use their own representation of the Dalvik bytecode: they cannot leverage existing analysis tools. For instance, Dedexer generates a format close to Jasmin but containing Dalvik instructions which hinders Java bytecode generation. Our tool uses Soot's internal representation which allows existing tools to analyze/transform the Dalvik bytecode.

Androguard [40] is a Dalvik bytecode analyzer. It features a disassembler and modules to analyze the Dalvik bytecode. Redexer [104] and AsmDex [90] are Dalvik bytecode instrumentation frameworks. They enable to instrument Android applications at the bytecode level.

None of those approaches can perform advanced static analyses such as data-flow analysis. On the other hand, our approach converts Dalvik bytecode to Jimple, the internal representation of Soot. Existing tools performing data-flow analysis on Jimple code can leverage our tool to analyze Android applications.

7.2 Permission Map Extraction

7.2.1 On the Java Permission Model

While the Android permission model is different from the one implemented in Java, the following pieces of research present related and relevant points to put our contribution in perspective.

Koved et al. described a new static analysis [74] to generate a permission list for a Java2 program (in the Java permission model). An improved methodology was presented by Geay et al. [58]. We also use static analysis but in the context of Android which differs from a Java environment especially with respect to the binder mechanism linking Android API to services. As shown in our evaluation, the binder prevents off-the-shelf Java static analysis tools to resolve remote call to a service.

Pistoia et al. [103] presented a static analysis to identify portions of the code which should be made privileged. This issue does not arise in the Android framework since code is not privileged per se, the access control is instead done at entry points. This means that the Android framework designers must be careful of creating unique entry points protected by permission enforcement points, but does not impact our static analysis.

Role-based access control (RBAC) mechanisms are analyzed using static analysis by Centonze et al. [28]. When a protected operation manipulates data, this data should not be directly or indirectly accessible by a path not defined in the policy. If not, the operation is said to be *location-inconsistent*. The tool they developed can check whether or not an RBAC policy for JavaEE programs is location consistent or present some flaws. The Android system defines permissions which protect operation which in turn manipulate protected data. Our goal consists

of computing permission gaps which may reveal a violation of the principle of least privilege. Whether Android protected operations are location consistent is out of scope of our approach.

Also related to role-based access control, Pistoia et al. [102] formally model RBAC and statically check the consistency of a JavaEE-based RBAC system. We check that permission lists of Android applications respect the principle of least privilege. The concepts are the same (Android permissions could be approximated to roles, and we check which roles are needed at every point of the Android framework) but the target systems are not. Interestingly, we use a similar approach for solving the Binder problem as they do for solving the remote method invocation problem: instead of statically analyzing the Binder/RMI code which would not resolve the method, a mapping is computed from the call to a remote method to the remote method itself. A major difference though is that in the case of Android system services and context must be initialized beforehand to simulate a correct system state.

7.2.2 On the Android Permission Model

The Android security model has been described as much in the gray literature [49, 118] as in the official documentation [128]. Different kinds of issues have been studied such as social engineering attacks [66], collusion attacks [86], privacy leaks [59] and privilege escalation attacks [55, 36]. In contrast, our approach does not describe a particular weakness but rather a software engineering approach to reduce potential vulnerabilities.

However, we are not describing a new security model for Android as done by [89, 93, 42, 31, 26]. For instance, Quire [42] maintains at runtime the call chain and data provenance of requests to prevent certain kinds of attacks. In our work, we do not modify the existing Android security model and we devise an approach to mitigate its intrinsic problems.

Also, different authors empirically explored the usage of the Android model. For instance, Barrera et al. [11] presented an empirical study on how permissions are used. In particular, they used visualizing techniques such as self-organizing maps to identify patterns of permissions depending on the application domain and patterns of permission grouping. Other empirical studies include Felt's one [54] on the effectiveness of the permission model, and Roesner's one [109] on how users react to permission-based systems. While our approach also contains an empirical part, it is also operational because we devise an operational software engineering approach to tame permission-based security models in general and Android's one in particular.

Enck et al [48] presented an approach to detect dangerous permissions and malicious permission groups. They devised a language to express rules which are expressed by security experts. Rules that do not hold at installation time indicate a potential security problem hence a high attack surface. Our goal is different: we don't aim at identifying risks identified from experts, but to identify the gap between the application's permission specification and the actual usage of platform resources and services. Contrary to [48], our approach is fully automated and does not involve an expert in the process.

PScout [6] is a static analysis designed in parallel as ours. It also uses Soot but only relies on CHA and do not use Spark. Our works compares and validates part of their results in Section 4.6.2.

Finally, Felt et al. [53] concurrently worked on the same topic as us. They published a very first version of the map between developer's resources (e.g., API calls) and permissions. Interestingly, we took two completely different approaches to identify the map: while they use testing, we use static analysis. As a result, our work validates most of their results although we found several discrepancies that we discussed in details in Section 4.6.3. But the key difference is that our approach is fully automated while theirs requires manually providing testing "seeds" (such as input values). However, in the presence of reflection, their approach works better if the tests are appropriate. Hence, we consider that both approaches are complementary, both at the conceptual level for permission-based architectures, and concretely for reverse-engineering and documenting Android permissions.

Mustafa et al. [88] worked on the analysis of system services. Their approach is to extract a sub-call graph using a context-sensitive backward slicing method starting from permission check methods. Their analysis is more precise since they capture conditions under which permissions are checked. However, they only consider independent system services and do not handle RPC. We, on the other hand, start the analysis from the Android API entry points and handle services RPC links.

7.3 Data Leak in Android Applications

As far as we know, our tool called IccTA is the first approach to seamlessly connect Android components through code instrumentation in order to perform Inter-Component Communication (ICC) based static taint analysis. By using a code instrumentation technique, the state of the context and data (e.g. an *Intent*) is transferred between components. To the best of our knowledge, there is no other existing static approach to detect Android privacy leaks tackling the ICC problem and keeping state between components.

7.3.1 Static Analyses

There are several approaches using static analysis to detect privacy leaks. PiOS [44] uses program slicing and reachability analysis to detect the possible privacy leaks. TAJ [130] uses the same taint analysis technique to identify privacy leaks in web applications. However, these approaches introduce a lot of false positives. CHEX [84] is a tool to detect component hijacking vulnerabilities in Android applications by tracking taints between sensitive sources and externally accessible interfaces. However, it is limited to at most 1-object-sensitivity which leads to imprecision in practice. LeakMiner and AndroidLeaks state the ability to handle the Android Lifecycle including callback methods, but the two tools are not context-sensitive which precludes the precise analysis of many practical scenarios. FlowDroid [5] introduces a highly precise taint analysis approach with low false positive rate, but it does not identify ICC-based privacy leaks. IccTA performs an ICC-based static taint analysis by instrumenting the code of the original app while keeping the precision high.

ComDroid [30] and Epicc [92] are two tools that tackle ICC problem, but they mainly focus on ICC vulnerabilities and do not taint data.

SCanDroid [56] is a tool for analyzing ICC-based privacy leaks. It prunes all call edges to Android OS methods and conservatively assumes the base object, the parameters and the return value to inherit taints from arguments. This approach is much less precise than our tool since we model all the Android OS methods (except native methods) with our dummy main method in the control-flow graph. Another tool SEFA [136] also resolves the ICC problem. It performs a system-wide data-flow analysis to detect possible vulnerabilities (e.g., passive content leaks). Both SCanDroid and SEFA use a matching approach to analyze inter-component leaks. SCanDroid defines all the methods importing data to an app as *inflow* methods and all the methods exporting data from an app as *outflow* methods. Then, it matches the *inflow* and the *outflow* methods to connect two components. SEFA defines ICC methods as *bridge-sinks* to distinguish with the *sensitive-sinks*. It uses the *bridge-sinks* to match with other components and thereby connecting two components. As we mentioned before, the matching approach has some drawbacks compared to our instrumenting approach. Therefore, even if we were not able to evaluate SCanDroid and SEFA on DroidBench, it comes that IccTA is more precise by design.

AsDroid [69] and AppIntent [140] are another two tools using static analysis to detect privacy leaks in Android apps. Both of them try to analyze the intention of privacy leaks. Analyzing the leaking intention is out of scope of our approach. However, we think it is necessary to distinguish whether a privacy leak is intended or not. We take this as our further work.

7.3.2 Dynamic Analyses

Dynamic taint analyses techniques, on the other hand, track sensitive data at runtime. TaintDroid [45] is one of the most sophisticated dynamic taint tracking systems. It tracks flows of private data of third-party apps. CopperDroid [105] is another dynamic testing tool which observes interactions between the Android components and the Linux system to reconstruct high-level behavior and uses some special stimulation techniques to exercise the app to find malicious activities. Several other systems, including AppFence [67], Aurasium [138], AppGuard [8] and BetterPermission [13] try to mitigate the privacy leak problem by dynamically monitoring the tested apps.

However, those dynamic approaches can be fooled by special designed methods to circumvent security tracking [114]. Thus, dynamic tracking approaches may miss some data leaks and yield an under-approximation. On the other hand, static analysis approaches may yield an over-approximation because all the application's code is analyzed even code that will never be executed at runtime. Both approaches are complementary when analyzing Android applications for data leaks.

7.4 In Vivo Instrumentation of Bytecode

7.4.1 Monitoring Applications

Monitoring smartphone applications at runtime is an idea which recently emerged, due to the explosion of "mobile" malware and the increasing sophistication of mobile OS.

Bose et al. [23] aimed at detecting malware based on their behavior at runtime. For this, they added hooks in the Symbian OS emulator to track OS and API calls. In other words, malware detection is only achieved in the emulator, *in vitro*. On the contrary, we aim for malware detection in live user environments, *in vivo* and showed that is it is feasible in the mid-term.

Enck et al. [46] presented a runtime monitoring framework called TaintDroid, which allows taint tracking and analysis to track privacy leaks in Android. Their prototype is based on a modified version of the Dalvik virtual machine which runs Android applications. Similarly, Costa et al. [34] extends the Java virtual machine for mobile devices (Java ME) for adding runtime monitoring capabilities. On the contrary, our feasibility study indicates that it is possible to achieve runtime monitoring in an unmodified Android system.

Recently, Burgera et al. [27] presented an approach to detect malware based on collected operating system calls. Runtime monitoring can be done at different granularity levels. While the approach described by Burgera et al. is at the OS call level, we aim at providing runtime monitoring at the API call level, i.e. much more fine-grained and closer to the application domain of mobile applications.

Davis et al. [37] presented an Android Application rewriting framework prototype, and discussed its use for monitoring an application, and for implementing fine-grained Access Control.

Finally, Shabtai et al. detects malware based on the collection and analysis of various system metrics, such as CPU usage, number of packets sent through the Wi-Fi, etc. This is an indirect way of detecting malware behavior. Again, by monitoring API calls, we observe the application behavior directly. The empirical results presented in this thesis shows that this is actually possible.

7.4.2 Advertisement Permissions Separation

Shekhar et al. [120] proposed a new Android advertisement system that would allow to have an application and its advertisement module to run in different processes, and hence have a different permission set. This new system has to be manually inserted into the application during the development phase, since no automated application modification is provided.

Pearce et al. [101] made the case for an advertisement framework that would be integrated inside the Android platform. Every developer would be able to use the custom-built API that would be available on Android devices. This approach requires a modification of the Android framework, and that a given user has a device with a Android version embeding this advertisement system.

7.4.3 Permission Policy

Erlingsson et al. and Evans et al. [50, 51, 52] were the first to manipulate bytecode to weave a security policy directly in Java programs. Their Inline Reference Monitor (IRM) technique allows (1) to completely separate the program development from the policy definition and (2) to have a policy mechanism independent of the Java Virtual Machine on which the program is running. We also weave the security policy directly in Android applications, obtaining robust Android applications whose security policy is independent of the Android system on which they are running.

Closest to our work are two Dalvik bytecode manipulation systems: I-Arm Droid [38] and Mr. Hide [72]. The main difference is that our approach runs in vivo whereas theirs do not.

In vivo bytecode manipulation is also achieved by AppGuard [7, 9]. However, the approach is based on *dexlib* a bytecode manipulation library which does not offer an abstract representation like Jimple with Soot. Thus, more advanced reasoning on the bytecode (on graphs for instance) is difficult with their approach.

Redirecting methods of interest to a monitor is the basic of IRM. Von Styp-Rekowsky et al. present a novel approach by modifying the equivalent of Dalvik function pointers at runtime [125]. Such an approach reduces the overhead and could easily be adopted by our fine-grained permission system.

Xu et al. present Aurasium [137], another permission management system. It does operate at the level of C libraries and redirect low level functions of interest to the monitor. Operating at this low level makes it difficult to inject fake values and to differentiate between normal and Java-level security relevant operations.

Reddy et al. [104] claim that security of the Android platform would be improved by creating "application-centric permissions", i.e. permissions expressing what an application can do rather than current Android permissions that express what resource an application can use. They wrote a library that allows the so-called "application-centric permissions" to be managed. In addition, they started developing a tool called "redexer" whose aim is to automatically rewrite existing applications in order for them to use these new permissions.

Nauman et al. [89] extended the Android policy-based security model so that it can enforce constraints at runtime. The tool they created, called Apex, allows a user to express limits imposed to an application's use of any permission: For example, it becomes possible with Apex to grant the `SEND_SMS` permission to any given application while ensuring that this application will not be able to send more than a user-defined amount a text message each day. The user also has the possibility to change her mind, and to totally prevent the application from sending short messages. This is an important improvement over the stock Android OS because it allows users to specify a much finer-grained policy, instead of having to choose between either granting an application every permission it may request at installation time or not installing this application. However, this approach requires modifications deep inside the Android framework, and hence would need to be supported by Google and integrated into future versions of Android to be widely used.

Usually a permission protects access to a raw resource such as raw pictures taken from the camera. The approach presented by Jana et al. [71] allows to have an even finer-grained policy by filtering the raw data before it reaches an application. Suppose you install an application recognizing faces from picture taken by the camera. With the classical permission-based system, the application would be given the `CAMERA` permission. With their approach, the application could be given only a new `FACE_RECOGNITION` fine-grained permission. The raw data from the camera would then be filtered to only keep information about faces and not about the context where the picture was taken. Since the application has only information about faces, it cannot retrieve privacy-sensitive context information anymore (e.g., name of streets on walls, text written on black boards).

The work described in this document is unique and the thesis a significant contribution. We have improved the state-of-the art in the following areas: (1) Dalvik bytecode analysis with Dexpler, a tool to fully type Dalvik bytecode through the Jimple representation, (2) Permission-based analysis with a generic methodology to analyze permission-based framework and extract permissions. as well as a generic methodology to analyze permission-based applications and find permission gaps. (4) Data leak analysis for Android with IccTA and (5) Dynamic analysis with in vivo instrumentation of Android applications through two use cases.

Chapter 8

Conclusions and Future Work

This chapter is organized as follows. Section 8.1 summarizes the contributions of this PhD thesis. Section 8.2 describes potential directions for future work.

8.1 Conclusions

In this work we have analyzed the Android permission-based system and its applications from the security point of view. Android applications have a different structure and encoding compared to traditional Java applications. New tools and abstraction techniques are required to correctly analyze Android applications.

We presented Dexpler, a tool to convert Dalvik bytecode to Jimple which is the internal representation of Soot. Dexpler has been evaluated on more than 25 thousand apps containing 135 millions of methods and finds a correct typing for 99.99% of the methods.

Second, we analyzed the Android system itself to extract a mapping between API methods and permissions. Our tool models Android specificities such as system services communication and service identity inversion. It extracts a permission matrix with 4962 entry points linked to permissions. We analyzed how developers write the permission list for Android application. We presented a tool to statically analyze Android application to check for non-required permission. We found that over 18% of applications declare at least one useless permission.

Third, we analyzed Android applications to find data leaks. We presented IccTA, a tool to find data leaks between components of Android applications and between Android applications. IccTA outperforms existing tools performing taint-analysis of Android applications by reaching a precision of 95% and a recall of 82%.

Finally, we examined how dynamic analysis can be use in conjunction with results from static analysis. We presented a toolchain to automatically transform Android applications in vivo, i.e. directly on a device running Android. We used the toolchain in two prototypes. The first one allows the user to choose a fine-grained permission policy and relies on the permission mapping obtained by statically analyzing the Android system. The second one removes advertisements from applications.

8.2 Future Work and Open Research Questions

This section describes potential future research directions.

8.2.1 Framework Analysis

Extracting a Generic Model of Permission Checks

It could be worth exploring how to express permission enforcement as a cross cutting concern, in order to automatically add or remove permission enforcement points at the level of application or the framework, according to a security specification. Such work would answer the following questions: Are permission checks always inserted in the code the same way? Is it possible to bypass a permission check? How can a permission-based system be validated?

Improving Precision

We are now working on a modular approach that would be able to analyze native code and bytecode in concert and to combine the permission information from both. The precision of the results are limited by the precision of the call graph construction and by the type of Android components that are handled. Moreover, improving the precision could also improve the scalability of the approach since a more precise call graph could contain less edges and thus require less computation to extract permission information.

Empirical Evaluation Revisited

Also, since our methodology is generic we could apply it to other permission-based systems such as FirefoxOS or Google Chrome.

8.2.2 The Future of Static Analysis for Android Applications

Can we Still Automate Application Analysis?

We only use static analysis to analyze Android applications. However, there are ways to hinder static analysis such code obfuscation, intensive use of reflection to hide method calls, encryption of the bytecode, encryption of strings, execution of the code in a virtual machine, etc. During our analyses we would miss leaks in such applications. Taking a step back, we see that those techniques only slow down the process of statically analyzing Android applications. They force us to perform the analysis in two steps: (1) run the application to find out what method values and string values are and (2) run the static analysis again with information extracted in step one. It remains to be seen to what extent the first step can be automated. The reader may think that only malware applications use this kind of techniques to hide their malicious code, so it would be easy to detect malware in a group of applications. However, some authors of benign applications do not wish to make their code understandable and also use obfuscation techniques. As obfuscation techniques become more and more common, static analysis alone is not enough. New tools must be developed to dynamically analyze applications is conjunction to statically analyzing applications.

New Obfuscation Techniques

A good place to start is to have a look at the limitations of all tools analyzing Android applications. For instance, tools computing links between components may fail if strings used to create Intent objects are constructed using complex operations (e.g., concatenation, flow through multiple methods). Tools statically analyzing Android application may not work properly if deep access-path are often used in the application. Finally, those tools work at the bytecode level and do not analyze native code: transforming the bytecode to native code would make them useless. Like for a hash function that is easy to compute but very expensive to reverse, are there limitations that are easy to exploit for an attacker but very hard and expensive to analyze?

Leak Categorization

Our approach to find leaks in Android applications has the following limitation: it gives no information about the leaks it founds. Indeed, it does not tell whether a leak is a feature of the application (e.g., the application is for instance supposed to send the list of contact to your own remote server) or whether it is malicious (e.g., a malware is sending the list of your contact on remote server somewhere on the Internet). One way of resolving this problem would be to analyze the textual description of application and match it with leaks given by our approach. An Android application would then only be flagged if a leak does not match the description. Recently, there has been interest in approaches mining the description of applications [60, 97, 63].

Market of Safe Applications

Instead of mining applications' descriptions, developer could provide a list of expected behavior for every application. This list could then be checked dynamically when running the application (and the app be stopped if its behavior is inconsistent) or statically when analyzing the code. Furthermore, to simplify the analysis reflection methods, class loading and native code should be limited or banned from applications. Existing approaches extract specification from the official documentation or textual information coming with programs [127, 68, 98]. Furthermore, instead of asking developer to write the list of specifications, automatic techniques could be developed.

Scalable Analysis for a Market of Applications

Our approach to find leaks works on small sets of applications. An open research question is to find a scalable approach to analyze a market of thousands of applications. A possible direction would be to consider the analysis of the market as a two-step process. In the first step, every application would be analyzed individually and an abstraction of the application computed. The abstraction would depend on the kind of problem that one wants to solve (e.g., inter-application leak). In the second step, abstractions are combined to solve the problem. The following challenges would have to be resolved: How to precisely connect applications? How to handle data-flows in abstractions when communication is possible is both directions? There has been little work for Android in this topic [20].

In this dissertation, we have (1) presented Dexpler which allows the analysis and instrumentation of Android applications, (2) analyzed the Android framework to extract the permission map and we have leveraged the permission map to find permission gaps in Android applications, (3) developed IccTA to detect leaks in Android applications, and (4) shown that results of static analysis can be useful for dynamic analysis. In brief, in this thesis we have laid the foundations for analysis of the Android framework, Android applications and more generally any permission-based framework or application. Those analyses are the first step to understand permission-based systems and their applications. They pave the way for the creation of permission-based system secure by design in which security issues would be reduced to a minimum.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles techniques and tools*. Massachussets, 1988.

[2] Lars Ole Andersen. "Program analysis and specialization for the C programming language". PhD thesis. University of Cophenhagen, 1994.

[3] *Android (operating system)*. Url: `http://en.wikipedia.org/wiki/Android_(operating_system)`. February Last accessed: 2014.

[4] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. *Susi: A tool for the fully automated classification and categorization of android sources and sinks*. 2013.

[5] Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*. 2014.

[6] 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*. 2012, pp. 217–228.

[7] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. "Appguard-real-time policy enforcement for third-party applications". In: (2012).

[8] 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. Rome, Italy: Springer-Verlag, 2013, pp. 543–548.

[9] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. "AppGuardEnforcing User Requirements on Android Apps". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.

[10] David F Bacon and Peter F Sweeney. "Fast static analysis of C++ virtual function calls". In: *ACM Sigplan Notices* 31.10 (1996), pp. 324–341.

[11] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, and Anil Somayaji. "A methodology for empirical analysis of permission-based security models and its application to android". In: *ACM Conference on Computer and Communications Security (CCS 2010)*. Chicago, Illinois, USA, October 4-8, 2010, pp. 73–84.

[12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 2012, pp. 27–38.

[13] Alexandre Bartel, Jacques Klein, Martin Monperrus, Kevin Allix, and Yves Le Traon. *Improving privacy on android smartphones through in-vivo bytecode instrumentation*. Tech. rep. `http://hal.archives-ouvertes.fr/docs/00/70/03/19/PDF/article.pdf`. may 2012.

[14] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot". In: *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*. Beijing, China, June 2012.

[15] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. "Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android". In: *Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering*. Essen, Germany, September 2012.

[16] D. Elliott Bell and Leonard J. LaPadula. *Secure computer systems: Mathematical foundations*. Tech. rep. 1973.

[17] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. "Efficient local type inference". In: *OOPSLA*. Ed. by Gail E. Harris. Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA. ACM, 2008, pp. 475–492.

[18] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. "Efficient local type inference". In: *ACM Sigplan Notices*. Vol. 43. 10. 2008, pp. 475–492.

[19] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. August 1998.

[20] Amar Shirish Bhosale. "Precise Static Analysis of Taint Flow for Android Application Sets". PhD thesis. Carnegie Mellon University, 2014.

[21] Kenneth J. Biba. *Integrity considerations for secure computer systems*. Tech. rep. 1977.

[22] Eric Bodden. "Inter-procedural data-flow analysis with IFDS/IDE and Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. SOAP ' 12. 2012, pp. 3–8.

[23] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. "Behavioral detection of malware on mobile handsets". In: *MobiSys*. 2008, pp. 225–238.

[24] Jurriaan Bremer. *Abusing Dalvik Beyond Recognition*. Url: `http://archive.hack.lu/2013/AbusingDalvikBeyondRecognition.pdf`, Last accessed: May 5, 2014. 2013.

[25] Eric Bruneton. *ASM 3.0, a Java bytecode engineering library*. Url: `http://download.forge.objectweb.org/asm/asm-guide.pdf`. 2007.

[26] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Tech. rep. TR-2011-04. Apr 2011.

[27] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for Android". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. SPSM ' 11. Chicago, Illinois, USA: ACM, 2011, pp. 15–26.

[28] Paolina Centonze, Gleb Naumovich, Stephen J. Fink, and Marco Pistoia. "Role-Based access control consistency validation". In: *ISSTA 2006*, pp. 121–132.

[29] Patrick P. F. Chan, Lucas C. K. Hui, and S. M. Yiu. "DroidChecker: analyzing android applications for capability leak". In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. WISEC ' 12. Tucson, AZ, USA: ACM, apr 2012, pp. 125–136.

[30] 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. Bethesda, Maryland, USA: ACM, 2011, pp. 239–252.

[31] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. "CRePE: context-related policy enforcement for android". In: *Proceedings of the 13th International Conference on Information security*. 2011.

[32] Fernando J. Corbató. *The Compatible Time-Sharing System: A Programmer ' s Guide*. The MIT Press, 1963.

[33] Fernando J. Corbató and Victor A. Vyssotsky. "Introduction and overview of the Multics system". In: *Proceedings of the November 30December 1, 1965, fall joint computer conference, part I*. 1965, pp. 185–196.

[34] Gabriele Costa, Fabio Martinelli, Paolo Mori, Christian Schaefer, and . Thomas Walter R. "Runtime monitoring for next generation Java ME platform". Anglais. In: *Computers & Security / Computers and Security* 29.1 (2010), pp. 74–87.

[35] M. Dahm. "Byte Code Engineering". In: *Proceedings of Java-Informations-Tage (JIT ' 99)*. Düsseldorf, Deutchland, sep 1999, pp. 267–277.

[36] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. "Privilege escalation attacks on android". In: *Information Security*. Springer, 2011, pp. 346–360.

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

[38] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. "I-arm-droid: A rewriting framework for in-app reference monitors for android applications". In: *IEEE Mobile Security Technologies (MoST), San Francisco, CA* (2012).

[39] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of object-oriented programs using static class hierarchy analysis". In: *ECOOP ' 95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. Springer. 1995, pp. 77–101.

[40] Anthony Desnos. *Androguard*. Url: `https://code.google.com/p/androguard/`, Last accessed 2011.

[41] Anthony Desnos and Geoffroy Gueguen. "Android: From reversing to decompilation". In: *Proc. of Black Hat Abu Dhabi* (2011).

[42] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. "Quire: Lightweight Provenance for Smart Phone Operating Systems". In: *20th USENIX Security Symposium*. aug 2011.

[43] *DroidBenchBenchmarks*. http://sseblog.ec- spride.de/tools/droidbench/. Feb. 2014.

[44] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications." In: *The Network and Distributed System Security Symposium (NDSS 2011)*. 2011.

[45] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *OSDI*. Vol. 10. 2010, pp. 255–270.

[46] 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: *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[47] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A study of android application security". In: *Proc. USENIX Security ' 11*. San Francisco, CA, 2011, pp. 21–21.

[48] William Enck, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification". In: *Proceedings of the 16th ACM CCS*. New York, NY, USA, 2009, pp. 235–245.

[49] William Enck, Machigar Ongtang, and Patrick McDaniel. "Understanding Android Security". In: *IEEE Security and Privacy* (2009).

[50] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. Tech. rep. 2003.

[51] Ulfar Erlingsson and F. B. Schneider. “IRM Enforcement of Java Stack Inspection”. In: (2000), pp. 246–255.

[52] David Evans and Andrew Twyman. “Flexible Policy-Directed Code Safety”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 1999, pp. 32–45.

[53] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. “Android permissions demystified”. In: *ACM CCS 2011*.

[54] Adrienne Porter Felt, Kate Greenwood, and David Wagner. “The effectiveness of application permissions”. In: *Proceedings of the 2nd USENIX conference on Web application development*. WebApps ’ 11. Portland, OR: USENIX Association, 2011, pp. 7–7.

[55] Adrienne Porter Felt, Helen Wang, Alex Moshchuk, Steven Hanna, and Erika Chin. “Permission Re-Delegation: Attacks and Defenses”. In: *Proceedings of the 20th USENIX Security Symposium*. 2011.

[56] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. “SCanDroid: Automated security certification of Android applications”. In: *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/ avik/projects/scandroidascaa* (2009).

[57] Aleksandar Gargenta. *Deep Dive Into Android Security*. San Francisco, USA, 11 2011.

[58] Emmanuel Geay, Marco Pistoia, Takaaki Tateishi, Barbara G. Ryder, and Julian Dolby. “Modular string-sensitive permission analysis with demand-driven precision”. In: *ICSE*. IEEE, 2009, pp. 177–187.

[59] Clint Gibler, Jonathan Crussel, Jeremy Erickson, and Hao Chen. *AndroidLeaks Detecting Privacy Leaks in Android Applications*. Tech. rep. 2011.

[60] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. “Checking app behavior against app descriptions.” In: *ICSE*. 2014, pp. 1025–1035.

[61] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[62] Ben Gruver. *Smali: An assembler/disassembler for Android ’ s dex format*. Last accessed: March 20, 2012.

[63] Mark Harman, Yue Jia, and Yuanyuan Zhang. “App store mining and analysis: MSR for app stores”. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE. 2012, pp. 108–111.

[64] Rebecca Hasti and Susan Horwitz. “Using static single assignment form to improve flow-insensitive pointer analysis”. In: *ACM SIGPLAN Notices*. Vol. 33. 5. ACM. 1998, pp. 97–105.

[65] Nevin Heintze and Olivier Tardieu. “Demand-driven pointer analysis”. In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 24–34.

[66] Stefanie Hoffman. “Zeus Banking Trojan Variant Attacks Android Smartphones”. In: *CRN* (2011). http://goo.gl/xAEGr.

[67] 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*. 2011, pp. 639–652.

[68] Einar W Høst and Bjarte M Østvold. "Debugging method names". In: *ECOOP 2009–Object-Oriented Programming*. Springer, 2009, pp. 294–317.

[69] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. "AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction". In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. may 2014.

[70] IBM. *The T.J. Watson Libraries for Analysis (Wala)*. Last accessed: March 20, 2012.

[71] Suman Jana, David Molnar, Alexander Moshchuk, Alan M Dunn, Benjamin Livshits, Helen J Wang, and Eyal Ofek. "Enabling Fine-Grained Permissions for Augmented Reality Applications with Recognizers." In: *USENIX Security*. Citeseer. 2013, pp. 415–430.

[72] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Nikhilesh Reddy, Yixin Zhu, Jeffrey S. Foster, and Todd Millstein. "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android". In: (2011).

[73] A. T. Kearney. *European Mobile Industry Observatory 2011*. 2011.

[74] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. "Access Rights Analysis for Java". In: *ACM SIGPLAN Notices* 37.11 (nov 2002), pp. 359–372.

[75] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. 2011.

[76] Patrick Lam, Eric Bodden, Ondej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. oct 2011.

[77] William Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[78] Estimation lemma. *Android Inc. - Wikipedia, The Free Encyclopedia*. [Online; accessed 03-March-2014]. 2014.

[79] Estimation lemma. *HTC Dream - Wikipedia, The Free Encyclopedia*. [Online; accessed 03-March-2014]. 2014.

[80] Ondrej Lhoták. "Spark: A flexible points-to analysis framework for Java". In: (2002).

[81] Ondej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using Spark". In: *12th International Conference on Compiler Construction*. 2003.

[82] Ondej Lhoták and Laurie Hendren. "Scaling Java Points-to Analysis Using Spark". English. In: *Compiler Construction*. Ed. by Gorel Hedin. Vol. 2622. Springer Berlin Heidelberg, 2003. Chap. LNCS, pp. 153–169.

[83] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. "Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM. 2001, pp. 73–79.

[84] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "CHEX: statically vetting Android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS ' 12. Raleigh, North Carolina, USA: ACM, 2012, pp. 229–240.

[85] P. K. Manadhata and J. M. Wing. "An Attack Surface Metric". In: *IEEE Transactions on Software Engineering* 37.3 (may 2011), pp. 371–386.

[86] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. *Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems*. Tech. rep. 724. April 2011.

[87] Sharir Micha and Pnueli Amir. "Two approaches to interprocedural data flow analysis". In: *Program flow analysis: Theory and applications* (1981), pp. 189–234.

[88] Tanveer Mustafa and Karsten Sohr. *Understanding the Implemented Access Control Policy of Android System Services with Slicing and Extended Static Checking*. Tech. rep. 2012.

[89] 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*. 2010.

[90] Julien Nevo. *ASMDEX 1.0, a Dalvik bytecode engineering library*. Url: `http://asm.ow2.org/asmdex-index.html`. 2012.

[91] Damien Octeau, Somesh Jha, and Patrick McDaniel. "Retargeting Android applications to Java bytecode". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, p. 6.

[92] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis". In: *Proceedings of the 22nd USENIX Security Symposium*. 2013.

[93] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDanie. "Semantically Rich Application-Centric Security in Android". In: *Journal of Security and Communication Networks* (2011).

[94] Gabor Paller. *Dalvik opcodes*. `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`, Last accessed: March 20, 2012.

[95] Gabor Paller. *Dedexer*. Url: `http://dedexer.sourceforge.net/`. Last accessed: March 20, 2012.

[96] Pancake. "Radare, the reverse engineering framework". In: *Phrack magazine* 66 (2009).

[97] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *USENIX Security*. Vol. 13. 2013.

[98] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. "Inferring method specifications from natural language API descriptions". In: *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press. 2012, pp. 815–825.

[99] Panxiaobo. *Dex2Jar: Tools to work with android .dex and java .class files.* Last accessed: March 20, 2012.

[100] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof". In: *Proceedings of the 7th ACM european conference on Computer Systems*. EuroSys ' 12. Bern, Switzerland: ACM, 2012, pp. 29–42.

[101] Paul Pearce, Adrienne P. Felt, Gabriel Nunez, and David Wagner. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proceedings of AsiaCCS*. Seoul, Korea, may 2012.

[102] Marco Pistoia, Stephen J. Fink, Robert J. Flynn, and Eran Yahav. "When Role Models Have Flaws: Static Validation of Enterprise Security Policies". In: *ICSE*. 2007.

[103] Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. "Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection". In: *ECOOP*. 2005.

[104] Nikhilesh Reddy, Jinseong Jeon, Jeffrey A. Vaughan, Todd Millstein, and Jeffrey S. Foster. *Application-centric security policies on unmodified Android*. Tech. rep. 2011.

[105] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors". In: *EuroSec, April* (2013).

[106] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 49–61.

[107] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability". In: *POPL ' 95*. 1995, pp. 49–61.

[108] Dennis M. Ritchie and Ken Thompson. "The UNIX time-sharing system". In: *Communications of the ACM* 17.7 (1974), pp. 365–375.

[109] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. *User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems*. Tech. rep. MSR-TR-2011-91. 2011.

[110] Atanas Rountev, Ana Milanova, and Barbara G Ryder. "Points-to analysis for Java using annotated constraints". In: *ACM SIGPLAN Notices*. Vol. 36. 11. ACM. 2001, pp. 43–55.

[111] Mooly Sagiv, Thomas Reps, and Susan Horwitz. "Precise interprocedural dataflow analysis with applications to constant propagation". In: *TAPSOFT ' 95*. 1996, pp. 131–170.

[112] Jerome H. Saltzer. "Protection and the control of information sharing in Multics". In: *Communications of the ACM* 17.7 (1974), pp. 388–402.

[113] Jerome H. Saltzer and Michael D. Schroeder. "The Protection of Information in Computer Systems". In: *Proceedings of the IEEE*. 1975.

[114] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. "On the effectiveness of dynamic Taint analysis for protecting against private information leaks on android-based devices". In: *10th International Conference on Security and Cryptography (SECRYPT)*. 2013.

[115] Marc Schönefeld. "Reconstructing Dalvik applications". In: *Hack In The Box Security Conference* (2010).

[116] Thorsten Schreiber. *Android binder*. 2011.

[117] Patrick Schulz. *Dalvik Bytecode Obfuscation on Android*. Url: `http://dexlabs.org/blog/bytecode-obfuscation`, Last accessed: May 5, 2014. 2012.

[118] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. "Google Android: A State-of-the-Art Review of Security Mechanisms". In: *CoRR* abs/0912.5101 (2009).

[119] Marc Shapiro and Susan Horwitz. "Fast and accurate flow-insensitive points-to analysis". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 1–14.

[120] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating smartphone advertising from applications". In: *CoRR* abs/1202.4030 (2012).

[121] Stephen Smalley, Chris Vance, and Wayne Salamon. "Implementing SELinux as a Linux security module". In: *NAI Labs Report* 1 (2001), p. 43.

[122] Manu Sridharan and Stephen J Fink. "The complexity of Andersen ' s analysis in practice". In: *Static Analysis*. Springer, 2009, pp. 205–221.

[123] Angelos Stavrou, Jeffrey Voas, Tom Karygiannis, and Steve Quirolgico. "Building Security into Off-the-Shelf Smartphones". In: *Computer* 45-2 (2012).

[124] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, pp. 32–41.

[125] Philipp von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. "Idea: callee-site rewriting of sealed system libraries". In: *Engineering Secure Software and Systems*. Springer, 2013, pp. 33–41.

[126] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. "Practical virtual method call resolution for Java". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. 2000, pp. 264–280.

[127] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. "/* iComment: Bugs or bad comments?*". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 145–158.

[128] *The Android Developer ' s Guide, Last-accessed: 2011-09*. Last accessed: March 20, 2012.

[129] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. "Andromeda: Accurate and scalable security analysis of web applications". In: *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 210–225.

[130] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. "TAJ: effective taint analysis of web applications". In: *ACM Sigplan Notices*. Vol. 44. 6. 2009, pp. 87–97.

[131] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Etienne Gagnon Patrick Lam, and Phong Co. "Soot - a Java Optimization Framework". In: *Proceedings of CASCON 1999*. 1999, pp. 125–135.

[132] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. "Management of an Academic HPC Cluster: The UL Experience". In: *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014.

[133] David C. Walden, Tom Van Vleck, and F. J. Corbató. *The Compatible Time Sharing System (1961-1973): Fiftieth Anniversary Commemorative Overview*. IEEE Computer Society, 2011.

[134] John Whaley and Monica S Lam. "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *ACM SIGPLAN Notices*. Vol. 39. 6. ACM. 2004, pp. 131–144.

[135] Robert P Wilson and Monica S Lam. *Efficient context-sensitive pointer analysis for C programs*. Vol. 30. 6. ACM, 1995.

[136] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. "The impact of vendor customizations on android security". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 623–634.

[137] Rubin Xu, Hassen Saïdi, and Ross Anderson. "Aurasium: Practical policy enforcement for android applications". In: *Proceedings of the 21st USENIX Security Symposium*. 2012.

[138] 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. Bellevue, WA: USENIX Association, 2012, pp. 27–27.

[139] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. O ' Reilly Media, Inc., 2013.

[140] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 1043–1054.

[141] Yajin Zhou and Xuxian Jiang. “Dissecting android malware: Characterization and evolution”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. 2012, pp. 95–109.