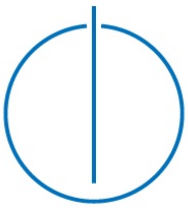
Technische Universität München

Lehrstuhl für Sicherheit in der Informatik

Fakultät für Informatik

Analysis and Mitigation of Security Issues on Android

Dennis Oliver Titze



Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Jens Großklags
Prüfer der Dissertation:	1. Prof. Dr. Claudia Eckert
	2. Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 22.11.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 27.02.2019 angenommen.

Acknowledgements

First of all I would like to thank my supervisor Prof. Dr. Claudia Eckert for giving me the possibility to work on interesting and scientific relevant problems at the Fraunhofer Institution AISEC. Moreover I would sincerely thank her for her encouragement and insights during the writing which were crucial for giving me the perseverance to finish this dissertation. I would also like to thank my second examiner Prof. Dr. Uwe Baumgarten.

Further I would like to thank my colleges at Fraunhofer AISEC for always making time for discussions, the many possibilities for collaboration, and their constructive feedback during the whole time of writing this thesis.

I would especially like to thank Dr. Julian Schütte for his constant encouragement and the many fruitful discussions we had, both during writing of publications and the dissertation.

Thanks!

Abstract

Mobile applications are ubiquitous in everyday life. But the questions, if they are secure, do not leak data or contain programming errors are still not solved completely. E.g., no technique exists to perform efficient data flow analysis on real world apps.

This thesis investigates the problem of automated application analysis and subsequent mitigation of found issues.

To perform app analysis, a framework for automated application analysis is proposed, which can handle dependencies between analysis techniques. The techniques in the framework are either static or dynamic, both having benefits and limitations if run on their own. Since the framework allows dependencies between analysis techniques, limitations of one technique can be mitigated by others, e.g., by using the output of static analysis in a subsequent dynamic analysis. The thesis shows which issues can be found with such techniques, and how they are applied to Android apps.

During the course of this thesis, multiple shortcomings in current approaches have been identified and improved:
Previous static data flow analysis of Android apps required a time-intensive calculation of entry points. This calculation is not necessary in the presented approach, thereby allowing faster analysis.
Since apps are comprised of multiple components, reconstructing transitions between them is an important analysis step which can be done using data flow analysis, or in a data flow agnostic manner. This thesis shows a data flow agnostic approach, and investigates how the accuracy and time and resource consumption changes.
Library code contained in Android apps is indistinguishable from code of the core functionality. The approach proposed in this thesis differentiates core code and library code in an obfuscation resilient manner by relying on precalculated signatures of libraries, allowing detection of the used library version with high accuracy.
To improve dynamic analysis, an approach to perform dynamic data flow analysis directly in an app is shown, which removes the need for a modified environment to perform the analysis. The approach modifies the analysed app to include the data flow analysis directly.

This thesis further shows how issues found during the analysis step can be mitigated, and which class of issues can be fixed in an automated manner or have to be fixed manually. Two instances of mitigations are shown in this thesis to illustrate which steps need to be taken to perform mitigation. Specifically, these mitigations ensure secure library loading, and prevent apps from leaking data.

Kurzfassung

Mobile Anwendungen sind aus dem täglichen Leben nicht mehr wegzudenken. Doch die Frage ob diese Apps sicher sind, also beispielsweise Datenlecks oder Programmierfehler beinhalten, ist nicht vollumfänglich geklärt. So existiert zum Beispiel kein Analyseverfahren um performante Datenflussanalyse von realen Apps durchzuführen.

Diese Dissertation untersucht das Problem der automatisierten App Analyse und anschließenden Korrektur der gefundenen Probleme.

Um die Analyse durchführen zu können wird ein Framework für die automatische Analyse vorgestellt, das Abhängigkeiten zwischen verschiedenen Analysetechniken abbilden kann. Diese Techniken sind statische und dynamische Analyseverfahren, welche beide Vor- und Nachteile besitzen wenn sie alleine angewendet werden. Durch die Kombination von Techniken können diese Nachteile abgeschwächt werden, beispielsweise indem die Resultate der statischen Analyse eine folgende dynamische Analyse steuern. In dieser Arbeit wird gezeigt, welche Probleme in Apps gefunden werden können, und wie die verschiedenen Techniken auf Android Apps angewendet werden.

Im Rahmen dieser Dissertation wurden verschiedene Schwachpunkte in bestehenden Ansätzen identifiziert und verbessert:
Bisherige Verfahren zur statischen Datenflussanalyse von Android Apps erfordern eine zeit- und resourcenintensive Berechnung von Startpunkten. Diese Berechnung ist in dem hier vorgestellten Ansatz nicht notwendig, was zu einer schnelleren Analyse führt.
Da Android Apps aus unterschiedlichen Komponenten bestehen, ist die Rekonstruktion von Übergängen zwischen diesen Komponenten eine wichtige Analyse. Die Rekonstruktion kann dabei mit Hilfe von Datenflussanalysen, oder ohne Datenflussanalyse erfolgen. In dieser Dissertation wird eine Analysetechnik ohne Datenflussanalyse vorgestellt, und betrachtet inwieweit sich die Genauigkeit und Geschwindigkeit verändern.
Ob Funktionalität in einer Android App durch Bibliotheken oder durch Code vom Entwickler ausgelöst wird, ist nicht ohne weiter Informationen ersichtlich. Der Ansatz der in dieser Dissertation vorgestellt wird, ermöglicht eine solche Unterscheidung auch nach Obfuskierung, mit Hilfe von vorab erstellten Signaturen von Bibliotheken.
Um die dynamische Analyse zu verbessern, wird ein Ansatz vorgestellt, der die dynamische Datenflussanalyse direkt in eine App integriert. Dies erlaubt es, die Analyse auch auf einer nicht modifizierten Plattform durchzuführen.

Darüber hinaus zeigt diese Dissertation, wie die Probleme die bei der Analyse gefunden werden, korrigiert werden können. Es wird gezeigt, welche Klassen an Problemen automatisch, und welche nur manuell korrigiert werden können. Anhand zweier Verfahren zur Korrektur von Problemen wird gezeigt, welche Schritte notwendig sind, um eine Korrektur durchzuführen. Konkret wird gezeigt, wie das Laden von Bibliotheken abgesichert werden kann, und die Datenflüsse in Apps verhindert werden können.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Sophisticated software analysis has been a topic for more than 50 years. For instance, data flow analysis was used by Hecht et al. [1] as early as 1970, and has been improved constantly. Recent research is still concerned with specifics of data flow analysis, which shows that the topic is not solved yet.

Over time, software grew more complex which also affected analysis techniques. These techniques needed to become more advanced to cope with new programming languages, new programming concepts and new environments.

In contrast to this increasing complexity, new software ecosystems have been introduced which simplify analysis to some extent. One example for such a software ecosystem is the Android Operating System (OS). The first version of this OS for smartphones was introduced in 2008 [2], and included several properties which are beneficial for software analysis. Namely, most of the functionality is written in Java and is compiled to dex bytecode which is clearly structured and can be easily transformed to several intermediate representations. Software on Android is packed into so-called *apps* which can be installed and run on the OS. As these apps are not encrypted on the smartphone, this allows detailed analysis.

Analysis of apps is of growing importance, as more than four billion people worldwide used smartphones in 2015 [3], with Android as clear market leader with more than 86% market share in Q3/2016 [4]. With 2.2 million apps in the official Google Play Store [5], it is obvious that there is a need for automated analysis of apps – not only for malicious apps, but also for apps which might contain programming errors or data leaks.

This thesis will investigate how different analysis techniques benefit from the properties of Android and which limitations still remain. The thesis will also investigate under which circumstances these properties allow automated mitigation of issues found in an app.

1.2 Properties Beneficial to Analysis on Android

Android already includes several features which are beneficial to app analysis, which will be used as preconditions for analysis and mitigation techniques shown in the following chapters:

1. Apps run in a well-defined execution environment, i.e., the functionality of the OS is fully known and can be exploited for analysis.
2. Calls to system functionality are encapsulated and occur via Application Programming Interface (API) calls to the Android framework, which acts as software middleware.
3. Apps are written in Java and compiled to dex bytecode which is clearly structured and contains a small number of different opcodes.
4. Third-party libraries besides core OS-functionality have to be included in the app.

Box 1.1: Properties Beneficial to Analysis on Android

Each of these properties is not absolute, as the Android platform includes functionality to circumvent them. For instance, apps on Android are currently allowed to include native code. This is no theoretical limitation, as the shown analysis techniques can also be applied to native code. In this thesis, the focus is the functionality of the dex bytecode, native code will not be analysed.

Section 1.4 will show how each property can currently be bypassed, and how far each of the above properties holds in reality.

1.3 Research Questions

This thesis investigates the following research questions:

1. How does an app analysis framework have to be designed to perform arbitrary automated analysis?
2. Which analysis techniques are possible for Android apps, how do they benefit from the properties of the Android OS, and which limitations remain?
3. How can the detected issues be mitigated? To which extent is it possible to remove issues in an automated manner, or with assistance of a developer?

This thesis therefore shows the possibilities and limitations of analysis and mitigation of security issues on Android.

In detail the research questions tackle these topics:

1. Analysis

   Operating on the compiled app, this thesis will show how issues in the app can be detected by showing which analysis techniques are possible on Android. It will further show in detail, which static and dynamic analysis techniques are possible, and which limitations remain.

   This thesis also shows how existing approaches can be improved further, namely concerning data flow analysis, component transition reconstruction, obfuscation-resilient detection of libraries, and dynamic data flow analysis.

2. Mitigation

   As issues are found in apps, the question arises how they can be mitigated. This thesis will investigate how issues can be categorized, depending on the ability to fix them in an automated or manual manner. The categories in this regard are: issues which can be fixed automatically in every case, issues which can be fixed after approval of a developer to fix the specific problems, and issues which need to be fixed manually.
   For each of these categories, this thesis will show which issues fall into the category, and how a process for fixing issues can be designed.

   Instances of mitigation processes shown in this thesis are the automated fixing of library verification, and the prevention of data flows.

1.4 Validity of Properties on Android

Whilst this thesis assumes the properties in Box 1.1 to be true in the following chapters, this is not completely the case for Android. This section will show if and how the properties can currently be circumvented on Android, and how widespread these circumventions are in real apps.

This is not a theoretical limitation of the presented approaches in this thesis, as all circumventions are either considered in the publications, or could also be considered by future work.

To show how far the properties currently hold for the current Android OS, 10.000 apps have been downloaded from the official Google Play Store and analysed how far they comply with the properties. The apps have been taken randomly in July 2016 from the 32 categories of the app store, which each offers 620 free apps.

Property 1) Apps run in a well-defined execution environment, i.e., the functionality of the OS is fully known and can be exploited for analysis.

Whilst it is true, that the functionality of the Android OS is fully known due to the fact that the OS is open-source, several components (e.g., drivers for the GPS, DRM, and sensors) are proprietary and not included in the open-source code of the OS.

Additionally, device vendors are allowed to modify the OS, to add their own functionality, or remove unwanted functionality from the core OS. This results in a modified Android

OS running on devices of different vendors. As apps might behave differently on those devices, the analysis results on one device might not reflect the results on a different device.

Figure 1.1 shows the current market share of smartphone vendors as of May 2017. This chart shows multiple vendors (i.e., Samsung, Huawei, OPPO, Vivo, and others) which sell smartphones containing Android as operating system. Each of these vendors can modify the operating system and add own apps, drivers, and other software components to the system.

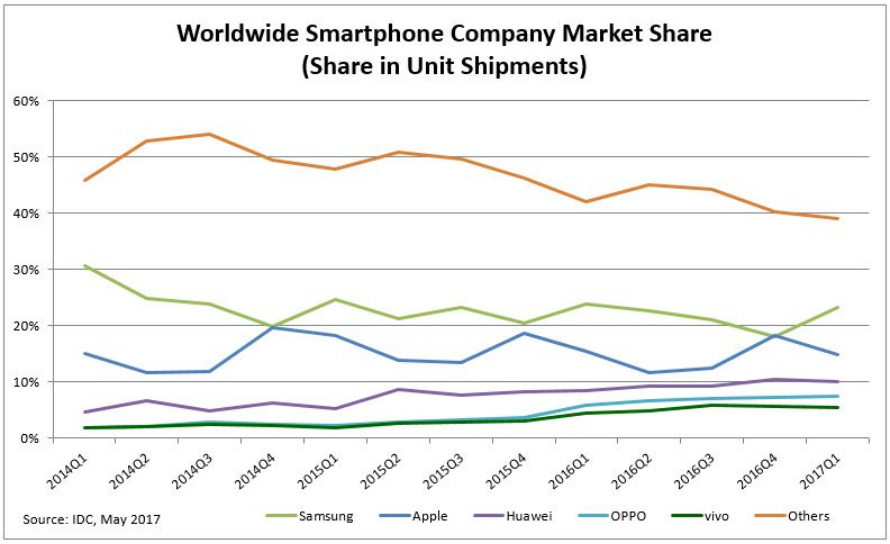


Figure 1.1: Worldwide Smartphone Company Market Share May 2017 [6]

Property 2) Calls to system functionality are encapsulated and occur via API calls to the Android framework, which acts as software middleware,

Access to the Android framework has to occur via defined API calls in Android apps. This API encapsulates system calls, including opening of files, network access, access to data on the phone.

An analysis of the 10.000 apps showed that 4350 apps (= 43,5%) contain native libraries. This is in line with other research, which shows that native code is included in 14% of non-popular apps and 70% of popular apps [7].

Property 3) Apps are written in Java and compiled to dex bytecode which is clearly structured and contains a small number of different opcodes.

In addition to code written in Java, apps on Android can include native code, which is compiled directly for the platform. Whilst some functionality is only available via the Android framework API (e.g., retrieval of contact data), other functionality can also be directly called from the native code (e.g., network access, and file access).

This is the case for all parts of an Android app which are not native.

Property 4) Third-party libraries besides core OS-functionality have to be included in the app.

On Android, third-party libraries can also be loaded dynamically, since code loading is allowed for any app. Such code can be loaded from files included in the app, and from remote locations. As this loaded code might not be available during analysis, the functionality in the loaded code is not analysed.

Analysing the 10.000 apps showed that 6197 apps (= 62,0%) included functionality which loads classes from a file or from a remote server via *PathClassLoader* or *DexClassLoader*.

Summary

This brief evaluation of how far Android currently complies with the properties shows that Android currently does not enforce any them. In reality, many apps include parts for which the properties do not hold. During analysis, this means that parts of the functionality are not analysed, or that the app might behave differently in reality than during analysis. E.g., during static analysis, only the available code is analysed. If the app loads code dynamically, this is typically not further analysed which can result in analysis imprecision. Approaches in current publications exist to mitigate these imprecisions, e.g., by combining static and dynamic analysis.

1.5 Contributions

This thesis makes the following contributions:

- a framework for automated analysis of apps which is focussed on allowing arbitrary combinations of analysis techniques and easy extensibility,
- an overview of possible analysis techniques showing the capabilities and limitations of the techniques,
- improvements to current analysis approaches:

  Data Flow Analysis: the presented approach calculates the data flows directly from all sources to the sinks of the app which is conceptually different to previous approaches. This allows faster analysis results, and can potentially find more data flows, if these do not occur on the control flow from an entry point of the app.

  Android Activity Flow Reconstruction: the approach shown in this thesis chooses a different approach than previous published work, as it reconstructs transitions in a data flow agnostic manner. The approach is less time and resource consuming than relying on data flow analysis at the cost of lower accuracy.

  Differentiation of Core Code and Library Code focusses on detecting the library name and version inside an app using obfuscation-resilient techniques. Compared to existing approaches, this allows detection of the used library version even in obfuscated apps with high accuracy.

  Data Leak Analysis by Injecting Targeted Taint Tracking modifies an app to include a data flow analysis directly in the app. In contrast to previous approaches, this does not require any changes to the underlying operating system, and can therefore be used on an arbitrary execution environment.
- a classification of issues in regards of the possibility to fix them automatically,
- a process for automatic issue mitigation in Android apps, and
- publications concerning mitigation techniques on Android which have not been proposed before:

  Library Loading is prevalent in most Android apps. The mitigation technique shows how library verification can be added to an Android app in an automated manner, to securely verify any library that is loaded by the app.

  Data Leak Prevention can be used as a mitigation technique for potential data leaks. To do so, data flow analysis and reaction to a data flow (e.g., prevention, or informing of the user) is added to the app in an automated manner.

In the context of this theses the following papers have been published in peer-reviewed conferences:

Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications [8]
Dennis Titze, Michael Lux, Julian Schütte
Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Sydney, Australia, 2017.

This publication shows a code analysis technique which can be used to distinguish core code of an app from code of included libraries. Section 4.6 is based on this publication.

Ariadnima - Android Component Flow Reconstruction and Visualization [9]
Dennis Titze, Konrad Weiss, Julian Schütte
published in Proceedings of International Conference on Advanced Information Networking and Applications (AINA), Taipei, Taiwan, 2017.

Ariadnima also performs code analysis on Android. Control flows between Android components are not configured explicitly anywhere in the app, but only implicitly by the logic of the app. To show these, Ariadnima proposes a technique to reconstruct such component flows in a data flow agnostic. Section 4.5 is based on this publication.

Preventing Library Spoofing on Android [10]
Dennis Titze, Julian Schütte
published in Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Helsinki, Finland, 2015.

This publication solves a typical problem on Android, which is the loading and executing of third-party libraries: if libraries are not packaged into the app, e.g., since they are system libraries, the authenticity of the code is often not verified. This publication shows how such a verification can be added to the app in an automated issue mitigation process in Section 5.5.

Apparecium: Revealing Data Flows in Android Applications [11]
Dennis Titze, Julian Schütte
published in Proceedings of International Conference on Advanced Information Networking and Applications (AINA), Gwangju, Korea, 2015.

Apparecium presents an improved data flow technique specifically for Android apps. The publication is shown in Section 4.4.

AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps [12]
Julian Schütte, Dennis Titze, and J. M. de Fuentes
published in Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Bejing, China, 2014.

AppCaulk combines static data flow analysis with modification of the app to prevent data flows in Android apps. Appcaulk is shown in detail in Section 4.7, where it is used as a modified dynamic analysis technique, which uses the modified app to verify the existence of data flows in an app. Further, Appkaulk can also be used in an automated mitigation process to prevent data flows which is shown in Section 5.6.

A Configurable and Extensible Security Service Architecture for Smartphones [13]
Dennis Titze, Philipp Stephanow, Julian Schütte
published in Proceedings of International Symposium on Frontiers in Networking with Applications (FINA), Barcelona, Spain, 2013.

This publication introduces a framework for app analysis on Android. The publication is basis for the framework shown in Section 4.2. The paper further introduced one possible strategy to execute apps during dynamic analysis, which is used in Section 4.3.3.6.

The following co-authored papers are not included in this thesis:

Practical Application-Level Dynamic Taint Analysis of Android Apps: Julian Schütte, Alexander Küchler and Dennis Titze, Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Sydney, Australia, 2017.

ConDroid: Targeted Dynamic Analysis of Android Applications: Julian Schütte, Rafael Fedler, Dennis Titze, published in Proceedings of International Conference on Advanced Information Networking and Applications (AINA), Gwangju, Korea, 2015.

1.6 Structure

The remainder of this thesis is structured as follows:

Chapter 2 gives the necessary background about the Android OS and Android apps.

Chapter 3 shows work related to this thesis, specifically to analysis of Android apps and mitigation of issues.

Chapter 4 goes into detail which analysis techniques can be applied to apps. In particular, Section 4.1 shows which threats can originate from apps, Section 4.2 introduces a generic framework for app analysis, followed by Section 4.3 which shows which analysis techniques are generally possible.
The following sections present new or improved analysis techniques: namely a data flow analysis in Section 4.4, activity flow reconstruction in Section 4.5, library detection in Section 4.6, and dynamic data flow analysis in Section 4.7.
Section 4.8 summarizes this chapter.

Chapter 5 shows how issues found during analysis can be mitigated. The chapter also shows two mitigation strategies, namely securing library loading in Section 5.5 and data leak prevention in Section 5.6.

Chapter 6 concludes this thesis.

CHAPTER 2

Background

This thesis shows different analysis and mitigation techniques applicable to apps. As the presented techniques are specific to Android, this chapter gives the necessary background on the Android platform and its applications.

2.1 Android Platform

Originally developed by Android Inc., Android is now an open source operating system developed by the Android Open Source Project (AOSP), lead by Google. It is based on a Linux kernel which has been adapted to run on mobile devices and extended to run the Android framework.

Although Android receives updates on a regular basis, and major releases were published every year since 2008, several core concepts remained since the beginning. The main concepts relevant for app analysis are Android's software architecture, the isolation of apps, code signatures and Android's permission system.

2.1.1 Software Architecture

Android's software architecture is designed in a modular manner. As shown in Figure 2.1, the core of the architecture is the Linux kernel containing drivers for the display, WiFi, audio, etc. Running above the kernel are libraries providing functionality to apps, abstracting the libraries of the kernel. Apps on Android typically perform functionality via these libraries and not directly through kernel components.

Also running on top of the Linux kernel is the Android runtime. Until Android 5.0, the app's bytecode was executed in a virtual machine called the Dalvik Virtual Machine (DVM). The DVM is comparable to a Java Virtual Machine (JVM), with several

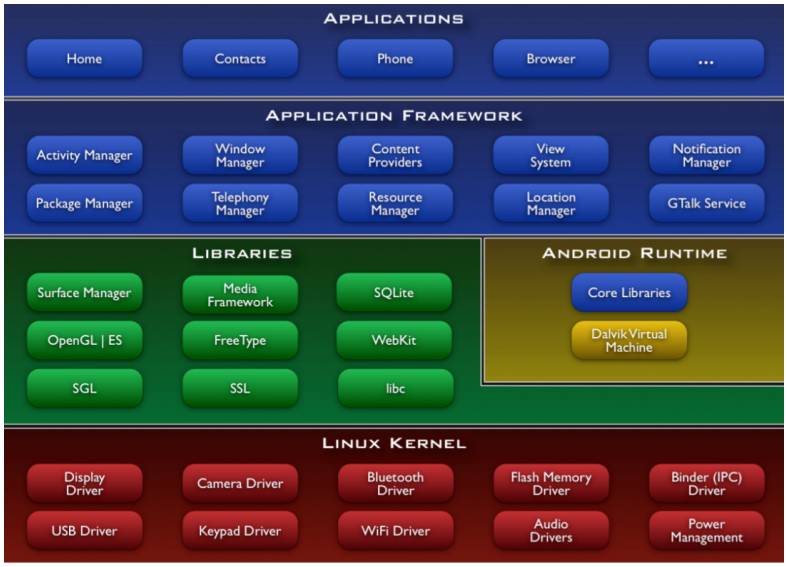


Figure 2.1: Android's Software Architecture [14]

differences (e.g., DVM is register-based, whereas a JVM is stack-based.). Starting with Android 5.0, the DVM has been replaced by the Android Runtime (ART) which compiles dex bytecode into native code upon installation of the app. Subsequently this native code can be executed as the app is run. More details on ART can be found in [15]. From an analysis point-of-view, the switch from DVM to ART does not change analysis techniques, as the app's code is still present as dex bytecode.

A further abstraction is introduced by the Android framework. This layer gives apps high-level access to functionality and resources of the phone.

Building upon this framework, apps can be installed and run on the system. Although apps are separated from each other, Android offers the possibility for communication between the apps via the so-called Binder component. This component resides in the Android kernel and allows Inter Process Communication (IPC) between apps through well-defined channels, which can also be secured via permissions. The exchanges messages are called *Intents*, and can contain information about the sender, receiver and additional data.

2.1.2 App Isolation

As apps are installed on the system, each is assigned a unique Linux User-ID (UID), and a data directory is created for each app. If the to-be-installed app is signed with the same certificate as an app already installed on the system, the UID of this app is reused. This can be used to easily share data between two apps of the same developer. This data directory is owned by the newly created UID.

As parts of the app are started, a new DVM is created for each app, running with this UID. Default Linux process separation is used to prevent other apps – having a different UID – from accessing the app's data directory.

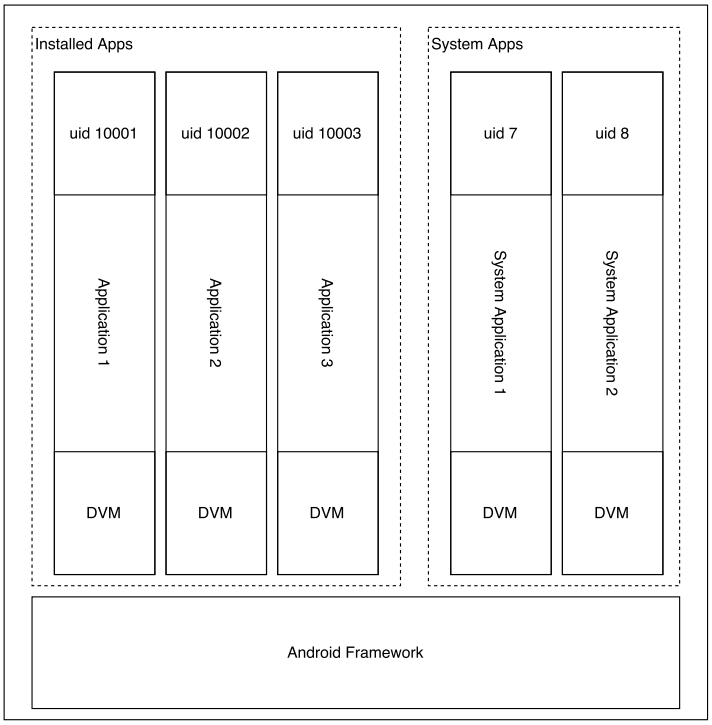


Figure 2.2: App Isolation

Figure 2.2 shows multiple apps running on the system, each with their own UID, and each running inside their own DVM.

2.1.3 Code Signatures

Apps on Android have to be signed, or they cannot be installed on the system. But different to e.g., Apple's developer program where all certificates used for distributing apps have to be issued and authorized by Apple [16], Android does not include any form of Certificate Authority (CA) to verify the authenticity of the certificate. Therefore, any developer can easily create a self-signed certificate to sign own apps. If the developer wants to distribute an update of the app, the updated app has to be signed with the same certificate as the original app.

The fact that certificates can be easily created and apps can be signed with arbitrary certificates allows analysis techniques to modify apps, and sign them with a new certificate. This is used in Section 4.3.4, to perform modified dynamic analysis.

2.1.4 Android's Permission System

Permissions are used in the Android OS to restrict access to functionality or data. A permission is a constant string consisting of a namespace and a name for the permission, e.g., `android.permission.INTERNET` or `com.android.launcher.permission.INSTALL_SHORTCUT`, whereas all permissions of the android core framework start with 'android.permission'. Apps can define their own permissions with arbitrary names to restrict access to its components.

Inside the Android OS, permissions are enforced on different levels: in the Android framework, directly inside apps, or inside kernel components.

During development of an app, all permissions required by the app have to be specified in the AndroidManifest. Before Android 6.0, the user had to grant all permissions at installation time, thereby either granting all permissions, or not installing the app. Since Android 6.0, the granting of permissions can be performed on a per-permission basis, and apps can request the permissions specified in the manifest during runtime. Permission usage is therefore more fine-grained: an app can ask the user to grant or revoke a certain permission, e.g., to access the current GPS position to determine the location. If the user grants the permission, the app can access the respective functionality. Otherwise, the app is informed about the denial of the permission, and can react accordingly. An app can e.g., show localized messages if this permission is granted, and generic messages if the permission is denied.

2.2 Android Apps

Apps in the Android ecosystem are distributed via different channels. Official app markets (e.g., the Google Play Store, or Amazon's app store) contain several million apps in total. Alternatively, apps can also be distributed outside these markets, e.g., directly from the developer.

This section will introduce the most important components of an app relevant for analysis.

2.2.1 App Structure

An app for the Android OS is bundled in one file with the file extension '.apk', which is a zip-packed file containing the app's functionality, meta data, and resources as shown in Figure 2.3. In detail, these contents are:

AndroidManifest.xml: the app's manifest file specifies the app's name, its requested permissions, its components, and several other meta data. See Section 2.2.4 for details.

META-INF/: for each file included in the app, its filename and SHA1 hash are stored in the file 'MANIFEST.MF'. The association of filename and corresponding hash is hashed again using SHA1 and stored in 'CERT.SF'. Finally, this file is signed with

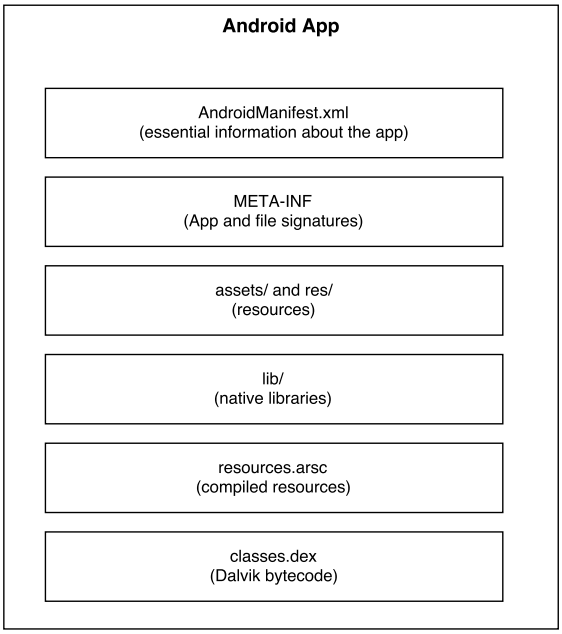


Figure 2.3: Android App Structure

the developer certificate, and the signature stored in 'CERT.RSA'. Using these three files, the Android OS can verify that no file was manipulated, and each file originates from the developer.

assets/, res/ and resources.arsc: the app can contain additional resources (e.g., images, videos, fonts). These resources are stored – depending on their type and how they are accessed – in one of these locations.

lib/: apps on Android can contain native libraries which are contained in this folder. These libraries can be loaded and called from the dex bytecode.

classes.dex: the dex bytecode of the app containing the core functionality of the app, including the functionality of the components defined in the AndroidManifest.

2.2.2 Components

On Android, four different types of components can be included in an app:

Activities represent a visible screen of the Graphical User Interface (GUI) of the app. Typically, each screen of the app is an own Activity, and transitions between these activities exist, e.g., as the user presses a button inside an Activity.

Services are background processes without a visible component. Services are typically used for longer running tasks, e.g., downloading of files.

Content Providers are components of the app which offer a CRUD-interface to other apps to *C*reate, *R*ead, *U*pdate, and *D*elete data of the app. This can be used to access internal data of the app.

Broadcast Receivers react to system-wide broadcasts. These broadcasts are sent as Intents by the system and indicates different events, e.g., change of battery charge, change of network availability, or that the device is unlocked.

For each of these components, the app can define and require its own permission. A permission can e.g., be required to access data through a content provider of the app.

Each of these components has its own lifecycle on Android, and different components (of the same or of different apps) can be active in parallel in different states.

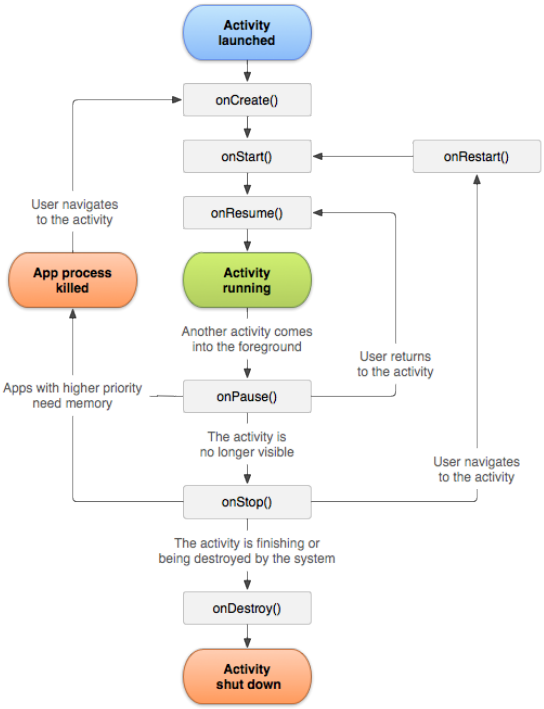


Figure 2.4: Android Activity Lifecycle [17]

Figure 2.4 shows the lifecycle of an Activity: as the app is started, the method *onCreate*, *onStart*, and *onResume* of the Activity's class are executed in this order. Once these are finished, the Activity is considered started. Once a different Activity comes to the foreground, *onPause* is called. In this state, the Activity can then be killed by the OS if memory is running low, or resumed if the Activity is sent to the foreground again.

This example shows, that an Activity not running in the foreground can be in one of many states.

2.2.3 Intents

On Android, 'an Intent is a messaging object you can use to request an action from another app component' [18], i.e., Android's main inter-app communication technique.

Intents can either be *explicit* or *implicit*. Explicit Intents are targeted at an exact recipient by specifying the receivers' fully qualified class name. Implicit Intents declare an action to be performed, without stating who exactly should perform it. An example for this is the action 'ACTION_DIAL'. If this is set in an Intent, the sending app wants to start any app which can perform the dial action. If such an app exists, it is started, if multiple exist, the user can choose which to use.

Main use cases of Intents are:

- to start an Activity,
- to start a Service, and
- to deliver a Broadcast.

In each of these cases, the Intent can contain arbitrary additional data, which is handed to the receiver by the Android OS.

2.2.4 AndroidManifest

A mandatory file of each Android app is its *AndroidManifest.xml*. This file contains essential information for the Android OS, without which no app could be installed or run.

```
1  <?xml encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3            package="com.example.test"
4            android:versionCode="1">
5    <uses-permission android:name="android.permission.INTERNET" />
6    <application
7        android:icon="@mipmap/ic_launcher"
8        android:label="Test␣Application">
9      <activity android:name=".MainActivity">
10       <intent-filter>
11         <action android:name="android.intent.action.MAIN" />
12         <category android:name="android.intent.category.LAUNCHER" />
13       </intent-filter>
14     </activity>
15   </application>
16 </manifest>
```

Listing 2.1: Example AndroidManifest.xml

Listing 2.1 shows an example of such a manifest file. Line 3-4 specify the name of the app (its package name), and the version of the app. This example app requires access to the Internet, which is requested as permission in Line 5. Line 7-8 specify the app's icon, and the human readable name of the app.
Further, the manifest has to specify which components are included in the app. In this example, only one component – an Activity – is included. This Activity is implemented

in the class *MainActivity*. The intent-filter of the Activity tells the Android OS that this Activity should be started once an Intent with the action and category shown in Line 12-13 is sent to the app. In this example, this is a predefined Intent sent to the app once its icon is clicked, i.e., as the app is started.

More information about the contents of an AndroidManifest.xml can be found in [19].

2.2.5 Intermediate Representation

The dex bytecode of the app is contained in the file *classes.dex*. To allow easy analysis and readability of this bytecode, it can be transformed into different Intermediate Representations (IRs). The two most prominent ones are Smali and Jimple.

2.2.5.1 Smali

Smali is an intermediate representation generated by the disassembler 'Smali', loosely based on Jasmin's/dedexer's syntax [20]. It supports the full functionality of dex bytecode and can be easily read or further parsed by an analysis technique.

```
1  .method public test()LA;
2      .locals 1
3
4      .prologue
5      .line 121
6
7      const/16 v0, 0x41
8      invoke-static {v0}, LTest;->generate(I)LA;
9      move-result-object v0
10
11     return-object v0
12 .end method
```

Listing 2.2: Example Smali Method

Listing 2.2 shows an example of a method *test* which calls the static method *generate* with the constant $0x41$ and returns the resulting object of type A. A full documentation of all possible instructions is shown in [21].

2.2.5.2 Jimple

A different IR is called Jimple. This is one of the possible representations used by the Soot static analysis framework [22]. Android apps can be transformed to Jimple using Dexpler [23].

Jimple has the following properties [24]:

- it uses 3-address code, i.e., each instruction has at most two operands,
- no stack is used, and
- all local variables are explicitly declared and typed

Listing 2.3 shows the same code as in the Smali example converted to Jimple IR. Line 2 shows that in contrast to Smali, Jimple registers possess a type, and the assignment of the constant – which is visible in the Smali code – is missing in Jimple. Instead the constant is directly used in the method call. Dexpler can perform this simplification, as the constant is only used in this location.

```
1  public A test() {
2      A $r1;
3
4      $r1 = staticinvoke <Test: A generate(int)>(0x41);
5      return $r1;
6  }
```

Listing 2.3: Example Jimple Method

For static analysis, Jimple has to be used for all analysis techniques building on Soot, whereas either Jimple or Smali can be used for manual analysis or newly developed analysis techniques. Studies have shown, that Smali 'most accurately preserves and provides the closest reflection of the original program behaviors.' [25]

CHAPTER 3

State of the Art in Analysis and Remediation

This chapter will give an overview about the state of the art in analysis and remediation of issues in apps.

3.1 App Analysis

Analysis of apps can be separated into three topics: analysis frameworks, static analysis and dynamic analysis. This section will introduce the current state of the art in each topic.

3.1.1 Frameworks

Several frameworks for analysis of Android apps have been proposed in the past. The most prominent ones are AppsPlayground [26], Mobile-sandbox [27], AASandbox [28], Andrubis [29], Marvin [30], and AppAudit [31]

AppsPlayground [26] integrates different dynamic analysis components to detect privacy leaks and malicious functionality in Android apps. This approach by Rastogi et al. executes an app in a virtual environment, and applies a strategy to execute the app. During its execution, the behaviour of the app is analysed and subsequentially evaluated. Their approach is designed in a modular manner which allows the addition of further dynamic analysis components.

Mobile-sandbox [27], AASandbox [28], Andrubis [29], Marvin [30], and AppAudit [31] are similar in the sense that all of them include static followed by dynamic analysis techniques. All of the approaches can use results from the static analysis in the subsequent dynamic analysis. This is e.g., used to increase the coverage of the dynamic tests, by starting components detected in the static analysis.

Paranoid Android [32] applies app security checking on 'security servers' in the cloud. The system records information on the phone to replay the execution on the security servers. Different analysis techniques can be applied to check an app, e.g., virus scanning, or dynamic taint analysis. Once such an issue is detected, the original device state can be restored using a previously stored replica.

3.1.2 Static Analysis

3.1.2.1 Meta Data Analysis

Signature-based Analysis

DroidAnalytics [33] uses 'a multi-level signature [...] based on [the app's] semantic meaning at the opcode level'. Zheng et al. create three different levels of signatures: on methods, classes, and application. The different levels allow their approach to be able to detect known malicious apps after repackaging or after applying various obfuscation techniques.

AndroSimilar [34] and DroidOLytics [35], both published by Faruki et al., create signatures over statistical robust features to detect unseen malicious apps. Their approach creates signatures of known malicious apps and checks new apps against these signatures. According to their evaluation, they are able to detect repackaged malicious apps with high accuracy and less than three percent false positive rate.

Semantic Analysis

Different publications focus on analysing the permissions of an app to deduce its behaviour.

Huang et al. [36] use the permissions of the app as well as easily extractable features from the app package (e.g., number of included executable files, number of requested permissions, number of required permissions) to train different machine learning algorithms. Their best classification algorithm (State Vector Machine) has a high precision of over 84%.

Sato et al. [37] detect Android malware solely by analysing the app's manifest. They extract specific information from the manifest (e.g., Permissions, and information about registered Intents) and calculate a 'malignancy score' based on the occurrence of known malicious strings. By using this score, they are able to detect 87.5% of the malicious apps in their test set.

Aung et al. [38] also use the permissions of the app as features for machine learning. Their algorithm is able to correctly classify apps as malicious or benign in at least 88% of cases over different sample sets. Similarly, PUMA [39] and MAMA [40] – both from the same authors – also use the permissions as features and evaluate different machine learning algorithms. Their best algorithm is able to achieve an accuracy of over 86%.

DroidRanger [41] analyse the permissions of an app to perform a preselection of apps which can include potentially malicious functionality. I.e., only apps requesting a permission which could be used for malicious actions (e.g., the sending of SMS) will be analysed further.

Several approaches (e.g., [42, 43, 44, 45, 46]) have been published which rely on identifiers in the app, or the package structure to detect libraries in apps. As this information can trivially be changed by obfuscators, these approaches cannot detect libraries with a high accuracy.

3.1.2.2 Code Analysis

Instruction Level Analysis

DroidAPIMiner [47] analyses the frequency of specific API calls included in the app's bytecode as well as the parameters and package information of the calls. Analysing a large set of benign and malicious apps, they determine which calls and parameters are more frequent in malicious than in benign samples. Using machine learning algorithms trained with this data, they are able to achieve 99% detection accuracy.

Drebin [48] analyse both meta data and the bytecode of an app, specifically the occurrence of restricted API calls (i.e., calls requiring a permission), as well as network addresses contained in the bytecode. They use machine learning to classify apps as malicious or benign. Their classifier is able to detect 94% of the malware.

Juxtapp [49] calculates hashes over sequences of instructions. These hashes are then compared with those of new apps, to detect the same functionality. Juxtapp uses this technique to detect code reuse, known malware and repackaged apps.

WuKong [50] and LibRadar [51] perform instruction level analysis of apps to detect libraries contained in the app. Both approaches rely on frequencies of Android API calls to detect the libraries included in the app. The approaches are able to detect which library is contained, but do not state which version of the library is used.

An approach which is able to determine the version of the library was published by Backes et al. [52]: a library database is created beforehand, against which apps are checked. The approach chosen by Backes et al. abstracts method calls into a more generic format used for comparison. Their approach relies on the package structure and therefore fails if even trivial obfuscation techniques are applied which e.g., move all classes into one package.

Control flow analysis

Grace et al. use control flow analysis to detect capability leaks (c.f., Section 4.3.2.4) in stock Android OS images [53]. In particular, they follow execution paths along possible control flows to detect if such a path exists from an exposed interface of the app to an API call which requires a permission.

Yang et al. extend a typical call graph with edges for callbacks, which are a fundamental part of the Android system [54]. This extended call graph is calculated by performing a context-sensitive control flow analysis to determine the possible sources and targets of callback functions.

EdgeMiner [55] analyses the whole Android framework to determine implicit control flow transitions through the framework. These reconstructed transitions are more accurate than previously manually specified callback transitions through the framework.

SMV-hunter [56] detects typical SSL/TLS implementation vulnerabilities in Android apps. This is done by searching for typical vulnerable implementations (e.g., an implementation which does not verify the server signature). In the next step, a control flow analysis determines, which entry points to the app can lead to such a vulnerable implementation. This information is then used for a following dynamic analysis, to confirm the vulnerability.

Bartel et al. [57] and Au et al. [58] use – amongst other techniques – control flow analysis to determine which API call requires which permission. This analysis is important, as the required permissions for each API call are not well documented, but needed both for development of apps as well as for different analysis techniques.

Desnos [59] use the control flow of an app as part of their input to generate a signature of the app. This signature can then be used to determine to which degree two apps are similar, e.g., to detect piracy. Gascon et al. [60] and Dendroid [61] use a similar approach to detect malicious code parts in an app by performing similarity analysis of two app. This similarity is based on different factors including the similarity of the control flow graph.

On Android, not only the control flow between functions can be analysed, but also the flow between components. This can e.g., be one app screen following the next screen, or a service being started once the user clicks a button. These techniques also require control flow analysis, as such component transitions are started by functions of the Android OS.

Component transitions often rely on so-called Intents, which specify the target which should be started. Dienst et al. [62] reconstruct such Intents by using the AndroidManifest and by performing code analysis on the dex bytecode.

Dexteroid [63] uses Android lifecycle models and a callback analysis of the app's bytecode to reconstruct transitions between Activities. Another more precise reconstruction is proposed by Octeau et al. called Epicc [64]. Epicc reduces Intent reconstruction to a generic data flow problem, resulting in a precise reconstruction.

Data flow analysis

The most prominent example for static taint analysis on Android is FlowDroid [65], an extensive data flow analysis framework built on top of Soot [66]. Soot originated as a Java optimization framework and has been extended to perform static taint analysis and to handle Android apps by translating the apps bytecode to Soot's internal representation [23]. FlowDroid implements an extensive taint propagation logic, covering different lifecycle callbacks. Since it builds upon Soot, it requires a entry point analysis to gather all event-triggered callbacks in a single entry method. FlowDroid requires the code of the app to be translated to one of Soot's internal representation, namely Jimple. This transformation is done by Dexpler [23].

Woodpecker [67] uses static taint analysis to detect capability leaks in stock apps. It uses its own data flow tracking implementation, and relies on entry points into the app. CHEX [68] use the same approach in their data flow tracking to detect potential component hijacking vulnerabilities.

DroidChecker [69] uses static taint checking to identify data paths responsible for capability leaks, but Chan et al. do not explain their taint tracking algorithm in detail.

Klieber et al. [70] use data flow analysis on a set of apps by building on the analysis of FlowDroid [71] and Epicc [64]. During a first phase, data flows in one app are analysed. In a second phase, data flows of multiple apps are combined, to determine if multiple apps collude in a data flow, e.g., via Intents.

Apposcopy [72] designed a specification language, which allows to specify semantic properties which are then searched for in an app. Such properties can be a specific data or control flow, resembling malicious activity. Apposcopy also builds upon Soot [22] to implement its data and control flow analysis.

Barros et al. [73] perform data flow analysis to resolve implicit and indirect control flows. To e.g., resolve Java Reflection, Barros et al. perform a data flow analysis to determine possible values of the variables defining the called methods. Using this approach they are able to determine the exact reflectively called method in 54% of their analysed cases.

ComDroid [74] checks if Intents are sent in a safe way by performing a data flow analysis and determining if data is sent with the Intent, and if the Intent can be intercepted by an attacking app (e.g., if not explicitly sent to the receiver).

AndroidLeaks [75] and ScanDal [76] perform static privacy leak detection in apps by applying data flow analysis. CHEX [68] use data flow analysis to find different vulnerabilities in apps, including data theft and capability leaks. Capability leak detection by applying data flow analysis is also performed by DroidAlarm [77].

Symbolic Execution

SymDroid [78] is a symbolic executor which works directly on dex bytecode. SymDroid not only models the bytecode, but also properties of the Android system, including its lifecycle models. SymDroid can be used to determine under which conditions an API call which requires a specific permission is performed.

Micinski et al. [79] propose a system where apps are checked against configurable policies using symbolic execution. A policy definition can e.g., state that certain data may only be sent to a server if a button is pressed. The symbolic execution is based on SymDroid [78].

Mirzaei et al. [80] use symbolic execution to generate test cases with high code coverage.

Riskranker [81] detects if an app contains dangerous behaviour like sending SMS. They use symbolic execution to distinguish between legitimate and malicious dangerous API calls. They argue that a legitimate call includes callback functions (e.g., clicking a button). If such a callback call is missing, the API call is classified as potentially malicious. Similarly, AppIntent [82] uses symbolic execution to determine if transmission of sensitive data is initiated by a user, or done maliciously in the background.

3.1.3 Dynamic Analysis

The most prominent dynamic analysis technique is TaintDroid [83]. This technique performs dynamic data flow analysis by modifying the Dalvik VM and the Android framework. On this modified framework, an app can be executed, and several hard coded sources and sinks will be monitored for data flows between them. Found data flows are written to the Android log, or shown to the user.

TaintART [84] updates TaintDroid [83] to work with the newly introduced ART. ART introduced a ahead-of-time compilation of the app's code, which renders TaintDroid's approach of modifying the Dalvik VM obsolete. TaintART solves this issue by modifying the compiler which translates the bundled app to the natively executed app on the device. TaintART is able to track multiple taint tags, and includes a policy enforcement framework, which can e.g., block certain data flows, or notify the user. As their approach modifies binaries of the execution environment, these binaries have to be included or updated on any device which should run TaintART.

VetDroid [85] analyses where an app uses a specific permission, e.g., as the app uses an API call requiring a permission to retrieve some resources. Such resources are further tracked through the app to detect if these resources are leaked to a sink. This is performed by a data flow analysis implemented directly in the Dalvik VM.

DroidScope [86] analyses an app in a environment and monitors system calls and calls inside the Dalvik VM. Using such monitoring, they are able to reveal the behaviour of the analysed app. Their system is also capable of performing dynamic taint analysis, e.g., to determine if an app leaks sensitive information.

CopperDroid [87] observes an app on system call level, i.e., outside the Dalvik VM, by modifying the virtual environment the app runs in. This approach is able to analyse the behaviour of an Android app in detail, by reconstructing complex objects, IPC, and Remote Procedure Calls (RPCs). According to the authors, IPC and RPC are sufficient to 'understand the fundamental actions Android malware perform' although they do not represent all fine-grained behaviour of an app.

AASandbox [88] implements a system for static and dynamic analysis of apps. Their dynamic analysis is based on the execution of an app in a sandbox, i.e., the Android emulator, and monitoring of the app's behaviour is also based on monitoring system calls. Their system does not reconstruct IPC or RPC, and only focusses on the actual executed system calls.

Min et al. [89] propose a framework to hook system calls, i.e., to add custom logging functionality to specific system calls. This can then be used during dynamic analysis to record these calls and determine the behaviour of the app.

Andlantis [90] designed a scalable dynamic analysis system which allows scanning of over 3000 apps per hour. The analysis capabilities are limited to analysis of the traffic generated by the app.

Andromaly [91] monitors various features and events of the execution of an app on a devices and applies machine learning algorithms to classify apps as malicious or benign. They assume 'system metrics such as CPU consumption, number of sent packets through

the Wi-Fi, number of running processes, battery level etc. can be employed for detection of previously unencountered malware', which they show by performing an evaluation of their approach. They achieve an accuracy of 99.7% for classifying an app as tool or game, but do not go into detail how to differentiate between malicious and benign apps.

Marvin [30] combines static and dynamic features to classify apps as benign or malicious using machine learning algorithms. They use several features, including file operations, network activity and detected data leaks, to correctly classify 98.24% of their data set. DroidWard [92] extends Marvin [30] and analyses 15 features collected during the execution of an app to determine if an app is malicious. Their approach e.g., uses the number of distinct sensitive API calls, number of API calls per time, and requests for root-access. Using these features, they are able to correctly classify 98.54% of the analysed apps. Their test set was taken from a Chinese app store, and Virustotal [93] was used to label the downloaded apps as malicious or benign, i.e., as ground truth.

Symbolic execution is added to the dynamic analysis in Condroid [94] to drive the execution of an app during its dynamic execution along a predefined path. The technique performs bytecode instrumentation to determine the path conditions which are on the current execution path. If the path did not reach the previously configured parts of the code, the last condition is inverted, and a subsequent dynamic execution of the app is started. In this execution, the variables for all conditions are set using instrumentation according to the constrain system which is created and solved in the previous step. Using this technique, Condroid is able to automatically drive the execution of an app, to e.g., determine which dynamic code is loaded at specific instructions.

3.2 Issue Mitigation

Mitigation of issues can be performed on different levels, e.g, directly in the app, or in the platform. Only few approaches for mitigation of issues in apps have been published.

The approaches by Zhang and Yin [95] and Ascia et al. [96] statically analyse an app for potential data flows, and add code to the app which monitors where potential data flows during execution. If a data flow is detected during execution, the user is warned about this data flow, but the data flow is not prevented.

TaintDroid [83] shows data flows from certain sources to sinks during runtime, but does not integrate the data flow analysis into the app, but into the Android OS. This has the downside that the platform has to be modified to perform such data flow analysis. Additionally, TaintDroid monitors data flows in all apps running on the system, which can result in performance issues.

XManDroid [97] is a security framework to 'detect and prevent application-level privilege escalation attacks at runtime' according to specified policies. The framework modifies the Android Framework to precisely monitor permission usage at runtime. Using this approach, they are able to detect and prevent several attacks, e.g., if two apps work together to perform malicious activities. A similar system to monitor and prevent unwanted permission usage after receiving an IPC message is proposed by Felt et al. [98]. Their system also monitors IPC on OS-level, and prevents apps from executing IPC

messages which would use permissions in other apps, but were not given to the sending app.

RiskMon [99] performs machine learning to assign a risk score to actions performed by an app. Depending on this score and a users 'risk assessment baseline', RiskMon then dynamically allows or revokes specific permissions, once the app performs actions which are not classified as expected behaviour. This effectively prevents execution of unexpected behaviour.

Batyuk et al. [100] propose a system which performs static analysis to detect specific flaws in apps, e.g., data leaks, or inclusion of unwanted libraries. After the analysis, a mitigation strategy is applied which removes the specific issue, or prevents the installation of the app if no mitigation is available. One possible mitigation strategy – which they applied in a prototype – is to detect privacy leaks of the International Mobile Equipment Identity (IMEI), and replace the reading of the IMEI with the unique identifier Universally Unique Identifier (UUID).

3.2.1 Code Loading on Android

In Section 5.5, a process for mitigating insecure code loading on Android is presented. Few work already exists which covers this topic to some extent:

Poeplau et al. [101] give a overview about the different possibilities for loading code on Android. They detail the motivation for loading external code, and discuss several problems of doing so, including insecure downloads, unprotected storage and the improper use of package names. Poeplau et al. try to solve the issues of external code loading by introducing different application verifiers which can be decoupled from official app markets. This verifier analyses libraries and issues signatures if they are benign. The authors system is a modification of Android 4.3, which performs the signature checks of a loaded library.

Hu et al. propose a library integrity verification called Duet [102]. Duet collects original library files from their developers to build a reference database for legitimate libraries. Afterwards, libraries included in apps are checked against this reference database, and malicious library usages can be detected. Although this prevents the malicious libraries to be distributed via an official market, it does not help in the case of other distribution channels (e.g., rogue markets or sideloading).

3.3 Summary

This chapter shows that analysis of Android apps is an already well researched area. In the following sections, several gaps in current approaches will be closed:

Analysis Framework: current frameworks can already use results of the static analysis in a subsequent dynamic analysis. The goal of current frameworks is to detect certain classes of security issues in apps, e.g., data flows, permission usage, or a combination of different security issues. This is different to the proposed analysis

framework in Section 4.2. Our proposed framework is designed as generic framework for analysis, which can be used to detect arbitrary security issues, by allowing arbitrary analysis techniques, and arbitrary dependencies between the used techniques. E.g., the framework can perform a static analysis which triggers a targeted dynamic analysis which triggers another static analysis. As each analysis phase can use the results of all previously run phases, this allows arbitrary combinations. Such flexibility was not in focus of previously published analysis frameworks.

Data Flow Analysis: previous data flow analysis techniques require a time and resource intensive so-called entrypoint analysis of the app which is used as starting point of the data flow analysis. In contrast to this, Apparecium calculates the data flows directly from all sources to the sinks of the app. This allows faster analysis results, and can potentially find more data flows, if these do not occur on the control flow from an entry point of the app. In contrast to the most prominent data flow analysis – FlowDroid – our approach works directly on the Smali representation which is beneficial as Smali 'most accurately preserves and provides the closest reflection of the original program behaviours.' [25] Apparecium is explained in detail in Section 4.4.

Android Activity Flow Reconstruction has already been proposed by different publications. Previous publications focus on a precise reconstruction of such flows, by relying on time and resource intensive data flow analysis. The approach shown in Section 4.5 chooses a different approach of reconstructing transitions in a data flow agnostic manner. Although this leads to a decreased accuracy, it is not as time and resource consuming as a data flow analysis.

Differentiation of Core Code and Library Code: previous work focussed on detecting libraries in unobfuscated apps. Once apps are obfuscated, those approaches cannot detect libraries any more. More recent approaches are able to also cope with obfuscated apps, but differing to the approach shown in Section 4.6, fail to detect libraries if the package structure is modified, or advanced obfuscation based on Java Reflection is used. Our approach is able to not only detect the name of the library, but also the version of the library with a high accuracy, even after obfuscation.

Data Leak Analysis by Injecting Targeted Taint Tracking focusses on dynamic taint analysis, i.e., data flows will be observed as they occur. Previous work, i.e., TaintDroid requires modifications of the Android OS to perform its analysis. As the implementation of TaintDroid is specific to one Android OS version, transferring it to a newer version or different hardware is difficult. Contrary to this, the approach presented in Section 4.7 adds taint analysis directly to the app itself, which allows the analysis to be performed on a unmodified Android OS.

Automated or manual mitigation is a topic not yet widely researched on Android. This thesis introduces two automated mitigations:

Library Loading: Section 5.5 explains library loading on Android, in particular, how loading code from other applications also installed on the device can be done correctly. This is done by adding verification to the app in an automated manner. The

approach is conceptually different to previous approaches which improve library loading either by modifying the Android OS, or by performing security checks in an app market. Adding the verification to the app directly enables independence of any OS version, or the app market itself.

Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps: Section 5.6 shows how data leaks can be prevented in the app by modifying the app itself. Differing to previous approaches, our approach also takes data flows across external tunnels (e.g., files) into account, which has not been considered before. Also differing, the approach works directly on the Smali representation and does not require transformation to e.g., Jimple.

CHAPTER 4

App Security Analysis

Detecting issues in apps is the goal of various analysis techniques. This section will introduce the different classes of analysis techniques and show what each of those classes can achieve, and where downsides exist.

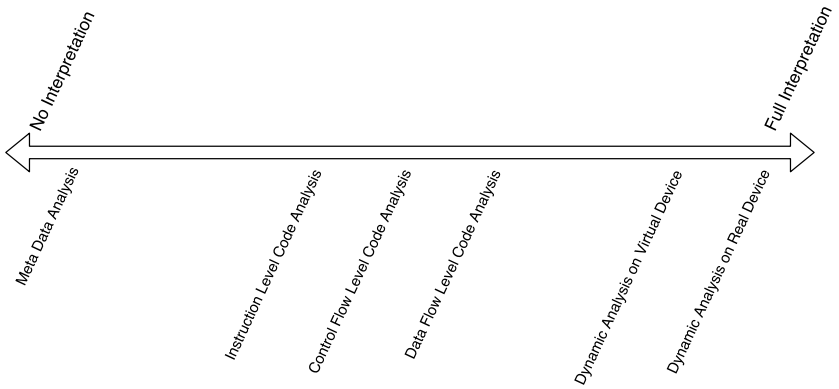


Figure 4.1: App Analysis on Varying Abstraction Level

The different analysis techniques to perform app security analysis techniques can be differentiated by the level of interpretation used by each technique. Whilst meta data analysis only interprets small parts of the app, dynamic analysis executes the app as it would be executed in reality. Figure 4.1 shows how the different analysis techniques can be ordered according to the level of abstraction that is performed during analysis. Each of these analysis techniques are shown in detail in Section 4.3.

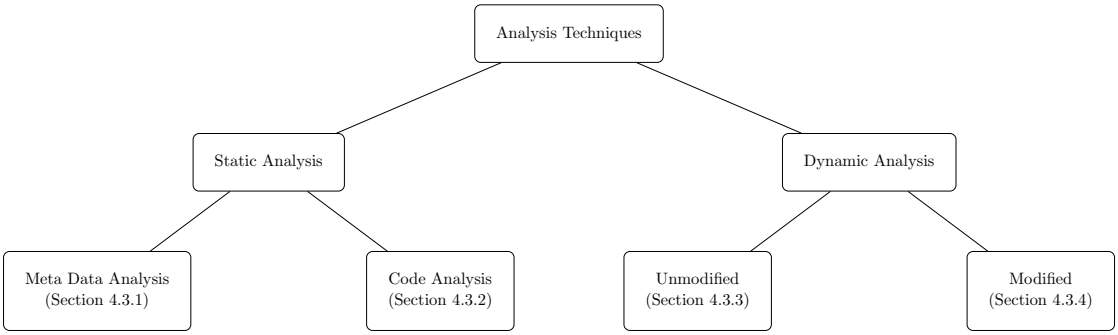


Figure 4.2: Analysis Technique Classes

In general, analysis techniques can be categorized as either being *static* or *dynamic* analysis techniques, and further subdivided into the generic analysis techniques shown in Figure 4.2:

Static analysis techniques are techniques which analyse the app (code or other parts of the app) without actually running any part of the app. Such techniques can further be distinguished into *meta-data analysis* and *code analysis*.

Dynamic analysis techniques always execute the app or parts of the app at some point during the analysis. These runtime analysis techniques can work with either an *unmodified* or a *modified* sample and environment.

For each of these techniques the following topics will be answered in detail in the following sections:

- How is the analysis technique defined and how does it differ from the other techniques,
- what prerequisites are required for this technique,
- which specific analysis techniques are contained in each analysis technique class,
- what are typical problems solved by this technique,
- which shortcomings still exist which are solved by approaches published in the course of this thesis.

4.1 Threats Originating from Apps

Figure 4.3: App Threat Levels

Apps for mobile devices do not only offer additional beneficial functionality to the user, but can also contain functionality which is unexpected or unwanted.

One possible classification used in the context of this thesis is shown in Figure 4.3: benign, suspicious or malicious.

Benign apps only contain the functionality the user of the app expects, and no other functionality at all, neither intended (i.e., malicious), nor unindented (i.e., programming errors) by the developer of the app. E.g., for a calculator-app, a benign app only includes functionality to perform calculations and nothing else.

If the app contains additional functionality, the app can be classified as suspicious or malicious. The extra functionality can be further analysed to determine if it is potentially unwanted behaviour of the app, or if the app performs malicious functionality. A suspicious app would be one which contains extraneous functionality which is unexpected but not necessarily malicious, e.g., contacting a server to retrieve advertisement. Such functionality can be further analysed to determine if anything detrimental functionality can be executed. This is strongly depend on the usage scenario of the app. E.g., a found SQL injection vulnerability in the app might be tolerable if there is no possibility to exploit it. More examples of suspicious behaviour originating in legitimate apps is shown in Section 4.1.1.

An app is classified malicious if it contains functionality which is clearly malicious, i.e., performs actions which are clearly detrimental to the user. Examples for such behaviour are banking trojans or information stealers. More examples of malicious behaviour are shown in Section 4.1.2.

Each analysis technique shown in Section 4.3 can classify the issue it finds as belonging to one of these classes. This information can then be used in the following issue mitigation, to give an indication which issues should be mitigated (e.g., all suspicious functionality).

4.1.1 Legitimate Apps

Threats contained in apps do not only stem from malicious apps, but can also originate in legitimate apps, i.e., without any malicious intent of the developer. Nevertheless, such threats can severely impact the security of the user.

Adding functionality to the app which contains threats can happen due to programming errors, insufficient knowledge of the developer, or simply by including third-party libraries. In the case of third-party libraries, a developer does not necessarily know the whole functionality of the library, as analysis of the library is time consuming and requires detailed knowledge of analysis techniques and possible threats.

The most prevalent threats in legitimate apps are privacy issues. An analysis of the 10.000 most popular apps from the official Google Play Store from December 2013 showed that every twentieth app sends the IMEI to a server at startup of the app [12]. Such a threat can e.g., be contained in advertisement libraries which are included by the developer to earn money. In several cases (e.g., as shown by [44, 103]), such libraries collect information about a user and send it to the advertisement server, resulting in a privacy leak.

Other threats in legitimate apps result from programming errors by the developer. E.g., capability leaks (explained in detail in Section 4.3.2.4) are a typical programming mistake found in Android apps which allow other apps to circumvent Android's permission system. A source for possible programming errors is the OWASP mobile top 10 list [104], which contains the most prevalent and serious programming errors in mobile apps.

If any such threat is found in the app, the app is classified as suspicious. Depending on the use case, such classification can be used to decide how to proceed further with the app, e.g., leading to automated mitigation of the issue.

4.1.2 Mobile Malware

Once apps are developed with malicious intent, they are classified as mobile malware. Trendmicro [105] – a company collecting and analysing Android apps – identified the following categories as being the top malware categories of 2016:

Banking Trojans are apps running on the smartphone which steal authentication token sent via Short Message Service (SMS) to the device, e.g., Mobile Transaction Authentication Number (mTAN). Having access to those tokens allows an attacker to bypass the two-factor authentication. If an attacker also gets access to the login data to a bank account, she can issue fraudulent transactions.

Bots are apps which retrieve further instructions from an attacker to perform certain actions. Bots can be used to send mass emails (spam-Mail), or to perform Denial Of Service (DoS)-attacks on servers.

Downloaders do not contain malicious functionality themselves, but download further code from remote locations and execute them at a later time. This loaded code can include any of the functionality explained in this section.

Exploit apps containing an exploit which uses a vulnerability in the OS, to circumvent security measures of the platform, e.g., to work around the Android's permission system. If such an exploit is successful, the app is able to circumvent all security measures and perform arbitrary functionality.

Information Stealer collect information of the device and the user and send this information to a remote server. This information can include personal information,

e.g., contacts, appointments, and information about the smartphone, including its IMEI, model, and serial number.

Ransomware holds parts of the phone to ransom. Such malware can e.g., lock the whole device and only allow access once a payment has been sent to the attacker. Another ransomware example encrypts files on the phone (e.g., photos, or personal documents), and hands over the encryption key only once a payment is made.

Spyware is similar to information stealer, as personal information is sent from the phone to an attacker. Whilst information stealer retrieve mostly static data from the phone, spyware can be used to directly spy on a person, e.g., by retrieving the current location, or recorded from the microphone, or the camera.

These categories show that a major motivation for mobile malware authors is monetary gain. E.g., *banking trojans* and *ransomware* are used to immediately earn money, whilst malware of the remaining categories can be used to earn money indirectly, e.g., by selling the services of the botnet to senders of spam-mail.

If any analysis technique detects such behaviour, the app is classified as malicious. A following mitigation process can then attempt to remove all such functionality from the app, e.g., before publishing the app in an official app store.

4.2 Analysis Framework

Analysis techniques take an app or parts of an app and additional configuration as input, perform their analysis and produce an analysis result. Different analysis techniques have specific limitations which lead to false positives or false negatives. For example, static analysis techniques can produce false negatives if certain problems never occur in reality, e.g., if they reside in parts of the code which never gets executed. On the other hand, dynamic analysis techniques can depend on the way an app is actually executed, and e.g., produce false negatives if certain problems are not executed during analysis.
It is therefore beneficial to combine different analysis techniques (i.e., static and dynamic analysis) to mitigate problems of one analysis technique.

This section introduces an app analysis framework which can be used to perform fully automated app analysis using different analysis techniques. The goal of the analysis framework is to allow fully automated analysis of any mobile app according to specified requirements. These can be amongst others: security related, privacy related, and programming error related. A requirement specifies what should be checked during analysis. For instance, one possible requirement can be: "The app does not contain any HTTP URLs".

Differing to previous proposed security analysis frameworks, the following framework is not focussed on only detecting one class of security issues, but being able to detect arbitrary issues by allowing arbitrary analysis techniques, and arbitrary combinations. In the framework, arbitrary dependencies between the analysis techniques can be realized, e.g., a static analysis can be performed which triggers a targeted dynamic analysis which triggers another static analysis. As each analysis phase can use the results of all previously run phases, this allows arbitrary combinations. Such flexibility has not been the focus of previously published analysis frameworks.

Key goals of the architecture are:

- provide a basis for app analysis with basic analysis techniques already implemented,
- allow easy extension of the framework to account for newly discovered issues or new analysis techniques, and
- allow arbitrary dependencies between the used analysis techniques.

The design of the framework is also generic in the sense that it is not specific to one platform, as all components are abstract. Therefore, the proposed analysis framework is not specific to Android app analysis.

The Architecture is partly based on the publication *A Configurable and Extensible Security Service Architecture for Smartphones* [13], published in *Proceedings of the Seventh International Symposium on Frontiers in Networking with Applications (FINA 2013)*, main author Dennis Titze, co-authors Philipp Stephanow and Julian Schütte.

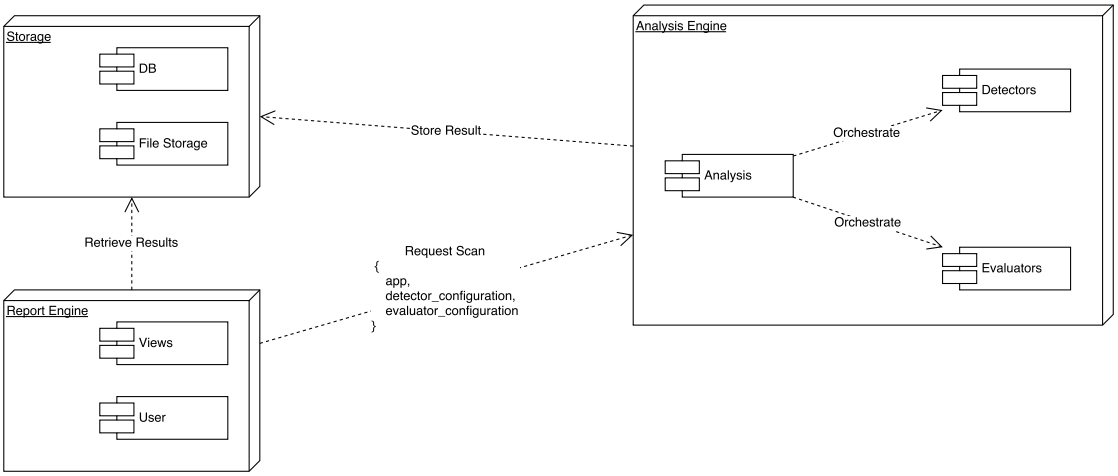


Figure 4.4: High Level Architecture

4.2.1 High Level Architecture

The main components of the framework are its *Report Engine*, *Storage* and the *Analysis Engine* as shown in Figure 4.4.

The *Report Engine* presents the user with different ways to interact with the framework, i.e., via a web-frontend and a REST-API. This component allows a user to retrieve the results from a scan once the analysis is completed. Depending on the needed level of detail of the results, the report engine aggregates results of the app, or returns parts of the raw data to a user.

The *Storage* component provides an interface to persist the results of scans to a database or file storage, depending on the current analysis technique.

The central component is the *Analysis Engine*. It takes care of receiving scan requests from users, scheduling different analysis techniques, aggregating results and storing data via the Storage component. The Analysis Engine contains multiple *Detectors* and *Evaluators*.

A *Detector* is a component that analyses the app using its implemented analysis technique (e.g., string analysis as shown in Section 4.3.2.4) and produces raw data. In this example, the string analysis detector produces a list of strings contained in the app. More examples for analysis techniques which can be implemented by a Detector are given in the following sections. Detectors can have dependencies between each other, i.e., a certain detector can require raw data from another detector. The Analysis Engine takes care of these dependencies and runs the detectors in an order which guarantees that the dependencies are met. To do so, the Analysis Engine performs a topological sort of the Detectors, with Detectors as vertices and the direct dependencies as edges between the vertices (for details on topological sort, see [106]). Cyclic dependencies in the Detectors would lead to an unsolvable execution order, therefore, cyclic dependencies have to be handled manually, e.g., by splitting Detectors into more fine grained Detectors without cyclic dependencies. Which detector, or which set of detectors is required by a specific analysis currently has to be configured once by hand. Future work has to

be done, if it is possible to determine the required detectors in an automated manner from the specific analysis requirements.

After the execution of detectors, the Analysis Engine runs *Evaluators*. These components take generated raw data from detector results, and evaluate if certain criteria are met. Evaluators are directly related to the requirements that should be checked in the analysis. If certain conditions are met, the Evaluator creates one or more *Issue* objects which represent the violation of the requirement and the location of the violation inside the app. These issue objects are stored in the database as well.

An example of the association between requirements, detectors and evaluators is shown in Table 4.1.

Requirement	The app does not contain any HTTP URLs.
Detector	A string analysis produces a list of all strings contained in the app.
Evaluator	The contained strings of the app are searched for any HTTP URLs. If any are found, this is stored for this app.

Table 4.1: Requirement-Detector-Evaluator Association Example

Separating Detectors and Evaluators, allows to easily extend the framework if a new requirement needs to be checked. Section 4.2.5 shows the benefits of such a separation.

4.2.2 Analysis Engine

Figure 4.5 shows a detailed component diagram, and how these can access each other:

For each scan the *Analysis Engine* generates a new *Analysis* object which contains numerous *Detectors* and *Evaluators*, as well as exactly one *AnalysisResult*. Each of the *Detectors* produces one *DetectorResult* which represents the raw data results of the *Detector*. These are stored inside *AnalysisResult*. A specific *Detector* (e.g. *BasicDetector*) implements its own result (*BasicDetectorResult*), which both extend the base classes. This allows generic storage of any *DetectorResult*.

All *Evaluators* are run after the execution of the *Detectors*. Once an *Evaluator* detects that the app violates a requirement associated with the *Evaluator* (e.g., the app communicates via HTTP), it generates a new *Issue* object and stores it inside the *AnalysisResult*. Each *Evaluator* can use raw data of one or more *DetectorResults* which were generated by the analysis technique of the respective *Detector*.

After the analysis, the complete *AnalysisResult* object is stored in the database including all *DetectorResults* and the *Issue* found by the Evaluators. This data is afterwards retrieved by the *ReportEngine* which renders it into different human-readable representations, e.g., as PDF or via a web-frontend.

The design of the framework in this manner allows arbitrary new *Detectors* to be implemented, which all can access all *DetectorResults* which are already available. This allows arbitrary dependencies between the *Detectors*.

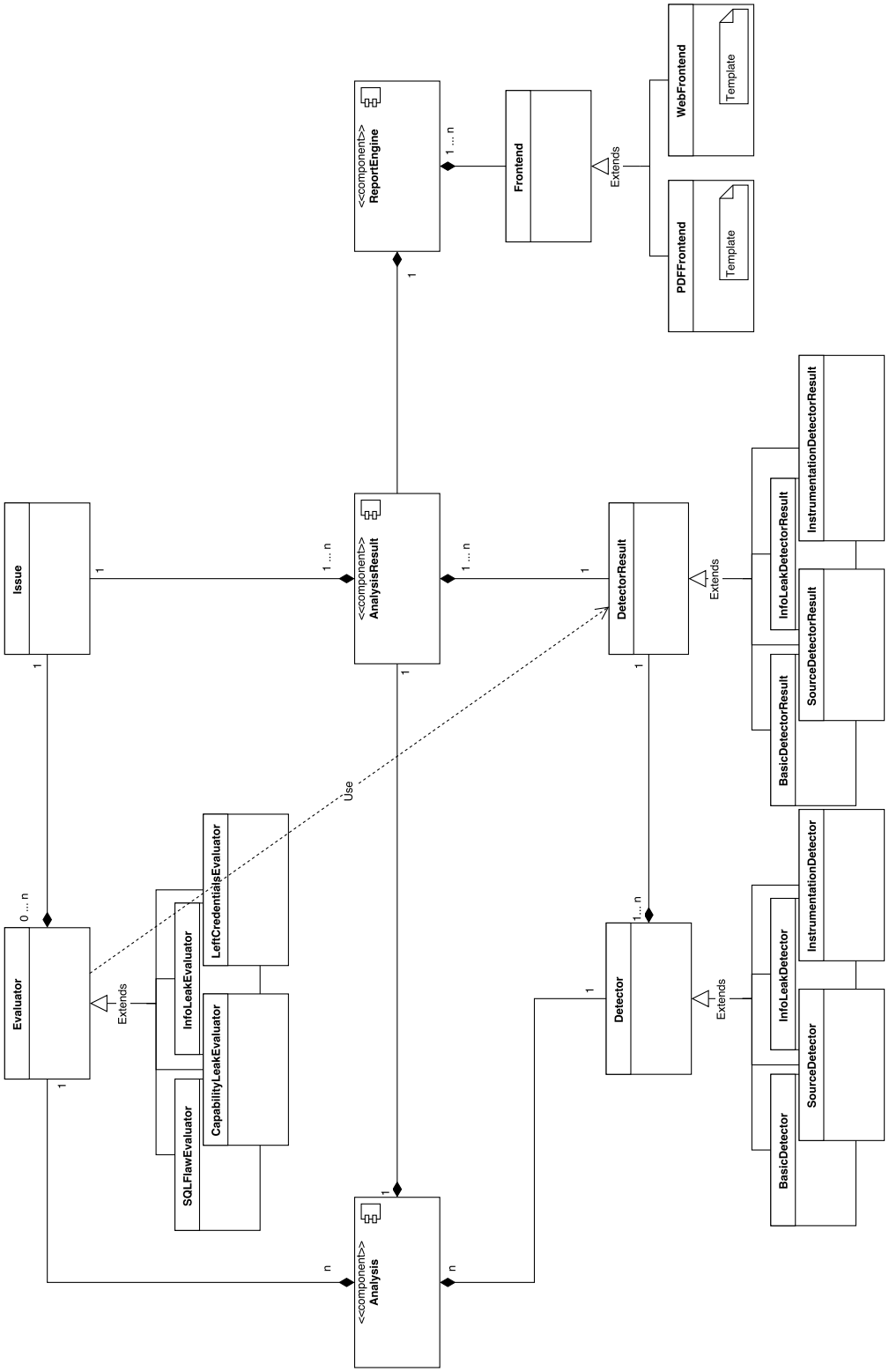


Figure 4.5: Analysis Engine

4.2.3 Analysis Sequences

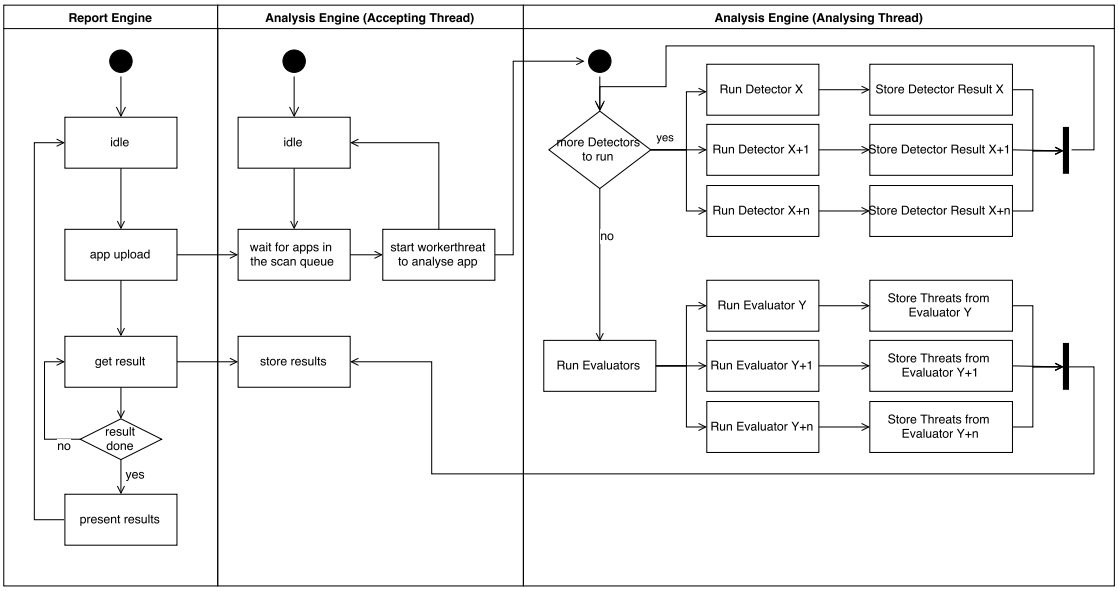


Figure 4.6: Analysis Workflow

Analysing an app can be separated into three different steps:

- Uploading the app,
- applying the Detectors and Evaluators, and
- returning the result.

In the analysis framework, these steps can be separated into three different components shown in Figure 4.6: *Report Engine*, *Analysis Engine - Accepting Thread*, and *Analysis Engine - Analysing Thread*.

The *Analysis Engine - Accepting Thread* waits for any new app to be analysed and starts a new *Analysing Thread*. This Analysing Thread runs every detector as needed and afterwards runs all evaluators. Once all evaluators are finished the result is stored and can be retrieved by the Report Engine.

The *Report Engine* is the component which allows a user to interact with the system to upload an app and retrieve the results.

4.2.4 Evaluators

Each Detector generates raw data about the app. This data is not yet classified in any way, e.g., as malicious, suspicious, or benign.

In a following step, this raw data is available in each Evaluator which can perform further analysis on it to determine if a certain issue exists in the app, e.g., if 'the app communicates unencrypted via HTTP'. A possible implementation for this issues looks through all recorded communication of the app with servers. As this communication

shows if the communication is encrypted (e.g., via standard protocols like TLS), the evaluator can determine if the app contains unencrypted communication.
In this example, the input data to the Evaluator is all recorded communication. The output of the detector is a statement if the app communicates unencrypted or not.

Since the Evaluators are strongly dependant on the Detectors to produce the needed raw data, the evaluators also know which data to retrieve from them. Therefore, they can retrieve the analysis results of one or more detectors which contain this data, and perform their evaluation.

The distinguishing difference between a detector and an evaluator is that the detector performs analysis on the app itself, whereas the evaluator only operates on the generated raw data.

Evaluators are developed according to the requirements of the analysis. A set of evaluators can e.g., be developed to check an app for the OWASP Mobile Top 10 2016 issues [104]. This list contains 'most prevalent and serious mobile issues that Software Engineers should fix' [107].

Separation of Detectors and Evaluators is done to decouple the analysis process from the classification. This allows the classification to be done according to multiple rule sets (e.g. OWASP Mobile Top 10, or company internal rule sets) without the need of performing the analysis itself again.

4.2.5 Extensibility

To allow further extensions on different levels of the architecture, it is built in a modular fashion.

Two different parts of the framework can be easily extended:

- Evaluators for data interpretation, and
- Detectors for raw data generation.

Depending on the goals of extension, these components can be added to the framework:

If a new Issue object should be added to the detection capabilities, the necessary extension depends upon the availability of the necessary raw data to check for this issue. If the raw data is not available, a new analysis technique has to be designed and implemented as a Detector.
After raw data to generate a statement about the issue is available, an evaluator can be implemented to interpret this data. The output of the evaluator will then show if the issue is found in an app or not.

An example of adding a new detection capability to the framework is shown in Table 4.2.

New Issue	'App communicates with servers abroad'
Needed Raw Data	Communication partners of the app
Possible Detectors	Static detection of potential communication partners, dynamic detection of actual communication partners
Evaluator	Check detected communication partners for IP-addresses associated with servers located abroad

Table 4.2: Adding detection of new Issue to the Framework

4.3 Analysis Techniques

This section shows in detail which app analysis techniques are generally possible. These techniques can be categorized as Meta Data Analysis (Section 4.3.1), Code Analysis (Section 4.3.2), Unmodified Dynamic Analysis (Section 4.3.3), and Modified Dynamic Analysis (Section 4.3.4).

These are the basic analysis technique categories, all specific analysis techniques can be categorized into. For each analysis technique category, the following sections go into detail what the technique can achieve, and what prerequisites it has. Additionally, typical problems solved by each analysis technique are shown, and the most prominent analysis techniques are introduced.

Existing analysis techniques still have several shortcomings which are shown in the respective sections. These shortcomings have been improved in the course of this thesis and are presented in the subsequent section.

The improvements namely concern static data flow analysis (Section 4.4), component transition reconstruction (Section 4.5), differentiation of core code and library code (Section 4.6), and dynamic data leak analysis (Section 4.7).

4.3.1 Meta Data Analysis

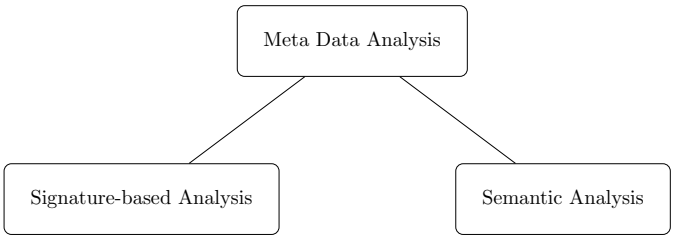


Figure 4.7: Meta Data Analysis Technique Categories

On Android, an app contains different files, e.g., the AndroidManifest, the file containing the bytecode of the app, and arbitrary further files. During *Meta Data Analysis* all data except the code of the app is analysed.

As shown in Figure 4.7, meta data analysis is further separated depending on how the analysed files are handled, i.e., if they are only analysed without interpreting contained data (i.e., *signature-based* analysis techniques), or with interpreting content of the files (i.e., *semantic* analysis).

4.3.1.1 Goal

The goal of this analysis step is to learn if known files are contained in the app using *signature-based analysis* and to extract as much information from the contained files as possible *using semantic analysis*. This includes the app's name, available components, required permissions, and information about the authors of the app.

4.3.1.2 Differences to Other Techniques

What distinguishes this phase from code analysis and dynamic analysis is that the code of the app is not analysed in this step, neither statically not dynamically. That is, the code is neither disassembled nor decompiled, and not executed in any environment.

4.3.1.3 Prerequisite

The prerequisites to perform meta data analysis are the availability of the app itself and the ability to read the files contained in the app. To read the files contained in the app, they have to be decompressed or transformed to a readable format. To understand the semantics of files, the analysed files need to have a clear structure, which is e.g., the case for the contained XML files. If other files contain data in proprietary format are included in the app, the content of these files cannot be analysed.

On Android all files contained in the app are unencrypted, and their content can be parsed. XML files (including the AndroidManifest) are compressed using an Android specific compression algorithm, which is open-source. This algorithm can be used to transform the compressed XML to regular XML format and back.

4.3.1.4 Meta Data Analysis Techniques

The files analysed during meta data analysis are analysed using *signature-based analysis* and *semantic analysis*.

Signature-based analysis calculates a signature of an app (or of a file contained in the app) and compares it against a database of well-known signatures. A crucial step for this analysis is the generation of the database containing the known signatures. To generate such a database, the app (or a file contained in the app) has to be analysed first. Any issue found in the app can then be stored alongside the signature of the app, and can later be retrieved again if a file with this signature is found in an app. Analysis techniques to produce the issues for such files can e.g., be chosen from the techniques shown in the following sections.

Well known hashing algorithms (e.g., sha256 or context triggered piecewise hashing [108]) can be used for signature creation. A general problem for any signature-based detection is the robustness against modified samples. As many parts of the app can be easily modified without changing the semantic of the app (e.g., adding spaces to text files, adding dummy statements to code files, or changing colours in images), the signatures of the files in the app can be easily changed. Such techniques are commonly applied during obfuscation of an app. This is problematic if only the hash of an app is analysed to see if the app is malicious or not [109]. A robust signature-based analysis should therefore take such modifications into account and detect files even after modification.

Many existing analysis techniques do not work reliably if the app is modified, e.g. by obfuscation. This shortcoming is mitigated in the analysis technique shown in detail in Section 4.6. There, a robust hashing algorithm is presented, which still works if the app is obfuscated.

Semantic analysis takes well known files and extracts information from them. This can be strings from known XML files, information from certificate files or information from contained images. On Android, apps have to contain certain files which describe parts of the app. For instance the *AndroidManifest* contains information about the app, from simple information as the app name, to more complex information like required capabilities or accessed functionality. Other files describing parts of the app are so-called layout files which contain information about the composition of screens shown to the user.

4.3.1.5 Typical Problems Solved by Meta Data Analysis

In the context of Android the following analysis techniques are used:

Signature-based analysis is used to detect previously analysed files again. If a previous analysis of an app (or part of an app) produced a result, the signature of the file can be calculated and checked for in new samples. This can e.g., be used to detect known malware in any of the contained files. This approach is similar to traditional antivirus solutions and can e.g., be handled by an offline scanner (e.g., clamav [110]) or an online scanner (e.g., virustotal [93]).

As an example, the SHA1 hashes of all files contained in the app can be calculated (see Listing 4.1). If any of these signatures have previously been associated with a certain issue using a different analysis technique, or by manual analysis, the included file is the same with a high probability (depending on the probability for a collision using the given hashing algorithm).

```
1  08292fd034764f6e74bb52519bbd3576217ffaaa  ./classes.dex
2  db2b529fabec1fbe43f33a987f86d6a8b27d2f94  ./res/layout/skeleton_activity.xml
3  d4483693ad8b14e0e45c6c96d11d177633d6ae77  ./res/layout/framelayout.xml
4  549d69bf17bd59dd237f7a46619faa1fb196ea6e  ./res/drawable/violet.jpg
5  4ce665842cba88aeaf099e84bfbd7d82a300ae05  ./AndroidManifest.xml
6  78b6abe156bb26dbb0c8821933cb60765f42f9f7  ./META-INF/MANIFEST.MF
7  9ede3a821f9c3abf63fb2c3052591b3cfcfe1ba7  ./META-INF/CERT.RSA
8  e54ad7e1a4c7385176d8071961633e86e067ab36  ./META-INF/CERT.SF
9  975cd8cd2ee0eecbde84baf71231738f7bfff895  ./resources.arsc
```

Listing 4.1: Sha1-hashes of App Files

The detection capabilities of signature-based analysis relies on a large number of known malicious sample-signature. Depending on the used hashing algorithm, any modification to a file can change the complete signature and therefore prevent a detection. Hash algorithms exist which are more robust against modified samples, e.g., context triggered piecewise hashes [108], or the approach shown in Section 4.6.

Semantic analysis is used on Android to analyse the following parts of the app:

- The AndroidManifest,
- the certificates the app is signed with, and
- the file types of the remaining files in the app.

The main file analysed during meta data analysis is the AndroidManifest. It contains detailed information about the components of the application and the app's capabilities. It specifies which activities, services, content provider and intent receivers the app possesses. In this manifest, the app must specify which functionality it wants to use, namely which permissions it needs. These permissions are required to perform certain actions, e.g., to call protected APIs, or access restricted parts of the system (see Section 2.1.4).

More information is stored in this file which can be relevant for security analysis:

- Which activity is started when the user clicks the app's icon,
- which interfaces can be used by other apps,
- the package name of the application,
- which permissions are requested by the app,
- which permissions are newly defined by the app,
- which components are secured with these new permissions, and
- which libraries of the OS are used.

On Android, permissions can be defined by an app, which can subsequentially be used to secure its own components, e.g., access to content providers. A simple semantic analysis of the AndroidManifest can therefore check if all components exposed to other apps are

secured by a permission. If this is not the case, this can lead to capability leaks in the app. A capability leak exists, if unprotected interfaces can be called by other apps which internally execute critical functionality (see Section 4.3.2.4).

The certificates are also analysed in a semantic way to retrieve information about the signing entity. For Android apps, signing certificates are generated without any certificate authority. Therefore, the values in these certificates can be arbitrarily chosen by the developer, so all information contained in the certificates is not very reliable. In legitimate apps, the signing entity is very likely the actual developer of the app.

The file types of other files in the app can give crucial information for the following analysis. The app can e.g., contain HTML-files which can be shown at some point using components for parsing and displaying such files. These files can then be used in the further analysis, e.g., to be displayed directly during dynamic analysis.

4.3.1.6 Limitations of Meta Data Analysis on Android

Meta data analysis can only analyse files which are stored in the app itself, and which are not encrypted. If an app e.g., decrypts configuration files during execution, these files are not available during meta data analysis. If such files are encountered, a possible solution is to execute the app, and analyse the file once the app decrypts it during runtime.

4.3.2 Code Analysis

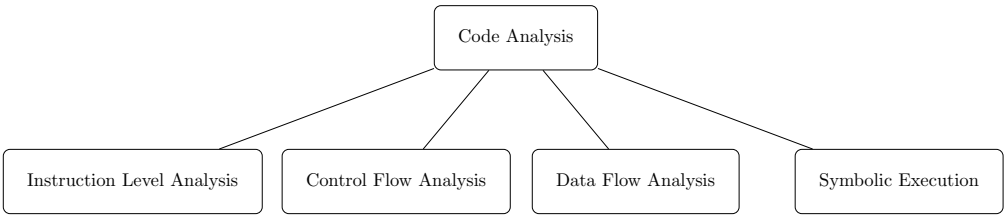


Figure 4.8: Code Analysis Technique Categories

Code Analysis contains analysis techniques which analyse the functionality (i.e., the code) of an app in a static manner without executing code of the app. The analysis techniques can disassemble or decompile the app and subsequentially analyse the resulting code. As shown in Figure 4.8, code analysis can be further subdivided into instruction level analysis, control flow analysis, data flow analysis, and symbolic execution.

4.3.2.1 Goal

The goal of code analysis is to analyse all code included in the app, regardless if this code is executed or not. In particular, this means that code analysis technique can be (depending on the actual technique) agnostic to the execution path, and find issues even if the functionality would only be executed under certain circumstances. Depending on the used analysis techniques, static analysis can produce false positives, i.e., issues which would never be executed during runtime of the app.

4.3.2.2 Prerequisite

Analogous to meta data analysis, code analysis requires the app binary in an unencrypted format. The structure of the app binaries has to be known to disassemble or decompile it. Some analysis techniques may benefit from the availability of the source code, but this is not a strict requirement for code analysis.

On Android, the code of the app is its dex bytecode. This format can either be directly parsed, or transformed to a intermediate representation (c.f., Section 2.2.5).

Further, it is necessary, that all parts of the app which contain functionality are analysed during code analysis. This section is focussed on the dex bytecode on an app. Whilst apps on Android can also contain native code, most scientific publications for Android are only concerned with the bytecode. Whilst there is no theoretical limitation to extend the approaches also to native code, most scientific publications omit this, as the technique itself does not change, but only needs to be implemented for a different code representation.

4.3.2.3 Code Analysis Techniques

Categorizing 30 peer-reviewed publications using static code analysis for Android apps shows that code analysis can be categorized depending on the abstraction level necessary for analysis:

- Instruction level analysis (e.g., [47, 48, 27, 28, 49, 111],
- control flow analysis (e.g., [56, 58, 53, 54, 57, 59, 60, 61]),
- data flow analysis (e.g., [73, 71, 74, 70, 64, 68, 75, 77, 76, 72]), and
- symbolic execution (e.g., [81, 82, 78, 80, 79, 112])

Instruction level analysis first extracts all instructions from the executable. This is a necessary step for all following code analysis techniques. In particular, the executable is transformed from an unstructured format containing only bytes to a structured format containing instructions, operands, strings, etc.

This analysis only gives information about all contained instructions including details for the instructions, but without context of the instruction, i.e., without the information which instruction is called from another location.

Listing 4.3 shows the instruction level representation (based on Android dex bytecode [21]) of the bytecode in Listing 4.2. In this representation, structure and opcodes are easily identifiable and can e.g., be checked for certain values.

```
1  14 01 00 00 12 3A
2  6e 00 00 00 00 01
3  0a 02
```

Listing 4.2: Example Bytecode

```
1  const v1, 4666
2  invoke-virtual method_0001(00, 00, 00, 00)
3  move-result v2
```

Listing 4.3: Transformed Disassembly

Separation of classes and methods is easily possible in dex bytecode. Therefore, Smali IR already separates the code into the containing functions, and separates different classes. This is an essential step for the following control flow analysis. During instruction level analysis, the code inside methods is a list of statements, without any flow information.

During instruction level analysis, the bytecode can also be transformed to Jimple IR (see Section 2.2.5.2) using Dexpler [23]. Whilst Jimple is beneficial for some analysis techniques (e.g., operand analysis), the transformation is more complex than Smali. Therefore Smali can be used if its representation is sufficient, otherwise, Jimple representation can be used.

Control flow analysis generates a graph of an app representing all possible control flows (also called execution paths) that can occur in the app, i.e., which instruction calls which instruction. This graph can be inter- or intraprocedural, i.e., describing which functions are called from which functions, or how the the execution flow can occur inside one function. If the graph describes the interprocedural control flow, it is called *Call Graph (CG)*, if it describes the intraprocedural control flow it is called *Control Flow Graph (CFG)*. A combination of both inter- and intraprocedural call graphs is also possible and is known as super-graph.

To generate both graphs, the instruction level representation (e.g., as seen in Listing 4.3) is further enriched with information. In particular, the unstructured disassembly of each function is separated into so called basic blocks. 'A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)'[113]. Therefore, a basic block is a sequence of instructions which will be executed without interruption, i.e., without branches. Once a branch is reached, this generates two new basic blocks, one for the true-branch, one for the false-branch.

The nodes of the graph are then the functions of the app for interprocedural call graphs and basic blocks for intraprocedural control flow graphs. The edges of the graph represent the calling relationship between the nodes, i.e., they represent that function A calls function B, or basic block A follows basic block B.

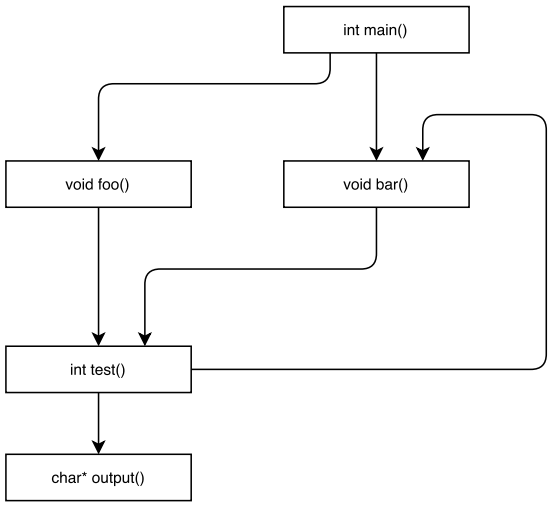


Figure 4.9: Exemplary Interprocedural Call Graph

Figure 4.9 shows an example interprocedural call graph. The graph shows which function each of the function calls. An example for a intraprocedural control flow graph is shown in Figure 4.10. The code shown in Listing 4.4 is split into its basic blocks. Edges connect the basic blocks and show possible call paths inside the function.

More details about the generation of CGs and CFGs can be found in 'Call graph construction in object-oriented languages' [114].

A subset of control flow analysis is component transition analysis. Inside an Android app, many components can interact with one another, e.g. starting an internal service, sending out broadcasts, or starting a different activity (details about Android components are shown in Section 2.2.2). Reconstructing how such components interact with each other, and in which order they are started can help during analysis, e.g., to see under which circumstances a specific screen of the app is shown. Previous approaches to reconstruct component transitions rely on data flow analysis which is very resource intensive. To make this analysis more usable in a real world scenario, the approach shown in Section 4.5 reconstructs the transitions based on the control flow of the app. Whilst this is less precise, it is shown that this approach is still able to reconstruct a considerable amount

```
1  void main() {
2          int a = 3;
3          int b = 10;
4          int c = 0;
5          if (a < b) {
6                  c += 20;
7          } else {
8                  c += 10;
9          }
10         printf(c);
11         a = b;
12         if (c > 10) {
13                 printf(a);
14         }
15         printf(b);
16 }
```

Listing 4.4: Example Code for Intraprocedural Control Flow Graph Generation

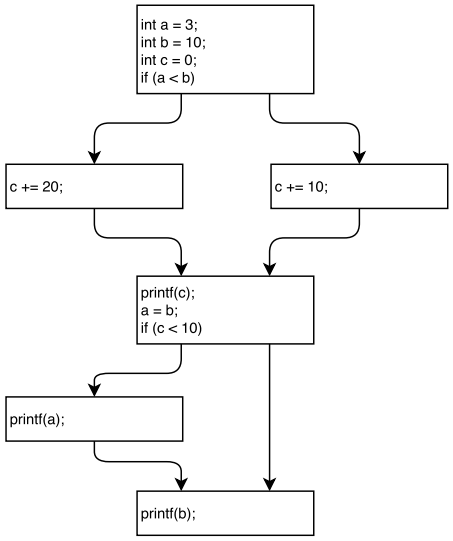


Figure 4.10: Exemplary Intraprocedural Control Flow Graph

of transitions.

Data flow analysis tracks data as it flows through different parts of the code. Similar to call graph analysis, data flow analysis can be performed inside a function (intraprocedural), or across multiple functions (interprocedural). A data flow always starts at a so called *source* and ends at a *sink*. A data flow can e.g., describe how the data 'contacts' flows through the app into the Internet. In this example, the source is the reading of the contact data, and the sink is any function sending the data to the Internet.

As data flows through the app, data flow analysis keeps track of all variables which currently contain data from a source. Such variables are also called *tainted* if only the fact that data flows though the app is tracked. If further information, e.g., which data is flowing is analysed, a so called *TAG* can also be assigned to tainted variables. Each instructions of the app can modify the taint status (the tag) of the variable. In an assignment, the taint status of one variable is simply copied to the assigned variable. If the variable is overwritten by another – untainted variable, the taint status is cleared. To perform a full data flow analysis, it is important that this taint propagation logic is specified and implemented for each possible opcode.

Current approaches require a resource-intensive entry-point analysis to have a single point in the app where the analysis can start. Such a starting point is similar to classic `main`-function in C or Java. On Android, no one function exists where the app starts. Instead, many different callbacks can be called by the framework during different lifecycle stages of the app (see Section 2.2.2 for details). Since such callbacks can also be registered during runtime, the analysis which functions can be an entry point of the app is not trivial. In contrast to current approaches, a more efficient approach is presented in Section 4.4, which calculates data flows directly between sources and sinks.

Symbolic Execution executes code symbolicly, inside a so-called symbolic execution engine. Whilst real execution of the app will assign specific values to variables during

runtime, symbolic execution assigns symbolic values to each variable. Each instruction then works not with specific values, but with these symbolic values. Doing so for the whole program will result in a constraint system, which can be solved to show which values for the symbolic variables result in which control flow path. This can e.g., be used to find possible assignments of variables which lead to a specific statement in the app. Symbolic execution is no dynamic analysis in the terminology of this thesis, as the app is not executed in its intended execution environment (i.e., the Android OS), but only executed symbolicly. Therefore it is classified as code analysis.

4.3.2.4 Typical Problems Solved by Code Analysis

Code analysis is used for different analyses on Android. The following examples illustrate possible analysis techniques building on the different abstractions.

Instruction Level Analysis

Executed functionality can be analysed using the contained instructions in the app. Using this information, it can be analysed if certain calls are included in the app, e.g., an API call to retrieve personal information. As this is only a static analysis, all potentially executed functionality will be returned. Since no information about the control flow of the app is available in instruction level analysis, no statements if this potential functionality will be executed during runtime can be made.

String analysis extracts all strings used in instructions. In Smali-representation, only the instruction to create new strings (`const-string`) uses strings directly. Strings extracted from an app can be used to search for known values, e.g., for URLs, or strings which resemble passwords (e.g., starting with 'AKIA' for Amazon Web Services (AWS)).

Operand analysis can analyse the operands of instructions contained in the app. This can show if a call would be done with an problematic configuration. E.g., if the called instruction is a cryptographic operation, the operands can specify details of the operation and therefore can influence the security of the operation, e.g., by specifying the algorithm. Listing 4.5 shows such an insecure operand in the Jimple syntax: the call to `getInstance("MD5")` returns an implementation calculating an MD5 hash. Often operands of instructions are not directly specified in the instruction, but are specified at some other location in the app and then passed to the instruction as parameter. To reconstruct such parameters, instruction level analysis is insufficient, and data flow based object reconstruction has to be used.

```
1 staticinvoke <java.security.MessageDigest: java.security.MessageDigest
    ↪ getInstance(java.lang.String)>("MD5");
```

Listing 4.5: Insecure Operand (Jimple Syntax)

Call Graph and Control Flow Graph Based Analysis

Reachability analysis can check an app if certain statements or parts of the code can be reached. To do so, the CG and CFG of the app are traversed to check if edges

from any entry point (i.e., a starting point of any of the components of the app) of the app to the investigated statement exist. If such a path through the app does not exist, the statement is considered unreachable.

Capability leaks describe a problem specific to Android apps. As the Android OS secures critical functionality (i.e., API calls) with permissions, an app has to possess those permissions to call the API function (e.g., the app has to possess the permission MODIFY_WIFI_STATE to switch on the WiFi of the device). On the other hand, apps can provide interfaces to other apps to perform certain actions. These interfaces do not have to be protected by any permission.
Figure 4.11 shows an exemplary capability leak: an application A contains a *Capability leak* if it exposes functionality to other apps via reachable interfaces which would require a permission in the Android system. In this example, app A might switch on the WiFi once app B sends a command to the unprotected interface of app A. Capability leak analysis checks if call paths between permission protected APIs and unprotected public interface of the app exist.

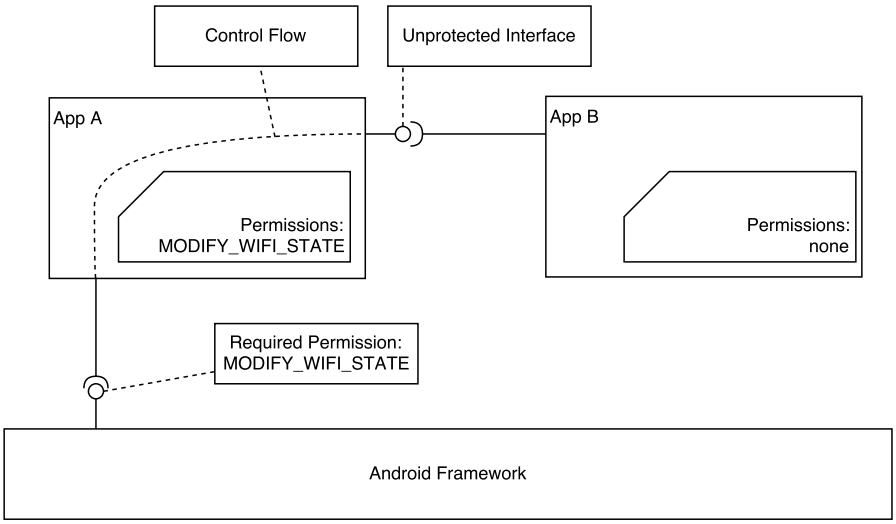


Figure 4.11: Capability Leak via Unprotected Interface

Implementation flaw analysis can check for certain semantics of the code which go beyond simple operand analysis. An example is the verification of certificates in an Android app. If an app implements such a check, code has to exist to verify the certificate and throw an exception if the certificate is invalid. CG/CFG based code analysis can iterate the CG/CFG of such verification and check if a exception is thrown at some point. Although such a analysis cannot easily check if the verification is done correctly, it can identify if no verification is done (i.e., no exception is thrown).

Data flow based analysis

Data leakage checks if a data flow from a source function exists to any sink function. If this is the case, the app contains a potential data leakage. If this leakage is actually executed during runtime is not known during static analysis, as this depends on the actual path of execution taken during runtime. Data leakage analysis is explained in detail in Section 4.4.

Object reconstruction starts at a specific variable in the code of the app and reconstructs possible values of the variable. The analysis performs a backward data flow analysis (see Section 4.4.2.3) starting at the variable to determine all possible values which can reach the variable. This can be used to determine all possible assignment of a String variable to analyse which URLs might be called, or to reconstruct parameters passed to a function. The result of object reconstruction does not necessarily have to be one single possible assignment, as the actual assignment can depend on the execution of the app.

Implementation flaw analysis can also be performed using a data flow based analysis. An example for this is the input to database queries. Such queries should be sufficiently sanitized before being part of a database query. Therefore a data flow based analysis can check if the data reaching the sink 'data base query' is sent to a sanitization function before. If this is not the case, the app contains a potential SQL injection.

Symbolic Execution

Variable assignment: symbolic execution is mainly used to determine which assignment of variables leads to a specific location, e.g., to a critical API, or a source of the data flow analysis. This assignment of variables can then be used to determine if – or under which circumstances – the critical API can be reached.

Input for dynamic analysis: symbolic execution can be used to generate input which is needed during dynamic analysis (see Section 4.3.3.6). Two examples for such approaches are ConDroid [94] and Dynodroid [115].

4.3.2.5 Limitations of Code Analysis on Android

Code analysis on Android OS shows several limitations:

Code Loading

On Android, loading and executing of external code is possible, and many apps do so. As no code is executed during code analysis, the potentially loaded code is not available, and therefore not analysed. Any functionality which only exists in the loaded code can therefore not be found. To mitigate, approaches exist, which reintegrate the loaded code back into the code base of the app (e.g. TamiFlex [116]).

A second solution to solve this problem is to combine the code analysis with a dynamic analysis which actually performs the code loading, and analyse the loaded code afterwards with the techniques shown in this chapter.

False Positives

As the app's functionality is not executed, any finding in the code does not necessarily mean that it is actually executed during runtime. It can be located in dead code, or only be executed under certain circumstances which might not happen during execution. Listing 4.6 shows an example for such unreachable code. Although line 3 contains a call to a malicious function which could be reached in theory, this will not be the case if the function is only called with a $current\_time \geq 1000$. As code analysis does not know the meaning of *current_time*, the analysis will also investigate line 3 and find the call to the malicious function.

```
1  void foo(long current_time) {
2    if (current_time < 1000) {
3      malicious_function();
4    }
5  }
```

Listing 4.6: Unreachable Malicious Code

False positives can be removed by performing an additional dynamic execution which verifies the issue. The challenge for this dynamic execution is to actually execute the path which shows this data flow. This is not trivial, possible ways to target the execution to actually reach specific locations in the code are shown in Section 4.3.3.6.

State Explosion

During data flow analysis, all paths containing tainted variables have to be analysed. Whilst this poses no theoretical limitation, it can be problematic to perform this analysis in practice, e.g., if the amount of memory exceeds the available memory. This happen e.g., if the number of sources and sinks is large. State explosion can be handled to some extent by applying techniques which prune certain paths. Such techniques have to be carefully designed, as pruning too many paths might lead to missed issues.

Call Graph Imprecision

Call graph generation suffers from the major problem of potential context insensitivity which can lead to an incomplete or to an overapproximated call graph.

Context insensitivity is the problem that the context (i.e., the current assignment of variables) might not be known at each location, or might not be used when generating the call graph. If a function call depends on the actual value of a variable which might only be known at runtime, this can also result in an imprecision.

Listing 4.7 shows an example of this problem (taken from [114]): If the call graph is created using context insensitive analysis, only the fact that the method *max* is called in each function will be represented in the call graph. The call graph will therefore have edges from A, B, and C to the function *max*. If the context is also analysed, the call graph generation determines that the variables which call the function *max* are the same for function A and C, but differ from function B. Therefore a context sensitive call graph will have edges from A and C to one *max*-function and from B to another *max*-function.

```
1  procedure A() {
2    return max(4, 7);
3  }
4  procedure B() {
5    return max(4.5, 2.5);
6  }
7  procedure C() {
8    return max(3, 1);
9  }
```

Listing 4.7: Required Context for Call Graph Generation

The opposite of missed edges is overapproximation. Overapproximation can occur e.g., when functions of classes are called and the actual type of the class is not known. Listing 4.8 shows such an example: the type of the class is not known in line 16. But since the possible called functions can only be in class A or class B (determined from the class hierarchy), the call graph generation algorithm can insert an edge to both class functions, resulting in an overapproximation.

```
1  class A implements InterfaceX {
2    void foo() {
3      // do sth.
4    }
5  }
6  class B implements InterfaceX {
7    void foo() {
8      // do sth.
9    }
10 }
11 // ...
12 int bar(InterfaceX x) {
13   x.foo();
14 }
```

Listing 4.8: Overapproximation during Call Graph Generation

Optimizing call graph generation can e.g., be achieved by performing additional dynamic analysis to determine the actual context. Since dynamic analysis is dependent on the actual taken path through the app, this might only result in one possible context which can lead to missed edges in the call graph.

General Problems of Data Flow Analysis

Since the data flow analysis needs to track data statically through an app, it requires a CG/CFG to operate on. Therefore the accuracy of the data flow analysis depends on the accuracy of the CG/CFG. Imprecisions in the CG/CFG can lead to missed data flows (in case of missing edges) or to false positives (in case of overapproximated CG/CFGs).

Another problem during data flow analysis is the exact definition of a propagation logic. For some functions, the taint propagation is not obvious and depends on the actual use case. Functions which alter the data might be considered to remove the taint in one use case, or retain the taint status. This is the case e.g., for a hashing function. If the data is passed to the hashing function, the result is still dependent on the input data and can be considered tainted. On the other hand the result cannot be used to retrieve the

actual input data, so the function can also be considered to remove the taint. Therefore the taint propagation logic must take such cases into account and be designed according to the analysis requirements.

Crucial to the accuracy of data flow analysis is the definition of sources and sinks. Automated methods (e.g., as proposed by Arzt et al. [117]) exist, which determine the sources and sinks from the operating system's source code. If such tools are not available, or not accurate enough for the analysis requirements, the sources and sinks can also be defined by hand.

4.3.3 Unmodified Dynamic Analysis

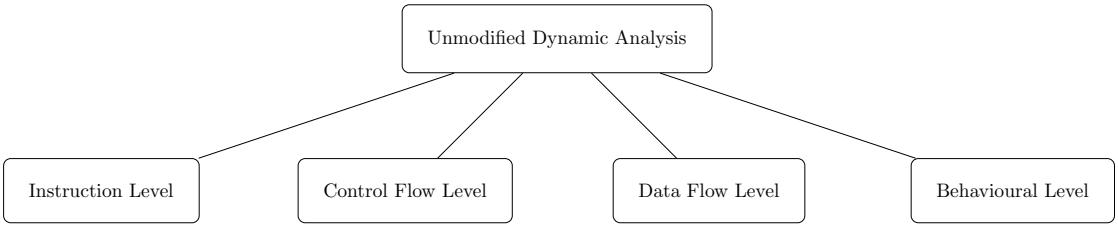


Figure 4.12: Unmodified Dynamic Analysis Technique Categories

During *unmodified dynamic analysis* the original app is executed and its behaviour during execution is observed at different levels. During analysis, the execution environment can either be a real smartphone, or a virtual environment. In this environment, the app is started and – depending on the execution strategy (see Section 4.3.3.6) – interacted with.

Dynamic analysis techniques observes the execution of the app on either instruction level, control flow level, data flow level, or on behavioural level (see Figure 4.12).

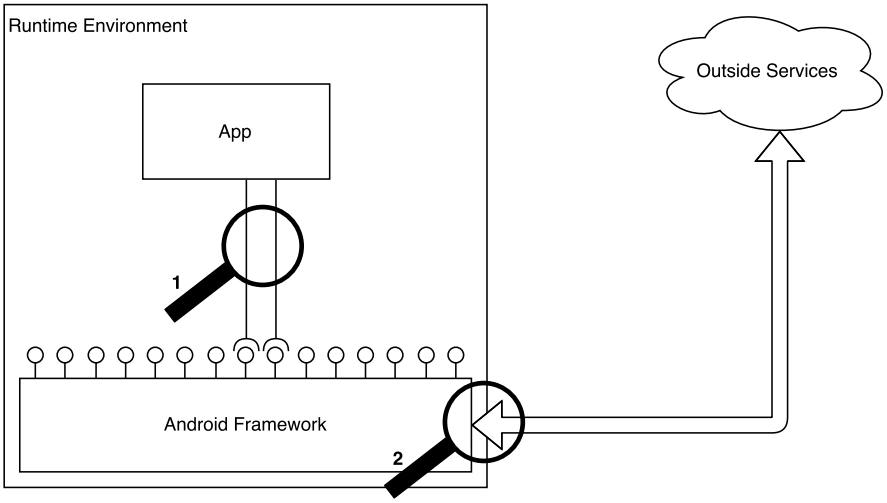


Figure 4.13: Locations of Dynamic Analysis

Figure 4.13 shows the different locations where dynamic analysis is performed: Instruction-, Control Flow-, and Data Flow Level analysis typically investigates the app at the connection between the app and the Android framework (magnifying glass 1). This results in a statement how the app interacts with the system.
The second location where dynamic analysis can be performed is at behavioural level between the runtime environment and any outside service (magnifying glass 2). Such analysis shows how the app running on a system interacts with the outside.

4.3.3.1 Goal

Unmodified dynamic analysis analyses all functionality which is executed by the app during runtime. The goal is to analyse what functionality the app actually executes on a device, and to eliminate false positives which can occur in static analysis. Dynamic analysis can also eliminate false negatives occurring in static analysis. This is the case if the code is not available to the static analysis technique, e.g., if it is loaded dynamically.

As all functionality recorded during dynamic analysis has been executed, dynamic analysis techniques do not produce false positives.

Any analysis which executes an app is dependent on an execution strategy. This strategy executes the app in a specific manner, e.g., to produce a high code coverage. Different strategies are shown in this section. If parts of the code are not executed, this leads to false negatives if issues are contained in the missed parts.

4.3.3.2 Differences to Other Techniques

Unmodified dynamic analysis differs from meta data analysis and static code analysis, as both do not execute any code. Meta data analysis and static code analysis can produce false positives if an issue is in a location which is never executed, if the execution context is not known, or if code is loaded dynamically. This is not the case for unmodified dynamic analysis as the app is actually executed. Every issue found during the execution are true positives.

Differing to modified dynamic analysis, the executed and analysed app is not modified.

4.3.3.3 Execution Environment

To analyse the runtime behaviour of an app, an environment to run the app in has to exist. This can e.g., be real hardware, or a virtual environment.

Real Hardware: If an app is executed on real hardware, the analysis results reflect the actual behaviour of the app. Depending on the used hardware, the analysis can be limited. This can be the case if components are closed source which cannot be monitored or analysed in detail. This is especially important if hardware components should be monitored which do not offer such capabilities. Real hardware also has the downside that it cannot be easily reset after an analysis. Resetting the device to a specific state might require manual formatting of the device, and installing all components needed for analysis.

Virtual Environments can be easily reset, as the environment can simply be thrown away and a new one with a specific state started. As all components of a virtual environment are components written in software, these can be extended with analysis capabilities. This includes changing components which are not accessible on a real device, e.g., adding monitoring directly to a virtual processor. But depending on the actual implementation of the virtual environment, the app can recognize that it is not executed on real hardware, and behave differently. This is especially

problematic, as the app can hide its true behaviour and e.g., only perform benign functionality.

Regardless of whether real hardware or a virtualized environment is used, the environment the app is executed in is referred to as *runtime environment*.

Deciding which environment should be used for the analysis depends on the fact if a virtual environment is available which resembles the actual hardware in a sufficient manner. A sufficient virtual environment is one which is so accurately modelled after the real hardware, that an app can be executed and shows the same behaviour. If such an environment is not available, real hardware has to be used to create accurate results.

As the Android framework is well-defined, and its functionality fully known, a runtime environment executing all functions of the framework can be implemented. Additionally, the Android framework is open-source, which allows modification of the platform itself to allow monitoring during its execution, e.g., to record traffic.

4.3.3.4 Unmodified Dynamic Analysis Techniques

Techniques using unmodified dynamic analysis can be classified similar to those of code analysis, depending on the level of abstraction. The different abstraction levels are:

- Instruction level analysis,
- control flow analysis,
- data flow analysis, and
- behavioural analysis of the execution environment.

Instruction Level Analysis

On this level, all executed instructions are recorded. This recording represents the exact execution sequence of the instructions including their registers. In a following step, the recording can be used to search for specific instructions, e.g., to check which cryptographic operations were performed.
The recording can be done in the execution environment of the app: as soon as a new instruction is handled, this will be written to a log file. If all instructions should be analysed, this will result in a considerable overhead, as for each instruction, several other instructions for generating the log need to be executed. Therefore, this analysis should be limited to only log function calls relevant to the current analysis requirements.

Control Flow Analysis

If the control flow is integrated in this analysis as well, new assertions about the execution can be made. In this analysis not only the single instructions, but additional information, which function called which function is recorded. This can be used to determine which path was taken through the app, and which functions were called along the path.

Data Flow Analysis

Data flows can be analysed during the analysis of an application. In the past, this required a modified runtime environment which stores additional information as data is

read from a source, traverses the app and reaches a sink. Such analysis can then show how data flows through the app, which functions are using certain data, and how it is modified along the execution path.

A modified runtime environment has the downside that an app can potentially detect the modified environment and behave differently. Additionally, modifications to the runtime environment to perform data flow analysis are typically extensive, and cannot be easily transferred from one version of the OS to another version. If apps require a newer version of the environment than the analysis technique is intended for, such analysis cannot be executed. This problem can be solved by the approach shown in Section 4.7 which adds the analysis directly to the app. This analysis technique is a modified dynamic analysis technique.

Behavioural Analysis of the Execution Environment

All previous dynamic analysis techniques observe one specific app in the runtime environment. Additionally, the runtime environment itself can be monitored for changes and for communication with the outside world. Such analysis techniques are called behavioural analysis of the environment. They give information about what is happening on the whole device during the execution on the app, i.e., how the app interacts with its environment.
If multiple apps or services are running on the device, behavioural analysis techniques tend to produce false positives, as the recorded behaviour might not originate in the app under analysis, but from other apps or services. These false positives can be mitigated by including additional analysis techniques, e.g., instruction level analysis.

4.3.3.5 Typical Problems Solved by Unmodified Dynamic Analysis

Instruction Level Analysis

Call Analysis evaluates the instruction recording for specific function calls, or patterns of function calls. As the parameters of called functions are recorded as well, these will further specify how the function is executed. Call analysis can e.g., be used to check if calls to sensitive information (e.g., retrieval of the IMEI, access to contacts), or calls which perform configuration of the device (e.g., switch on the WiFi, or switch on the microphone) are performed.

Implementation Flaws can also be detected during dynamic analysis. The recorded instructions can be evaluated to determine if specific functions are called correctly, e.g., during configuration of encryption. Differing to static analysis which produced potential assignments of variables for implementation flaw analysis, during runtime, the actual parameters can be recorded. These recorded parameters can then be used to check if the correct configuration e.g., for encryption functions is used.

Control Flow Analysis

Implementation Flaws can also be detected during control flow analysis. If e.g., the secure configuration of a function depends on previous calls to other functions, this

can be analysed when recording and analysing the control flow of the app. If the necessary function calls appear in an incorrect order, this indicates an implementation flaw.

Coverage Analysis records which parts of the control flow are executed during runtime. This can indicate if dead code is included in the app which is never executed. As the coverage of dynamic analysis depends upon the execution strategy (see Section 4.3.3.6), unused code might not indicate dead code, but simply indicate that the execution strategy was not able to reach these parts of the app. Therefore coverage analysis gives an indication about how much of the app was executed with the current execution strategy. Coverage analysis can also show which parts of the app are executed if specific actions are performed (e.g., pressing a button in the app), or if external circumstances change (e.g., the location of the device).

Data Flow Analysis

Taint Analysis tracks the taint status of each variable during the execution. A variable is *tainted* at a source and its taint status is propagated along the actual control flow of the app. Once a tainted variable reaches a sink, this flow is recorded and can be evaluated afterwards. For taint analysis during dynamic analysis, the runtime environment can be modified so that the tainted status of variables can be monitored during the app's execution (c.f., TaintDroid [83]).

Behavioural Analysis of the Execution Environment

Inter Component Communication Analysis records all communication of the app with other components on the smartphone. As parts of the framework or of other installed apps can be triggered using ICC, this analysis shows the interaction of the app with the rest of the system. Such interaction can also contain data which should be sent to other components. As all ICC can be intercepted during behavioural analysis, this data can be analysed as well. It can e.g., show what data is sent or received by the app.

External Communication Analysis records all communication with outside parties, including communication to a server in the Internet, or SMS communication with other parties. This can either happen directly inside the analysis environment, i.e., the smartphone or the simulated environment, or from the outside, e.g., the WiFi the device is connected to. Communication analysis gives crucial information about who is contacted, how the communication is performed (i.e., encrypted or unencrypted), and in the case of unencrypted communication, also shows the content sent between the parties.

4.3.3.6 Limitations of Unmodified Dynamic Analysis on Android

Dynamic analysis for Android apps shows several inherent limitations:

Execution Strategy

Since dynamic analysis investigates the actual behaviour of the app, the results depend on the input to the app. Inputs required by the app can be complex, e.g., if the app

requires a user to log in with her personal account as seen in Figure 4.14, or in a game where sequences of touch events have to be performed. Therefore, an automated analysis approach might not be able to produce such input. On the other hand, if the input is generated by a human, no guarantee can be made that the input is deterministic or can be reproduced.

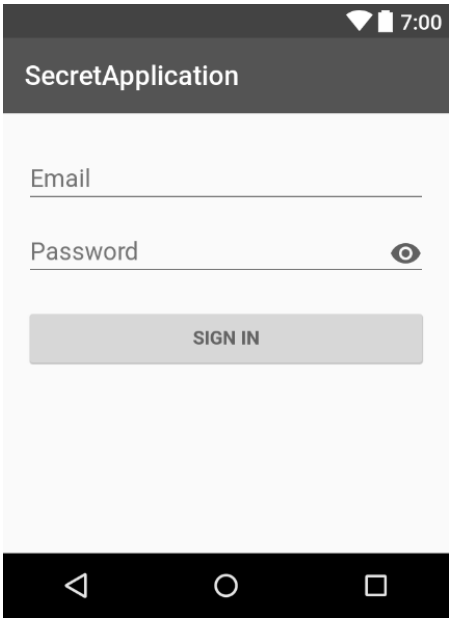


Figure 4.14: Activity Requiring a Username and Password

Several concepts exist to cope with this limitation. Each of the current available approaches can be classified as one of the following:

Random Execution executes functionality of the app by applying random input to the app. This includes clicking at random locations, inserting random input, and randomly changing sensor values. It is easy to see that whilst this is an easy solution, the coverage of this approach is low for a complex app. Zeng et al. showed that coverage of such random execution (as done by Android Monkeyrunner [118]) is as low as 10.3% [119].

Targeted Execution runs the app in a more strategic way. One strategy is to start all entry points of the app, one after the other. Inside the started parts of the app, targeted execution can also click all buttons of the app, or insert predefined text into input fields. Whilst this does not guarantee complete coverage of the app, this will result in a higher coverage than random execution. Several different approaches exist to perform targeted execution, e.g., [26, 119, 120, 121, 122].
One of the targeted execution approaches is 'concolic execution': this approach uses a combination of symbolic execution and app modification, which allows the execution to follow a specific execution path through the app to reach a target statement. This is achieved by iteratively executing the app and recording the symbolic values and their modifications. Once the current execution deviates from the specific target execution path, the execution stops and inverts the last branch

condition which resulted in this deviation. The next execution is then performed with an assignment of variables which follow the targeted execution path. A detailed implementation and evaluation of concolic execution can be found in ConDroid [94].

Replicated Execution records the execution of an app on a real device and replicates this execution during analysis. This includes replication of all user input, as well as replication of sensor values. This replication approach is very similar to manual interaction, as a user has to interact with the app beforehand. Benefit of replicated execution is the reproducibility of the actual analysis, as a subsequent analysis can be performed with the exact same input. The approach is shown in [13, 32].

Manual Execution allows a human to interact with the app during dynamic analysis. Whilst this does not scale to a higher number of apps under analysis, this execution strategy allows to analyse the app with real world behaviour. Manual execution does not guarantee a high coverage of the code, as the user might e.g., only use a small fraction of the functionality of the app. For instance, Azim et al. performed a study on manual interaction with popular Android apps, and showed that the combined coverage of seven users only resulted in a total coverage of 30% of the app screens and only 6.46% method coverage [121].

To perform meaningful dynamic execution of an app, an execution strategy has to be chosen which fits the intended analysis depth. If only the behaviour during startup of services should be analysed, targeted or random execution can suffice. If specific functionality of the app should be performed, automated execution might not be able to execute it. In such a case, manual execution of the app can be required.

Limits of Virtualization

If the app is executed in a virtual environment, or the analysis environment on a real device is modified, the app can be able to determine that it is analysed and behave differently.

The virtual environment therefore has to be modelled very closely to the real device. If certain features of the real hardware are not available in the virtualized environment, the app might behave unexpectedly and the results of the analysis will be meaningless.

This limitation can be mitigated by modifying the app, e.g., by removing calls which fingerprint the environment. This is shown in Section 4.3.4.5.

Dependency on the Outside World

As apps can interact with outside services (e.g., communicate with servers), this can problematic for dynamic analysis for multiple reasons:

- the service might not be available during analysis, which might prevent the app from running,
- if the availability of a service changes from one analysis to the next, the results can differ even if the same app is executed in exactly the same way,
- if the outside service behaves differently (e.g., due to an update), the results can differ even if the app is unchanged.

If an app is tested with a live service, analysing the app dynamically can result in unwanted activity in the service, e.g., if the execution strategy runs the app in a way that is not intended.

To resolve this dependency, an optimal analysis system would access a testing backend of the service, which would always be available during analysis.

False Positives in Behavioural Analysis

If the whole execution environment is analysed to determine the behaviour of an app, this can lead to false positives, if more than one app or service are running on the system. As more services are running inside the OS, the possibility of parallel running services cannot be excluded. E.g., on Android, the framework can check if a new update is available from an official Google server. This results in communication which does not originate from the app under analysis.

Therefore behavioural analysis should be coupled with other analysis techniques to verify if an issue is such a false positive.

4.3.4 Modified Dynamic Analysis

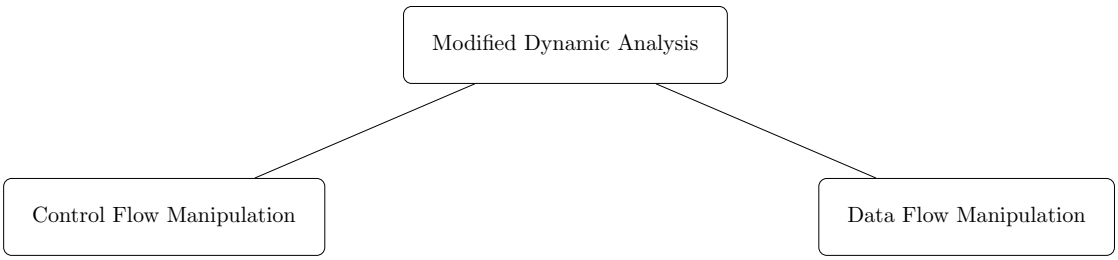


Figure 4.15: Modified Dynamic Analysis Technique Categories

Modified dynamic analysis is similar to unmodified dynamic analysis in that it also executes an app in a runtime environment and monitors its behaviour at different levels. Differing to the unmodified analysis, not the original app but a modified version of the app is analysed. The modifications are applied to allow advanced analysis techniques.

To modify the app, it is disassembled into an intermediate representation (e.g., Smali or Jimple). On this intermediate representation, code transformations are applied to add new code, modify existing statements, or remove existing code. After modification, the app is assembled again and can then be analysed during runtime. The app is modified on one of the levels shown in Figure 4.15: on control flow level or on data flow level.

The actual execution and monitoring is the same as for unmodified dynamic analysis as shown in Section 4.3.4.

4.3.4.1 Goal

Goal of modified dynamic analysis is to analyse all executed functionality, but gain more insight into the app as would be possible without modification. Depending on the modification of the app, this does not necessarily mean that these are true positives of the unmodified app as well, but the issues might have been introduced by the modification itself.

Modified dynamic analysis techniques also incorporate static analysis techniques, e.g., to find relevant function calls in the original app which would then be modified.

As the modified app is finally executed to monitor its behaviour, the analysis techniques shown in this section are classified as dynamic.

4.3.4.2 Differences to Other Techniques

Modified dynamic analysis differs to static analysis, as the app is executed during analysis. Differing to unmodified dynamic analysis, the app is always modified before execution.

4.3.4.3 Prerequisite

As modified dynamic analysis builds upon static and unmodified dynamic analysis techniques, prerequisites for static and unmodified dynamic analysis techniques have to be met to be able to perform modified dynamic analysis. See Section 4.3.2 and Section 4.3.3 for those prerequisites.

Additionally, modified analysis techniques require the ability to modify an app and run the modified app. This requires mechanisms to package the disassembled or decompiled app again. This is the case for Android, as the app is not encrypted, and can be unpackaged, modified and repackaged again.

4.3.4.4 Modified Dynamic Analysis Techniques

Analysis techniques using modified dynamic analysis can modify the app on different levels. These levels are the abstraction levels of code analysis:

- Control flow based, or
- Data flow based

Modification of Control Flow

If the control flow of the application should be changed, this can also be achieved by modifying the app before analysis. Since the app can be changed arbitrarily, the control flow can be adjusted arbitrarily as well. The control flow can be diverted to arbitrary new functions, or certain functionality can be bypassed.

Modification of Data Flow

Similar to the modification of the control flow, data flows can also be changed before analysis. Instructions can be inserted into or removed from the app which change data, thereby altering the original data flow.

4.3.4.5 Typical Problems Solved by Modified Dynamic Analysis

This section shows different analysis techniques using modified dynamic analysis on the different levels.

Modification of Control Flow

Remove analysis-hindering calls: for apps containing calls to check if the app runs in a virtual analysis environment. As this would prevent dynamic analysis in such an environment, these instructions can be removed before running the app. For instance, if the app should be analysed using the default Android emulator, the IMEI of the emulator is always 0. During modification, the calls to retrieve the IMEI can be replaced with an assignment to a constant. This example is shown in the following listings. The call to retrieve and store the IMEI of Listing 4.9 is replaced by storing a constant IMEI in the result variable in Listing 4.10.

```
1 invoke-virtual {v2},
    ↪ Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;
2 move-result-object v1
```

Listing 4.9: Retrieval of IMEI Before Modification

```
1 const-string v1, "123456789012345"
```

Listing 4.10: Retrieval of IMEI After Modification

Call tracking: to perform method level tracking, new instructions can be inserted before each method call to record the name of the to be called method as well as all parameters of the method call. This recording can be stored for retrieval, either in a file, or directly in the Android log. The log will then show all executed instruction in their order of execution. The execution trace can then be used to analyse if the app performed specific functionality during its execution.
Listing 4.11 shows the call trace of an app. In this trace, a database (test.db) is opened[1], followed by a database query to retrieve the column *record* from the table *sessions*. The resulting *SQLiteCursor* is used to retrieve the *record*-Blob which is a byte-array.

```
1 SQLiteDatabase.openDatabase("test.db", null, 805306368, null) ->
    ↪ SQLiteDatabase(SQLiteDatabase: test.db)
2 SQLiteDatabase.query("sessions", [ "record" ] , "recipient_id␣=␣?", [
    ↪ "1234" ] , null, null, null) -> SQLiteCursor@2e1fa22
3 SQLiteCursor.getBlob(0, 0) -> [B(......!..(.Q.....]
```

Listing 4.11: Call Trace of Modified App

Targeted execution: one limitation of dynamic analysis is the need for user input for a majority of apps. This can partly be solved by modifying the app. If a certain function is interesting for analysis, the control flow can be altered to simply jump to the function and continue its execution at the target location. Depending on the functionality of the app, this might result in a crash of the app. If e.g., a login is needed in the app, the control flow can be diverted to jump over the login. As the app expects the login to be performed afterwards, the app can potentially crash, if results of the login are accessed.

Modification of Data Flow

Remove encryption: if the app stores files encrypted in the file system, analysis of the files' content can become difficult. If the data flow of the data to the app is modified so that the data is not transferred to the encryption functions before writing, this can simplify the analysis. To do so, the data flow path resulting in the writing to the file is traversed backwards, and all performed encryption functionality removed. Additionally, all decryption functionality used when reading the file can be removed as well. This allows the execution of the app without breaking the functionality.

[1] the value 805306368 indicates which flags are used for opening the database

Taint analysis: data flows can be analysed in a static manner (see Section 4.4), or in a dynamic manner by modifying the platform (see Section 4.3.3.5). As both solutions have its downside, a third solution is to modify the app in such a way that the data flow analysis is performed directly inside the app during its execution. This approach has the benefit that it is not dependant on the execution environment (i.e., the analysis can by performed on any OS the app originally ran on), and also only shows the data flows which actually occur. This approach has been published in the course of this thesis and is shown in detail in Section 4.7.

4.3.4.6 Limitations of Modified Dynamic Analysis on Android

As modified dynamic analysis executes the app after modification, the benefits and limitations from unmodified dynamic analysis also apply to this analysis. Additionally, more limitations arise as the app is modified for this analysis.

Preserved Semantics

As the app is modified by each of the techniques in this category, the semantics of the app can be changed as well. Even for the most simple modification of adding one statement, the semantic of the app can be changed if the app e.g., behaves differently depending on a checksum of the app. As this checksum will be changed by any modification, the semantics is changed as well.

The change of semantics can result in false positives in the app, i.e., detected issues which are not contained in the original app.

To cope with this problem, for each of the results from such analyses, a further analysis has to determine if the functionality is the same or similar in the original app.

Countermeasures preventing Modification

The app can contain functionality to prevent modification, by means of calculating its own checksum and comparing it against a known value.

Depending on the actual implementation of the app, this check can be removed from the app during the modification step.

Dependency on Static Analysis

As all modifications depend to some extent on results from a static analysis, the limitations of static analysis also exist for modified dynamic analysis, e.g., its imprecisions of generated call graphs. The modification techniques have to take these imprecisions into account to produce meaningful results, e.g., by modifying all possible instructions.

4.4 Data Flow Analysis in Android Apps

This section explains in detail, how our static data flow analysis is performed. The technique named Apparecium analyses an Android app for data flows from configurable sources to sinks, to determine e.g., if an app is able to leak personal information to the Internet. The approach is different to existing approaches, as it does not require a time and resource intensive entry point analysis of the app, but calculates data flows directly between sources and sinks.

> The technique *Apparecium: Revealing Data Flows in Android apps* [11] was published in *Proceedings of the International Conference on Advanced Information Networking and apps (AINA)*, main author Dennis Titze, co-author Julian Schütte.

Previous analysis techniques (including the most prominent one, FlowDroid [65]) require an entry point analysis before the data flow analysis can be performed. FlowDroid requires this analysis as it is based on Soot [22], which starts its analysis at one main function. As there is not one entry point in Android apps, but e.g., multiple for each activity, a dummy main method has to be created which includes calls to all possible entry points of the app. As Android also allows dynamic registration of entry points during runtime, the creation of such a dummy main method is already a complex and time consuming task.

In contrast to this, Apparecium does not require entry points into an app. Apparecium calculates the data flows directly from all sources to the sinks and requires only the bytecode of an app but no further information such as the apps' source code. The result of such an analysis will show possible flows between sources and sinks inside the app, but differing to FlowDroid, not statement is made if these data flows will be called during execution of the app, e.g., as they might reside in dead code.

The taint analysis itself is based upon well known program slicing algorithms [123] adapted to the Android OS.

Apparecium aims at being practically usable on real life applications, at the potential cost of being less precise than other approaches.

4.4.1 Sources and Sinks

An essential step for all data flow analyses is the generation of sources and sinks. Since the data flow analysis checks if flows between these sources and sinks exist, a thorough configuration of these locations is essential.

The straightforward approach – which was chosen for Apparecium – configures sources and sinks as lists of functions. A source is a function call where the data flow starts, e.g., the function call `TelephonyManager.getLine1Number()` which retrieves the phone number of the device. The return value of a source function is automatically marked as tainted. Similarly, a sink is a function which operates on some input, e.g., `Writer.write(...)` which can write data to a file.

The alternative is to define sources and sinks in a context sensitive manner, which results in a more precise analysis. Consider the example in Listing 4.12. Here, data is written to a Socket. If the sink should be the writing of data, `PrintWriter.print(...)` has to be configured as sink, but this function does not state anything about the type of the PrintWriter. This can result in found data flows to arbitrary PrintWriters, not just such from a Socket, and therefore introduces false positives. The context sensitive specification of this sink would therefore be all PrintWriter.print for which the PrintWriter operates on a Socket. Finding these locations in the app requires a data flow analysis to determine the possible context, which will increase the complexity of the whole analysis.

For Apparecium, no context sensitive source and sink definition is used. Whilst this decreases the accuracy of the approach, it also reduces the complexity of the analysis.

```
1  Socket socket = new Socket(IP, PORT);
2  OutputStream outstream = socket.getOutputStream();
3  PrintWriter out = new PrintWriter(outstream);
4  out.print(DATA);
```

Listing 4.12: Writing to a Socket

The second challenge is even more critical: how is this list of sources and sinks generated? Since a manual configuration using all Android APIs is hardly possible (e.g. Android 5.1 contains more than 25.000[2] callable framework APIs), automated tools are needed. Two conceptually different approaches can be used for this task:

- Directly generating source and sink lists from the Android source code. For this task, SuSi [124] was specifically designed to generate such lists, by using heuristics on the API's function names to determine if a function can potentially be a source or a sink. SuSi uses machine learning to determine if a function of the Android framework is a source, a sink, or neither. It uses several features of the function as input to the machine learning algorithm, e.g., the function name: if it starts with 'get', this is an indication that the function can be a sink.
- Source and sinks can be generated indirectly by first generating a permission map, and then deducing a list of sources and sinks. The idea is that functions which retrieve sensitive information and functions which communicate with the outside world (e.g., via Internet or writing to files) are protected by Permissions in the Android system. Starting from these functions, sources and sinks can be identified, by manually specifying which permission corresponds to a source and which to a sink.

Permission maps were intended to provide a mapping between Android APIs and Android permissions, which is not officially available (c.f. [125, 58]). This mapping can also be used as input for sources and sinks: a manual decision has to be made, which permission should be regarded as source or sink. The permission map can then be used to determine the API calls for these permissions, which can then be used as sources and sinks.

[2] reconstructed from the Android framework

For instance, the permission READ_PHONE_STATE can be regarded as source permission, and all APIs which need this permission can then be treated as sources. But generating the list solely using the permission maps will miss several important APIs. In the example from Listing 4.12, according to the permission maps, the instantiation of the socket would require the permission INTERNET. But the actual call to `PrintWriter.print(...)` is not found in the permission maps. Therefore if the permission map solely filled the list of sources and sinks, the leak to the Internet would only be found if the IP or the port used in the Socket instantiation would contain tainted data.

Both approaches only configures sources and sinks as list of functions. If sources and sinks are defined in a context sensitive manner, both approaches have to be altered, to also take the context into account. A source can then e.g., be all function calls where the construction of the context requires the permission READ_PHONE_STATE.

As a compromise between practicability and completeness, these automated approaches can be taken as starting points, but have to be enriched by manual definitions. For Apparecium, the sources and sinks are defined using the permission maps, with additional sources and sinks added, depending on the use case (e.g., adding `Socket.write(...)` when inspecting data flows to the Internet).

Carefully specifying the sources and sinks is an important step in every data flow analysis. Missing sources or sinks will result in missed data flows, whilst too many sources or sink might result in a longer runtime or in too many results which might be harder to interpret.

In this scenario, only information leakage is considered, and the configuration of sources and sinks is tailored to information leakage. A different data flow analysis could perform vulnerability analysis, for which the sources and sinks have to be defined differently. An example for vulnerability data flow analysis is the check if unsanitized input is handed to critical functions (e.g., SQL queries). The sources for this use case would be the reading of input data, and the sinks the critical functions.

4.4.2 Static Taint Analysis

This section describes the algorithms used by Apparecium to perform static taint analysis from the defined sources to the sinks as displayed in Figure 4.16. The slicing algorithm is based upon the algorithm by Weiser [123].

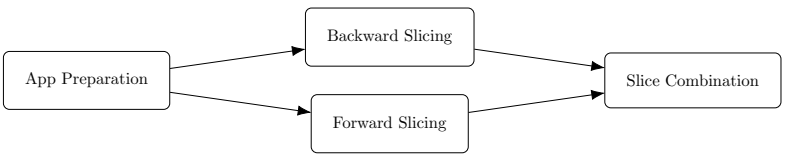


Figure 4.16: Sequence of Required Analysis Steps Performed by Apparecium

The analysis is composed of the following steps:

1. App preparation (including the generation and extension of the call graph)

2. Backward slicing
3. Forward slicing
4. Slice combination

4.4.2.1 App Preparation

In the first step, the app is disassembled using *bakSmali* [20], which transforms the binary bytecode into its textual representation called *Smali*. Parsing Smali code allows the analysis to work directly with an app without the need for source code. Differing from other approaches, Apparecium does not translate Smali to any higher representation to increase efficiency and avoid semantic deviations [25]. Especially for apps which try to hide their behaviour (i.e., malicious apps), translating the assembly to a higher representation is problematic, since an app can be specifically crafted to crash this translation. E.g., an app can easily be developed which runs without errors, but cannot be translated back into semantically correct Java.

As a next step, the basic blocks for each function are generated. 'A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)'[113]. As soon as there is a branching statement in the function, it marks the end of the basic block and the two successors of that statement (the true and the false branch) are the first statements of two further basic blocks. Figure 4.17 shows a simple function split into its basic blocks. The function contains one if-statement which results in the next instructions being split into a new basic block.

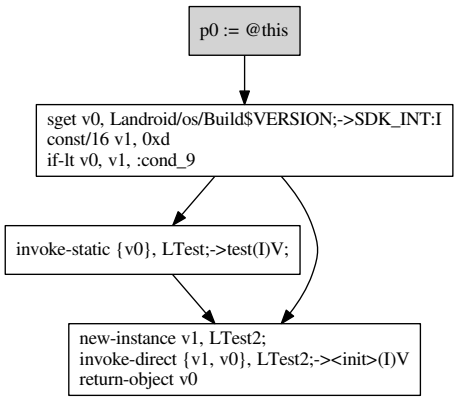


Figure 4.17: Basic Blocks of a Simple Function

In the app preparation step, a class hierarchy is also constructed from the Smali code, which is subsequently needed to determine the callers of functions. The class hierarchy is a tree showing the relationship between classes, i.e., if one class extends or implements another class. This hierarchy can be easily constructed from the smali code, as each class has to specify which classes it extends, and if it implements any interfaces.

This hierarchy is needed in the following analysis to determine the possible callers of a function. If the app contains the class listed in Listing 4.13, and the class hierarchy shows that a class B extends the class A, possible callers of $a()$ are $A.a()$ and also $B.a()$. To not miss any data flows, both callers need to be analysed.

4.4.2.2 Extending the Call Graph

To perform accurate data flow analysis, the call graph has to be as precise as possible, i.e., include all edges of the app, and not contain additional edges, since both backward and forward slicing follow the paths in the call graph extensively. A missing path in the call graph can therefore lead to undetected data flows.

Several edges are not part of a traditional call graph, but can increase the accuracy of the analysis. A call graph with these additional edges is called Data Flow Graph (DFG).

```
1  class A {
2    public static int STATICVAR;
3    public void a() {
4      STATICVAR = source();
5    }
6    public void b() {
7      sink(STATICVAR);
8    }
9  }
```

Listing 4.13: Data Leak through a Static Variable

Consider the example in Listing 4.13. If Apparecium would only look at the default call graph, the backward analysis would start in function `b()`, but since there is no caller of `b()`, the analysis would simply stop. But an app can contain this snippet and functionality which first calls a and then b from outside the analysis context (e.g., from native code). This would result in an undetected data leak. To solve this problem, an edge from all writing location of static (or instance) variables to all reading locations is added to the DFG. If the backward analysis therefore encounters the reading of such a variable, it can determine all writing locations and continue its analysis there.

New paths can be easily added to the DFG in Apparecium. To do so, the corresponding locations have to be linked, e.g., the reading with the writing of files. Callbacks which are used extensively in Android can be added in the same fashion.

Figure 4.18 shows the external call paths currently included in Apparecium. One linked location in Apparecium is `File.write(...)` with `File.read()`.

Adding flows for file read and writes to the DFG leads to the discovery of more data leaks, but it will also increase the false positives rate. For instance, if data is written to a file and read at another location in the app, an actual data flow occurs only if at both location the same file content is accessed, i.e., if it is the same file and it has not been modified in the meanwhile. Tracking these conditions is only possible at runtime, since access to files may happen outside of the analysed app. One optimization would be to heuristically determine the instance of the file handler, which could reduce the number of false positives.

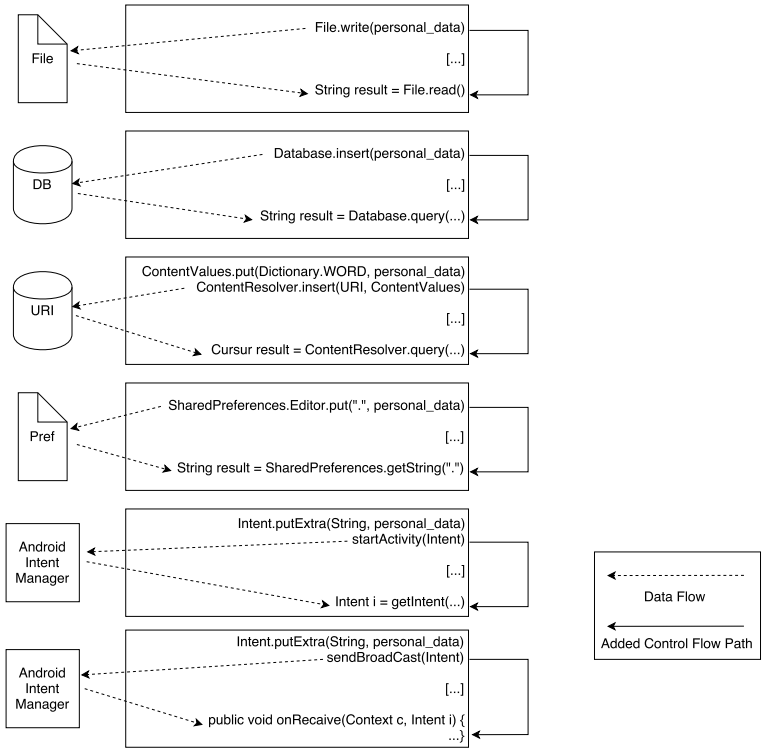


Figure 4.18: Addressed External Tunnels

4.4.2.3 Backward Slicing

After the preparation step, backward slicing analyses the prepared app (i.e., its Smali representation) and generates for each line of code all variables which can reach a sink. That is, if variable `v0` is in this list, there exists a data flow were the data from `v0` reaches a sink.

The essential steps of the algorithm are shown in Figure 4.19. The analysis starts at all configured sinks and adds these locations to a worklist. The algorithm runs as long as there are entries in the worklist. An entry consists of a pointer to the function, a pointer to the location inside the function, and a call stack.

During the backwards analysis, the algorithm may encounter a function call and therefore needs to jump to that function (i.e., adding it to the worklist) to determine if the taint status is propagated or not. In such a case, a call stack records the current function and program counter. This is the location where the analysis will continue once the called function is analysed. Since this function itself can contain new function calls, the call stack can record multiple function locations.

If the function to be examined exists in the current app (i.e., it is part of the app's code base), the analysis simply continues at this function. If the function is out of the scope of the app's code base (e.g., for framework functions), Apparecium overapproximates the taint propagation and assumes that the function propagates the taint status. An alternative approach would be to heuristically try to determine if the taint is propagates

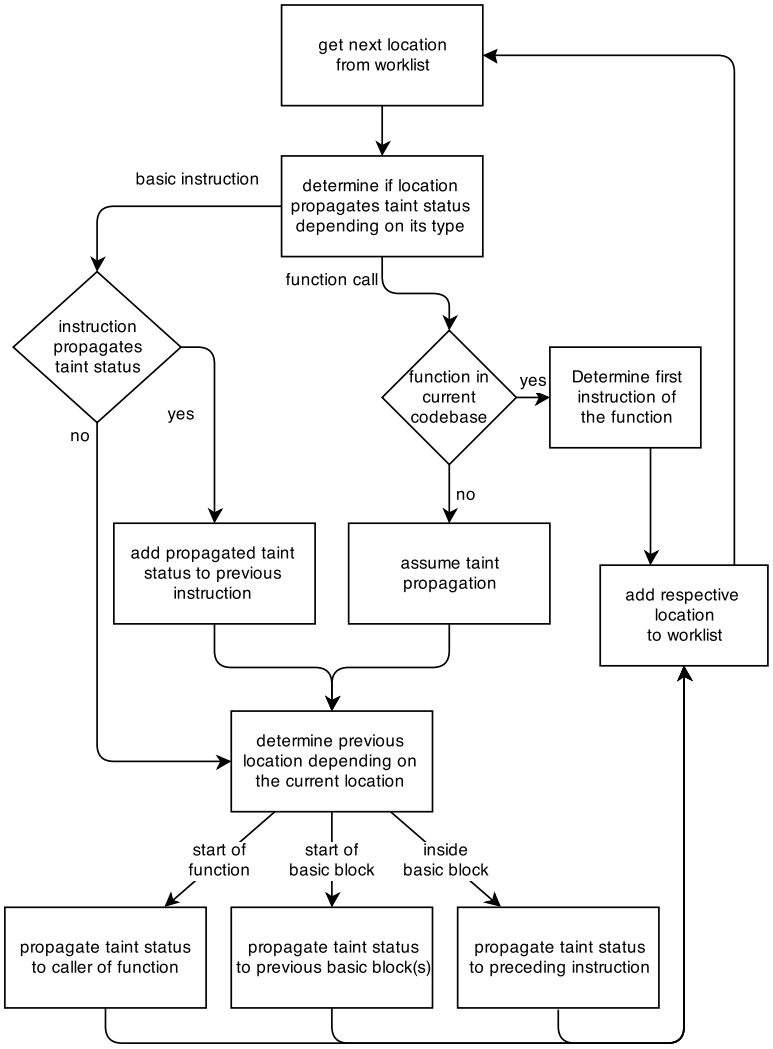


Figure 4.19: Backward Analysis Steps

(similar to SuSi [117]). By overapproximating, Apparecium can produce more false positives, but it is guaranteed that no data flow is missed due to a wrong heuristic.

In each iteration, the backward slicing algorithm takes an element from the worklist and determines if the current instruction propagates the taint status. If at least one of the parameter after this instruction is tainted, it has to be decided if any parameter just before this instruction will also be tainted. Consider the instruction `move v1, v2` which moves the content of variable `v2` into variable `v1`. If variable `v1` is tainted after this instruction, the input variable `v1` will be tainted before this instruction. How the tainting is propagated can be easily configured and has to be done accurately for each instruction of the dex bytecode. An instruction can also untaint a variable, e.g., overwriting it with a constant value (i.e., `const v0, 0`). If the backward analysis encounters such an instruction, it removes the variable from the currently tainted parameters.

If any input variable is tainted (there could be multiple new variables tainted), the previous location needs to be determined. Three possibilities exist for previous locations:

- the previous instruction,
- the previous basic blocks, or
- the caller of the current function.

If the current location is not the beginning of a basic block, the previous location is simply the previous line of code. If the analysis reaches the beginning of a basic block, the predecessors of the basic block need to be determined. If the current basic block is not the first basic block of the function, the analysis continues at the last instruction of predecessors of the current basic block. If the current basic block is the first basic block of the function, no precessing block exists. The analysis will therefore either continue at the top element of the call stack, if available, or at all locations inside the app where the current function is called. Therefore all locations which can call the current function are added to the worklist. Currently direct function calls and function calls using any assignable class are considered. An assignable class is a class which can handle the investigated function call. E.g., if class `A` is derived from class `B`, and both implement the function `a()`, a call to `A.a()` can also end up in `B.a()`. `B` is therefore an assignable class for the call `A.a()`. Considering all assignable classes is an overapproximation, but does not add any false negatives.

In the last step of an iteration, the algorithm checks if the tainted variables in the previous locations already contain the newly tainted variables. If not, these variables are added to the locations and the previous locations are added to the worklist. If the previous location already contains all currently possible tainted variables, it has been already added to the worklist at some point. In such a case, the locations are not added to the worklist again.

The backward slicing will terminate once no element remains in the worklist. Since there exist only a finite amount of lines of code with a finite amount of variables, the algorithm will terminate.

4.4.2.4 Forward Slicing

The forward slicing performs a similar analysis than the backward slicing, by generating a list of variables which can contain data from a source. The algorithm works in the opposite direction of the backward slicing: starting at all configured sources, an analysis is conducted to find all paths originating in one of these sources which propagate the taint status. To do so, the sources are added to a worklist. The worklist contains entries consisting of the current function, the current program counter and the call stack. The call stack serves the same purpose as in the backward analysis: it is used to store the caller of a function to know where to return to at the end of a function.

In each iteration of the forward slicing algorithm, an element is taken from the worklist and it is determined if the taint status is propagated from the input parameter to the output parameter of the instruction. Considering the example `move v1, v2` again, the taint status from `v2` is propagated to the variable `v1`. As in the backward slicing, if the instruction untaints the variable, it is removed from the currently tainted variables.

If at least one variable remains in the currently tainted variables in the current line, the next location has to be determined. The next location can be either:

- the beginning of a function if the current instruction is a function call,
- the next instruction inside the current basic block,
- the next basic blocks, or
- the return address of the function.

If the current instruction is a function call, the algorithm has to determine if the function propagates the taint status of an input variable of the function to an output variable of the function (e.g., for the function `a = sqrt(b)`, `b` would be an input variable, `a` would be an output variable). To do so, the algorithm adds the beginning of the function to the entries in the worklist, with the input variables tainted.

If the next location is an instruction inside the current basic block, the algorithm simply copies the list of currently marked variables to the next instruction, adds the instruction to the worklist and continues.

If the current location is the end of a basic block, the next basic blocks have to be determined. This can e.g., be the two basic blocks following an if statement, or an additional catch block for exception handling. The algorithm again copies the currently tainted variables to the start of these basic blocks and continues.

If the current location is the return statement of the function, and the call stack is not empty, the algorithm propagates the status of the returned variable to the caller. The returned taint status can either be the status of the actual returned variable, or the current instance of the class, if the class itself is tainted (i.e., the this-variable inside the function was tainted). If no entry exists in the call stack, the function was not called by any other function. In such a case, the analysis must continue if the function's return value is tainted. A data flow can occur to all callers of this function, so the algorithm has to take all possible callers of the function into consideration. Analogous to the backward slicing, possible callers are direct function calls and function calls using any assignable class.

The forward slicing algorithm terminates once the worklist is empty. As with the backward slicing, the forward slicing will terminate since only a finite amount of lines of code with a finite amount of variables exist in the app.

The forward analysis could be performed either parallel to the backward analysis, or after the backward analysis is done. In the latter case, the forward analysis can already use some results of the backward analysis for runtime improvements. Namely the forward analysis can disregard any paths which would lead to any functions which were not on a data flow path in the backward analysis.

4.4.2.5 Slice Combination

After generating both the backward and forward slice, these can be combined to show the complete data flow from sources to sinks. The combination simply iterates over all locations and adds a variable to the final result if it is present in both the forward slice and the backward slice.

Reason for the combination of forward and backward analysis is the truncation of data flow paths which are not on the path between a source and sink, but only on one of them. Such a path would e.g., start at a source, but never reach any sink. If only the statement *'Does some data from a source reaches a sink?'* needs to be answered, the forward analysis can already provide this information. But the data flow graph would contain paths and nodes which are not part of any data flow. Therefore we also performed the backward analysis to truncate any unnecessary paths.

Since the backward slice contains a list of variables for each location which can reach the sink, and the forward slice contains the list of variables which can contain data from a source, this final result is a program slice containing all locations with variables which can be part of the data flow. The resulting slice is therefore a subgraph of the DFG along whose edges a data flow can occur.

4.4.3 Visualization

The data flows generated by Apparecium can be visualized for further examination by the user. This is done by transforming the textual output of Apparecium into a D3 [126] graph.

The visualization displays the data flow on different levels of detail. Figure 4.20 shows a data flow on the highest level of detail: each node represents one function through which the data flows from a source on the left side to a sink on the right side. If there are many data flows the visualization on function level will contain too many nodes to provide any benefit. For this reason, the level of detail can be reduced, so that each node only represents a class participating in the data flow. The sources and sinks are still displayed as functions, to be able to see the actual function. Figure 4.21 displays a data flow on this lower detail level.

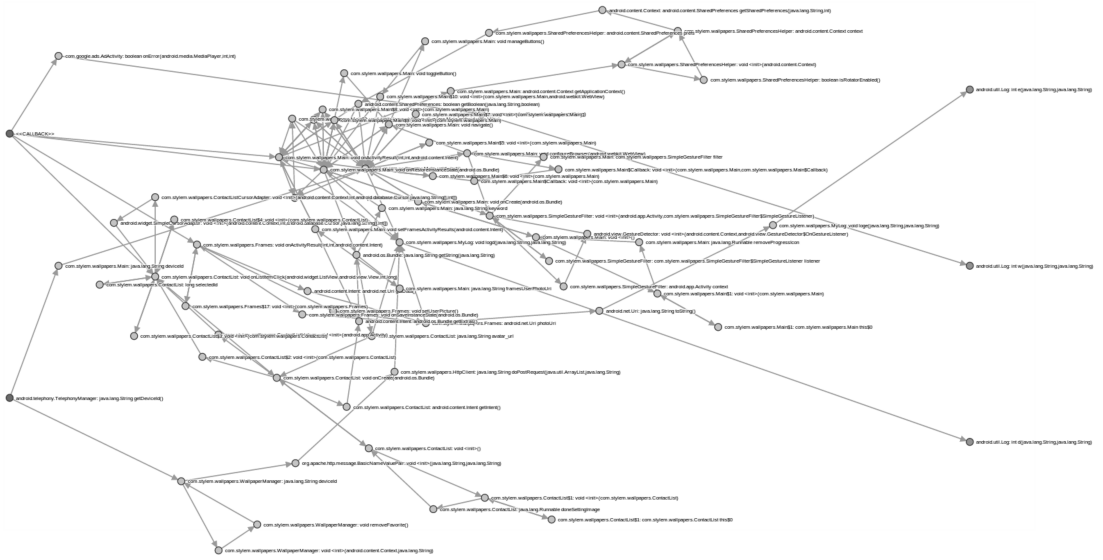


Figure 4.20: Data Flow Visualization on Function Level

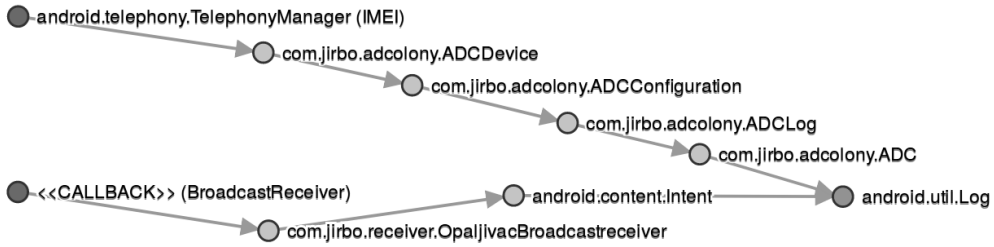


Figure 4.21: Data Flow Visualization on Class Level

4.4.4 Evaluation

While different tools for static taint analysis of Android apps exist, our goal was the implementation of an approach which is highly practical and applicable to real world apps. In our evaluation Apparecium was compared against the publicly available data flow analysis tool FlowDroid [65].

The evaluation provides answers to the following two evaluation questions:

EQ1 What is the duration of an analysis, number of successfully analysed apps and number of found data flows of Apparecium compared to FlowDroid?

EQ2 How do the data flows found by Apparecium compare to the ones found by FlowDroid?

EQ1 To answer *EQ1* we collected 100 random apps among the most popular ones from the Google Play Store and evaluated the effectiveness and efficiency of Apparecium, compared to FlowDroid, which is currently the prevalent and most mature static taint analysis tool for Android. The app selection includes a range of different apps, ranging

from a very small flashlight app, to complex apps like Facebook and WhatsApp. FlowDroid is used in the configuration from Arzt et al. [65], and both tools are executed sequentially on the same PC (Intel Core i7-3520M) with 4GB of RAM assigned to them.

To reflect the usage of both tools in a productive environment such as an app security check or a testing tool, a timeout of one hour was set for both, after which the analysis was interrupted. The lists for sources and sinks were filled with the same inputs, namely the default sources and sinks from FlowDroid.

Basic details of the evaluation are shown in Table 4.3, details about the run times are shown in the box plot in Figure 4.22 and the details about the determined data flows in Figure 4.23.

FlowDroid	
Total analysed apps	100
Average Runtime	815 Seconds
Total Runtime	1359 Minutes
Data Flows[3]	10
Out of Memory Errors	50
Other Errors/Exceptions	30
Timeout after 1 hour	5
Apparecium	
Total analysed apps	100
Average Runtime	1175 Seconds
Total Runtime	1959 Minutes
Data Flows	68
Out of Memory Errors	0
Errors	1
Timeout after 1 hour	17

Table 4.3: Statistics of the Evaluation

In total, Apparecium needed 1959 minutes to complete the analysis compared to 1359 minutes of FlowDroid. But the total runtime cannot be used as a criteria of quality, since more than half of the apps could not be analysed by FlowDroid due to the memory constraint of 4 GB.

Comparing the results shown in Figure 4.23 shows that 10 apps with data leaks were detected by both Apparecium and FlowDroid. A manual comparison of the apps which contained a data flow showed, that all 10 data flows which were found by FlowDroid were also found by Apparecium. Although Apparecium found a data flow for 58 apps

[3] FlowDroid was able to finish some apps even though there were Exceptions. The 10 data flows therefore were not necessarily in apps which did not contain any errors.

which were not found by FlowDroid, this does not necessarily mean that FlowDroid misses these data flows, or we detect 58 false positive. The number of additional data flows comes from the fact that 85 apps could not be analysed by FlowDroid within one hour and 4GB of RAM. So the first test can not be used to compare the quality of these tools. To compare the quality of the found data flows, we randomly selected a subset of these apps and analysed the data flows manually (in EQ2)

Table 4.3 shows that the high accuracy of FlowDroid comes at a cost: half of the 100 apps could not be analysed with only 4 GB of RAM by FlowDroid. This suggests that FlowDroid exceeds practical resource limits when applied to apps of real world size and complexity. In contrast, Apparecium is able to cope with a significantly larger number of apps and identifies more data flows. It must however be considered that the higher number of found data flows is not only due to the higher reliability of Apparecium but also due to a higher false positive rate.

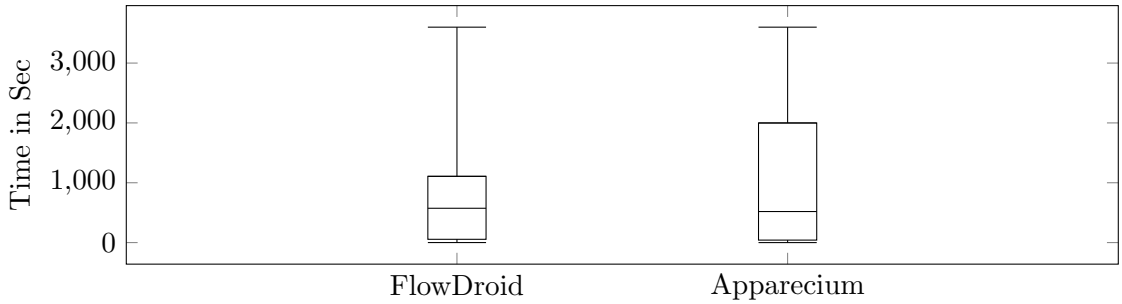


Figure 4.22: Runtimes of Apparecium and FlowDroid[4]

EQ2 To compare the actual data flows of Apparecium and FlowDroid we used a second definition of sources and sinks. Android's phone identifiers were used as sources, and the writing to the Android Log was used as sinks. This allowed us to perform a manual comparison of the found flows. We analysed 10 apps in which FlowDroid found at least one such data flow. The timeout was set to one hour and the maximum memory to 4 GB.

The resulting data flows for these 10 apps were manually compared with the following results:

- For 7 apps the detected data flows in both tools were the same.
- For one app Apparecium detected a data flow which was not found with FlowDroid and which turned out to be a false positive due to Apparecium's overapproximations for external tunnels.
- For two apps Apparecium detected one data flow less than FlowDroid. A manual validation of the data flows showed that these two data flows were false positives of FlowDroid and were correctly omitted in Apparecium.

This shows that in a real world scenario, Apparecium is able to detect data flows with a similar detection rate than FlowDroid. Since both tools overapproximate at different points, the found data flows are not exactly the same.

[4]The comparison of runtimes has only limited significance, since FlowDroid was not able to analyse more than half of the apps due to memory constraints.

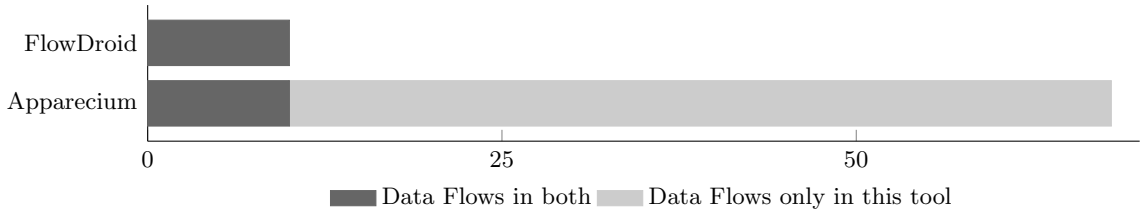


Figure 4.23: Data Flows found in Apparecium and FlowDroid

4.4.5 Discussion

In its current implementation, Apparecium is missing some optimizations, e.g., the addition of more external paths to the DFG as previously described, or a more fine grained specification of external paths to reduce the number of false positives. But these limitations do not impose a conceptual limitation in the proposed static taint analysis.

A further limitation arises from the fact that Apparecium overapproximates at several points, e.g., by adding all function calls of assignable classes, and including external call paths. Although this increases the potential false positives, it also allows us to find more data flows which might be missed otherwise.

Overapproximation also implies that finding a data flow inside an app does not necessarily mean that this data flow will be actually executed. Since Apparecium does not perform its taint analysis using entry points into the app, identified data flows can e.g., take place in dead code or require instantiation of objects which do not occur at runtime. Nevertheless, finding such flows is relevant, since the app can use different techniques to execute such code, e.g., via reflection.

4.4.6 Conclusion

Apparecium is a tool to statically detect data flows in Android apps from arbitrary data sources to sinks. The aim of Apparecium is to provide efficient and practically usable static taint analysis to discover data leaks in Android apps. Unlike previous approaches, Apparecium does not rely on an expensive entry point analysis, and performs its static taint analysis directly from sources to sinks, thereby achieving a high level of efficiency.

By means of an evaluation against 100 of the most popular apps from the Google Play Store we have shown that Apparecium is in fact able to successfully analyse more apps than current static data flow techniques.

In the evaluation, the tests were performed on commodity hardware with one hour timeout, to compare the practical usability of Apparecium versus FlowDroid. This comparison showed, that much more apps can be analysed in this setting with our approach than with FlowDroid. Apparecium was able to analyse 82 apps successfully, whereas FlowDroid was only able to analyse 15 out of 100 apps.
A manual comparison of a subset of these apps showed that Apparecium and FlowDroid detect data flows with similar detection rate.

This section further showed how a static data flow analysis can be performed on Android apps, and discussed general challenges in static taint analysis for Android, such as a highly precise context-based definition of sources and sinks which is currently not possible and motivates further work on the topic.

4.5 Android Activity Flow Reconstruction

Android apps are comprised of one or more components as shown in Section 2.2.2. Once apps contain more components, the transitions between them are not trivially visible in the compiled app, as several different API methods can initiate different transitions, depending on parameters of the method.

Such transitions can be interesting during analysis, e.g., to see under which circumstances a certain screen is shown, or a service started. Previous approaches require a resource-intensive data flow analysis to perform such analysis. In this section, a new component transitions reconstruction approach is proposed, which does not rely on such intensive data flow analysis, but still reconstructs transitions. Whilst this approach is less precise, it is shown that it is still able to reconstruct transitions inside Android apps.

This section shows in detail how component transitions can be performed inside an app, and how they can be reconstructed. This is based on the publication:

> *Ariadnima - Android Component Flow Reconstruction and Visualization* [9] was published in *Proceedings of International Conference on Advanced Information Networking and Applications (AINA), Taiwan, Taipei, 2017* by Dennis Titze, Konrad Weiss, Julian Schütte.

Allowing developers of Android apps to separate their app into different components allows flexible creation of new components, and separation of functionality. Whilst this flexibility is a useful feature for developers, it can complicate the analysis, as the transitions between these components can be needed, but are not easily visible. This graph containing the components of an app with their connections is called Android Component Flow Graph (AFG).

Android apps are comprised of four different components: *activities* are used for the GUI of the app, *services* for longer running background tasks, *content provider* allow access and modification of the app's data, and *broadcast receiver* react to system-wide events.

User interaction with the app is done via activities which typically each represents one screen visible to the user. Reconstructing the flow between activities often shows which tasks can be completed in the app. This can e.g., be used to determine if certain screens are present in the app, e.g., if the user can navigate to a privacy statement, and how this screen can be reached from the start of the app.

Transitions between components on Android are handled via Inter Component Communication (ICC). As other messages can also be transmitted via ICC, reconstructing an AFG is a subset of ICC reconstruction. Differing to other approaches, our approach – called Ariadnima – reconstructs component transitions using a data flow agnostic approach. Whilst this can result in false positives during analysis, the reconstruction does not have to perform a data flow analysis, which can be time and resource consuming for real apps.

Ariadnima only reconstructs the component transitions, i.e., as one component starts another component. The actual transmitted data is not reconstructed in Ariadnima. If this data reconstruction is important, other approaches can be used (e.g., Epicc [64]).

The following sections show, which API methods can initiate component transitions, and how parameters of such methods are reconstructed. The resulting AFG is further visualized, which is shown with an example.

4.5.1 Example Transitions

Figure 4.24 shows two exemplary Activities *MainActivity* and *SecondActivity*. *MainActivity* constructs the Intent *intent* and starts a transition to *Secondactivity* via the API call *startActivity*. In this example, the target of the API call *startActivity*, as well as the source (i.e., the class which calls the containing method) have to be reconstructed. Whilst this is not difficult in this example, the intent could be reconstructed in a more complex method, which makes an analysis more difficult.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        if (someCondition) {
            startNewActivity();
        }
    }

    /** Called when the user clicks the Send button */
    public void startNewActivity() {
        Intent intent = new Intent(this, SecondActivity.class);

        startActivity(intent);
    }
}
```

```
public class SecondActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }
}
```

Figure 4.24: Code Example of Activity Transition

4.5.2 Transition Reconstruction

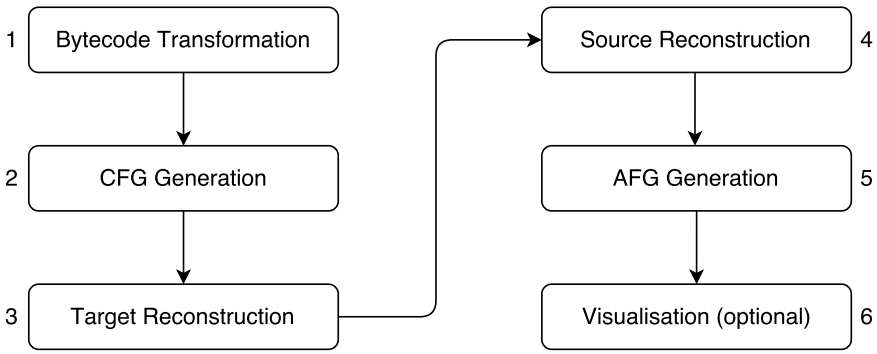


Figure 4.25: The Six Main Phases of the AFG Generation

Ariadnima performs several steps for reconstructing the AFG which are shown in Figure 4.25: first the bytecode of the app is transformed to the intermediate Jimple representation using Soot [22]. Soot also generates an interprocedural Control Flow Graph (CFG) which is needed for the following analysis steps. For each possible transition, the target and the source of the transition need to be reconstructed. This is done in step 3 and 4. Using this information, the AFG can be constructed in step 5. An optional step 6 runs the app in a dynamic environment to create screenshots of all possible activities, which are shown in the visualisation.

4.5.2.1 Transition Target Determination

Several API methods can start a component transition. As no comprehensive list of all API methods which have an effect on the AFG is available, the Android Developer Documentation [127] and the source code of the Android OS [128] have been searched for possible transitions. Table 4.4 shows examples for such API methods resulting in a transition. A more comprehensive list is shown in [129].

Table 4.4: Example Functions Starting an Activity Using Intents

Base class	**Return type**	**Function**
Context	void	startActivity(Intent i, ...)
Context	void	startActivities(Intent[] i, ...)
Activity	void	startActivity(Intent i, ...)
Activity	void	startActivities(Intent[] i, ...)
Activity	void	startActivityForResult(Intent i, ...)

For example, the method *startActivity*(*Intent i*, ...) starts a new activity. The target activity is defined by the Intent i passed to this method. Ariadnima reconstructs this parameter by searching backwards from the method invocation along all possible intra- and interprocedural execution paths. Each location which manipulates the parameter of the method is recorded, and later evaluated.

Transition targets mainly depend on the parameter types *String*, *Class* and *ComponentName* (e.g., inside an Intent). These objects are reconstructed in this backward analysis. Ariadnima uses a data flow agnostic approach for reconstruction. In particular, all possible assignments of the variable are recorded, even if e.g., an instruction would overwrite the assignment as shown in Listing 4.14. Although this results in an overapproximation of possible transition targets, it can be performed faster than a precise data flow analysis. An analysis of the example in Listing 4.14 would return 'Activity1' and 'Activity2' as possible transition targets. This data flow agnostic approach can result in imprecise reconstruction, e.g. if the String describing the activity is constructed differently in different execution paths of the app. As Ariadnima's approach does not handle this precisely (i.e., it does not analyse the data flow). The imprecise reconstructions can then result in reconstructed transitions to targets which do not exist in reality.

```
1  String str = "Activity1";
2         str = "Activity2";
3  startTransitionWithString(str);
```

Listing 4.14: Overapproximated Transition Target

4.5.2.2 Transition Source Determination

The reconstruction of the component target starts in a function containing an API method which initiates the component transition. As this method can be called from different components, or be included in a specific component, the origins of the transition have to be reconstructed.

Section 2.2.2 showed the lifecycle of an activity. All other components on Android have their own similarly complex lifecycle, which contains defined methods called by the Android OS. For instance, the method *onCreate* of a class is called once the activity implemented by this class is started.

To determine all possible callers of an API method, the call graph created by Soot is traversed backwards, and all lifecycle methods are recorded. Each of these methods is the start of a transition.

The result of this traversal is a list of components which can initiate a component transition.

4.5.2.3 Activity flow graph construction

The resulting Android Component Flow Graph can then be constructed with this information, as transition sources and targets are now known.

4.5.2.4 Reconstructing Example Transitions

The following example illustrates the technique used by Ariadnima. Figure 4.26 shows an exemplary app containing two Activities *MainActivity* and *SecondActivity*.

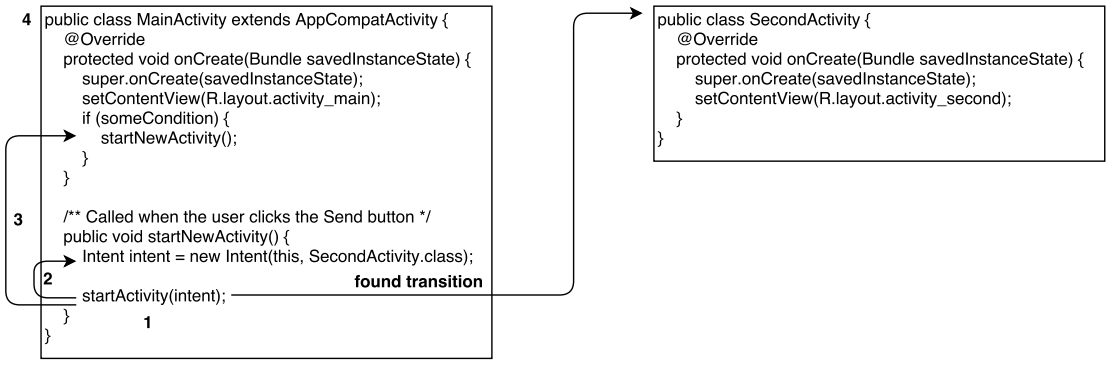


Figure 4.26: Example of Activity Transition Reconstruction

1. First, all method invocations in the code of the app which can initiate a component transition are retrieved. Ariadnima contains a list of API methods which can initiate such a component transition. This list was manually created by checking the

Android documentation for all methods which can start a component transition. In the example, this returns the method *startActivity*.

2. The next step reconstructs the possible values of *intent*, as this parameter specifies the target of the transition, but is unknown in the location of the method invocation. Since the *intent* is only used once in the function, and is constructed in this location, the reconstruction returns the value 'SecondClass.class' as possible transition target.

3. Finally, the source of the transition has to be reconstructed. As *startNewActivity* is no function of the Android lifecycle, the interprocedural control flow graph is traversed backwards. The calling method is *onCreate*, which overwrites the method of the super class *AppCompatActivity*. This method is the starting point of this activity in the Android lifecycle. As no other method calls *onCreate* in this example, only one transition source exists: *MainActivity*.

In this example, Ariadnima finds the transition from the activity *MainActivity* to the activity *SecondActivity* using the API method *startActivity(Intent)*. This is the only possible transition in this example.

4.5.3 Visualization

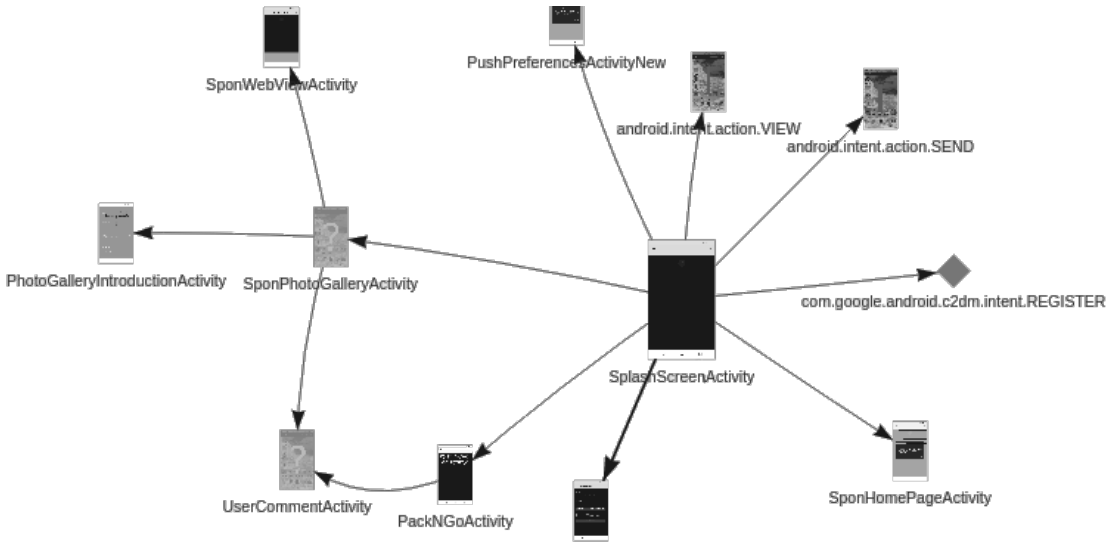


Figure 4.27: Visualization of a Android Component Flow Graph

To allow easy analysis of the AFG for a human analyst, Ariadnima's textual representation can be visualized. The textual representation includes information about the source component, target component, and additional data (e.g., about the parameter values used in the transition). The information about the source and target component allows running the app in an execution environment (i.e., an emulator or a real device), and start the respective component. If the component is an activity, a screenshot is made, which is subsequentially used for the visualisation.

Ariadnima represents this data using the Javascript visualisation library visjs [130]. An example visualisation is shown in Figure 4.27. The example shows 8 activities and their the transactions between each other. Screenshots are shown as nodes of the graph, if the component is an activity. Not all target components have to be contained inside the analysed app. The target of the transition can also be in other apps, or the Android framework. The example shows such a transition from the activity *SplashScreenActivity* which starts a component registered to the Intent 'com.google.android.c2dm.intent.REGISTER'. The receiver of this Intent is not included in the app, and can therefore not be shown in the graph.

4.5.4 Evaluation

To evaluate Ariadnima, 100 apps have been randomly selected from Google Play's 'newest' and 'most-popular' category (in March 2016).

The evaluation was conducted on a virtual machine with 16 GB RAM, and 4 CPUs. Timeout for the code analysis (i.e., Soot's call graph construction and the target and caller reconstruction) was set to two hours. For 5 of the 100 apps, the analysis was interrupted after this timeout, due to a very time consuming call graph generation in Soot. These apps were excluded for the following evaluations.

4.5.4.1 Reconstructed Transitions

Completeness of target reconstruction was measured similar to the measurement used by Epicc [64]: for all locations where an API call was found which can initiate a component transition, the percentage of locations where target information was reconstructed to locations where no target information could be reconstructed was calculated. For 54% of the API methods target information could be reconstructed. Due to the data flow agnostic analysis, target information could not be reconstructed for the remaining cases. For 47% of all API methods which can initiate a transition, both source and target component could be reconstructed using Ariadnima's approach. For the remaining cases, no path from an Android lifecycle method to the transition was detected.

A path can not be reconstructed, if the transition resides in dead code of the app, i.e., no call path from an entry point of the app to the transition exists, if the transition uses input from external sources (e.g., the user or the Internet) to construct the Intent, or if the code of the transition is loaded dynamically, which is not handled by Soot's call graph generation.

4.5.4.2 Runtime

The median time spent in the different phases of Ariadnima is shown in Table 4.5. As the screenshot phase is only optionally used for the visualisation, this phase can be omitted if the resulting AFG is used for other analyses.

Table 4.5: Median Runtimes of the Reconstruction Phases

Phases	**Unpack**	**Screenshot**	**Soot**	**Target**	**Source**	**AFG**	**Total**
***Time** (s)*	8.55	569.1	150.23	0.36	0.34	0.03	732.39
***Relative** (%)*	1.16	77.7	20.51	0.05	0.05	0.004	100

4.5.5 Conclusion

This section showed how the transitions between components of an Android app can be reconstructed using Ariadnima.

An important analysis step of Ariadnima is to search the app for API methods responsible for component transactions. These API methods were manually retrieved from the developer documentation and the source code of the Android OS. Since this API can be changed with an update of the Android OS, a reconstructed AFG is specific to one Android version, and the analysis has to be repeated with the methods extracted from the updated OS.

Ariadnima uses a data flow agnostic approach to reconstruct the arguments of a component transition. Therefore the resulting AFG is an overapproximation. This could be further optimized by e.g., performing a intraprocedural data flow analysis on limited parts of the code.

The evaluation of Ariadnima with real apps showed that although it is able to calculate an AFG in a reasonable amount of time, the resulting graph lacks precision, as only 47% of transitions could be reconstructed. A more precise reconstruction can be reached, if a (partial) data flow analysis is included in the reconstruction, which will increase the resource consumption of the approach.

Depending on the requirements to the analysis, Ariadnima can be used, if precision is less important than runtime. If more precision is needed, this analysis can be enriched by a data flow analysis, to reconstruct the parameters of transitions more precisely.

4.6 Differentiation of Core Code and Library Code

A problem not fully solved yet exists during signature based meta data analysis: even basic obfuscation (e.g., renaming of identifiers) already changes the signature of classes and the app. This has the effect that techniques relying on such signatures are not able to cope with obfuscated apps.

This problem can be solved by more advanced signature generation techniques as shown in this section. In this case, a non-trivial signature is calculated over parts of the app which allows the analysis to detect parts of the app in another app, even if these parts are obfuscated.

In the specific instance, recognition of parts of the app is used for detection of library code in an app, to determine which library is contained. On Android, this is an important analysis, as libraries are merged with the core code of the app, and therefore it is not easily possible to state which libraries are contained in the app. For analysis it can be important to distinguish library code from core code, e.g., to determine if an issue originates in an included library.

This section gives an overview how a obfuscation-resilient signature can be calculated over the code of the app, to distinguish core code from library code. It focusses on libraries in the form of dex bytecode, and is based on the publication:

> *Ordol: Obfuscation Resilient Detection of Libraries* [8] was published in *Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Australia, Sydney, 2017, authored by Dennis Titze, Michael Lux, and Julian Schütte.

Libraries are included in many Android apps to provide functions to the app developer. Such functionality includes displaying of advertisement, libraries for compatibility with older OS versions, advanced UI elements and many more.

As shown in Section 2.2.1, an Android app only contains one file with its bytecode, namely the *classes.dex*. As the developer includes libraries, the code from libraries is merged with the core code of the app during the build process.

If a packaged app should be analysed, no information about which library is used is available in any meta data file. The only clues about included files are given in the classes.dex file. The situation is even more complicated by the use of obfuscators. These tools perform semantic-preserving manipulation of the app's bytecode, e.g., to make reverse engineering of the app's functionality more difficult. Such obfuscators can e.g., inline different methods in other methods, merge classes, and rename identifiers and package names.

If no obfuscation is applied, separating library code from core code can be achieved easily, as libraries are typically contained in known package names. This is shown in Section 4.6.1. But since an obfuscator can easily rename these package names, or even move classes to other packages, the separation is more complicated. Several concepts

exist which allow detection of libraries even after obfuscation. This section will show such a concept called Ordol.

Ordol builds upon concepts from plagiarism detection, i.e., it relies on semantic properties of an app. Such properties cannot be changed completely by obfuscation as this would also change the functionality of the app. In a first step, Ordol builds a database of known libraries by extracting semantic properties of each library. Once the database is created, an app can be checked against this database to determine which library is contained in the app. As the semantic of libraries can change from one version of a library to the next, Ordol is also capable of detecting the version of the library used in the app.

The evaluation in Section 4.6.4 shows that Ordol is, with few exceptions, accurate.

4.6.1 Detecting Libraries by their Package Name

Code for Android apps is written in Java, and compiled to dex bytecode during the build process. Java classes are typically separated into different packages. This separation is kept as the app is compiled to bytecode. Figure 4.28 and Figure 4.29 show a typical package structure of an app in Java and in the resulting bytecode.

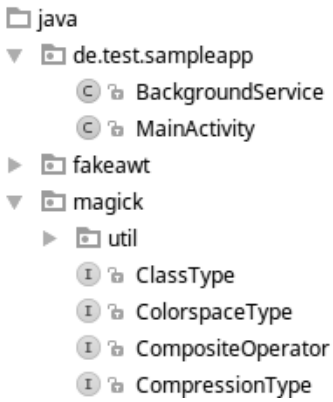


Figure 4.28: Exemplary Package Structure of an App in Java

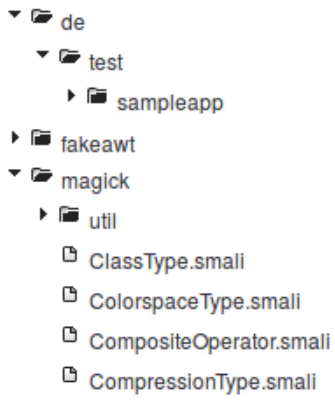


Figure 4.29: Exemplary Package Structure of an App in Smali

The example shows an app with three different packages: *de.test.sampleapp* which is the core code of the app, and *fakeawt* and *magick.util* which are two included libraries. Since *de.test.sampleapp* contains the app's main activity and a service, it is reasonable that this is the package containing the core functionality of the app. Since the app is not obfuscated, the remaining packages indicate included libraries. For instance, *magick.util* refers to the image manipulation library Magick.

Detecting libraries only by their package name is possible in this example, but already shows a problem of this technique: the exact version of the library is unknown. The version is needed if one library version contains a vulnerability and the analysis must determine if this vulnerable version is included in the library. To determine the exact version, the simple approach of comparing package names does not suffice.

The second problem arises once obfuscation is applied. Even simple obfuscators can rename packages, which would e.g., change the package name *magick.util*. Once obfuscation is applied, different approaches become necessary.

4.6.2 Birth Marks

To detect semantically similar code, it is necessary to extract features – so-called Birthmarks (BMs) – from library code. A BM is a "unique characteristic" [131] of an app which can be used to detect libraries in an app. To minimize false positives and false negatives, the chosen BMs should be as unique as possible.

A BM can e.g., be an API call, a specific parameter, or a specific execution order. A resilient BM remains unchanged in the app, even if advanced obfuscation techniques are applied. Additionally, BMs should be different for different classes, to reduce the number of false positives.

The BM used in Ordol should meet the following requirements:

High Resilience: the BM should be resilient against semantic-preserving obfuscations, including control flow manipulation (e.g., method inlining), identifier renaming, and changes to the package structure.

Exhaustive information exploitation: the BM should exploit all information contained in the library's bytecode, including information from very small methods.

Credibility: using the BM should result in a low false positive rate when matching libraries, i.e., no libraries should be detected which are not part of the analysed app.

For Ordol appropriate BM have been designed by including both the instructions and the control flow into the BM. The exact construction is shown in the following Section.

Building on these BMs, Ordol performs a matching of methods and of classes. This matching is done using Maximum Weight Matching (MWM). 'Given a weighted bipartite graph, the MWM problem is to find a set of vertex-disjoint edges with maximum weight.'[132]. The vertices of the bipartite graph represent the methods and classes of the library, and the app which are matched against each other. The weights of the edges of the graph represent the similarity of the BMs of these vertices.

4.6.3 System Design

4.6.3.1 Analysis Steps

Before detecting libraries in an app, the library version database has to be generated. To generate such a database, each library is analysed, and details about the methods and classes of this version are stored. These details include the BMs, the name and the version of the library. This calculation has to be done once for each library version.

After preparation of the library version database, libraries can be detected in an app. The major steps for this detection are shown in Figure 4.30.

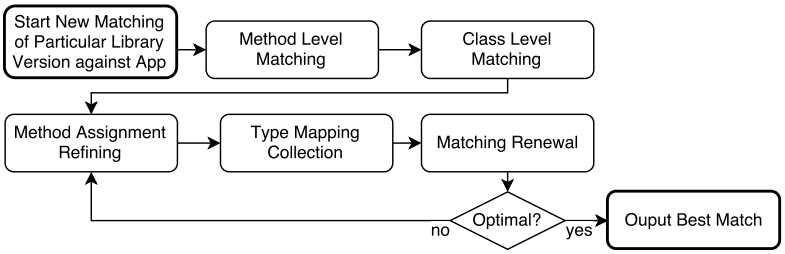


Figure 4.30: Ordol's Library Detection Flow

Before any matching, the app is translated to an appropriate intermediate representation. Working with this representation, a matching of the app against every stored library version is performed on method and class level. After the initial matching, an iterative refinement is applied to improve the detection result.

The result of the analysis is a list of potentially included libraries with the similarity score of the app's classes and the library.

4.6.3.2 Instruction Representation

Ordol builds upon the Jimple representation (see Section 2.2.5.2). In an information-reduction step, the following transformations are applied:

Removal of variable names: local variables can be easily changed in the code, due to reordering of instantiations, or removal. Therefore all variable names are replaced by their type in angle brackets. E.g., $b = a + 1$ is replaced with $< int > = < int > + 1$.

Removal of non-primitive types: as class and function names can be easily changed, all locations in the code where such types are used are changed by replacing the name of the object with '#'. E.g., $foo(int, com.test.Obj)$ is changed to $\#(int, \#)$.

Listing 4.15 and Listing 4.16 show an example before and after the simplification.

```
1  i1 = i2 + 1
2  virtualinvoke $r10.<com.test.Headers$1: void foo(int, com.test.Obj)>(0, $r9)
```

Listing 4.15: Jimple Before Simplification

```
1  <int> = <int> + 1
2  virtualinvoke <#>.<#: void #(int, #)>(0, <#>)
```

Listing 4.16: Jimple After Simplification

4.6.3.3 k-Grams

Since obfuscations can change the order of basic blocks, e.g., by inlining methods or removing dead code, k-grams can be used as suitable BM [133]. K-grams are sequences of instructions of length k. Listing 4.18 shows the k-grams extracted from the pseudocode shown in Listing 4.17 for $k = 3$.

```
1  A B C if (...) { D E F } else { G } H I
```

Listing 4.17: Pseudocode Example

```
1  ABC, DEF
```

Listing 4.18: k-grams for $k = 3$

One limitation of traditional k-grams results from small basic blocks. Traditional k-grams are only formed if the size of the basic block is at least k. As seen in the example in Listing 4.18, much information is not used if the k-gram is constructed in this manner, as all basic blocks of length $< k$ are disregarded.

As the order of k-grams is not used for calculating the similarity, any approach using traditional k-grams as BMs is immune to reordering of basic blocks.

4.6.3.4 Ordol's Flow-Based k-Gram Birthmark

To overcome the limits of traditional k-gram construction, Ordol uses a slightly different BM construction, called 'flow-based k-gram BM', which uses information of the control flow between basic blocks to generate k-grams. This is done by adding all possible successors of a basic block to the k-gram until its length is reached. This approach is similar to the approach by Hyun-il et al. [134].

```
1  ABC, BCD, BCG, CDE, CGH, DEF, EFH, FHI, GHI
```

Listing 4.19: Flow-Based k-grams for $k = 3$

The flow-based k-gram for the example in Listing 4.17 is shown in Listing 4.19. This example shows that all instructions are now included in the k-grams. Due to the usage of the CFG during generation of the k-grams, flow-based k-grams inherently contain information about the possible control flow.

4.6.3.5 Similarities for k-Gram-Based Birthmarks

Different metrics can be used to compare the similarity of k-grams. The most used metric in related work is the Jaccard similarity (e.g., used by Hanna et al. [135]). The Jaccard similarity of two sets A and L are is defined as

$$J(A, L) = \frac{|A \cap L|}{|A \cup L|}$$

Building on this metric, Ordol divides the number of k-grams in both the app A and the library L by the number of k-grams extracted from the library L:

$$QS(A, B) = \frac{|A \cap L|}{|L|}$$

This metric improves robustness against insertion of code [8] and is used as weight for the MWM.

4.6.3.6 Matching on Method Level and Class Level

Ordol uses MWM to calculate the similarity of an app with a library. This matching produces a maximal matching between the app and a library. In particular, the result is an assignment between methods and classes of the app and methods and classes of the library with maximal similarity score. The details of this matching are shown in [8].

4.6.3.7 Iterative Detection Refinement

Using a MWM with similarity-based edge weights produces good library-to-application-class mappings, but in practise several classes will be assigned incorrectly.

Two main reasons are responsible for these incorrect assignments:

1. Parts of app methods can be contained also in the library method, e.g., for commonly used algorithms or programming patterns. As this will result in a high similarity, such methods can be detected as library methods even though these are contained in the app.
2. Obfuscation can remove classes from the app which are not used (i.e., dead code). If a class of the app is similar to a class of the library which is removed, the app class can be matched with the library.

To further improve the result, and remove such incorrect assignments, Ordol uses an iterative heuristic approach. This approach takes the types used in instructions into account. E.g., if two methods operate with the same types in the app, the matched methods must also use the same types. So even if an obfuscation renamed the types, the fact that the same types are used at different locations can be used for assignment refinement. Details about this technique are shown in [8].

4.6.3.8 Termination

Once one round of the iterative refinement ends, the mappings are saved. If these mappings did not change during the refinement, the solution is deemed stable and the refinement ends. Practical experiments showed that a stable solution is typically found in less than 10 iterations.

4.6.4 Evaluation

To show the accuracy of Ordol, its detection results are compared against the results of the publicly available tool LibRadar [51].

To create a library database, 40 distinct libraries and 1112 unique versions have been downloaded manually and included in Ordol. The libraries were selected from the online available libraries which were included in the top libraries of LibRadar [51] and Appbrain [136].

1000 Android apps have been randomly chosen from Google Play's most-popular app category in July 2016 and were chosen as test apps. To generate the results of LibRadar, their web interface was used. Since not all of Ordol's libraries can be detected by LibRadar, only those distinct libraries were chosen for the evaluation which could be detected by both approaches.

4.6.4.1 Resilience of Ordol Results

Table 4.6 shows the libraries which were detected by LibRadar in the 1000 apps. The confirmation rate indicates how many detections were confirmed by Ordol. A confirmation rate < 100% indicates a false negative of Ordol, i.e., Ordol was not able to detect the library.

With few exceptions, Ordol provides a high detection rate. Possible explanations for the lower detection rates are:

Google Mobile Services: GMS versions older than version 6.5.87 were not evaluated in Ordol, as these were bundled differently and could not be used directly.

Parse.com: only four recent version of Parse.com were available. The lack of older versions means that older versions could not be detected.

Inmobi: only four versions of Inmobi were available online. If other versions were included in any app, these are not detected.

4.6.4.2 Correctness of Ordol Results

The correctness of Ordol was evaluated by manual verification. If a library is detected by Ordol but not by LibRadar, this can be a false positive of Ordol, or a false negative of LibRadar.

Table 4.7 shows the results of this evaluation: the concordance rate represents the percentage of libraries detected by LibRaday, which were correctly detected by Ordol. The error rate represents the percentage of false positives in Ordol, i.e., wrongly detected libraries.

The low concordance rate for several libraries shows that Ordol is able to detect significantly more libraries inside an app correctly. The low error rate shows that Ordol has a low false positive rate for most libraries.

The manual verification of mappings with higher error rates revealed the following problems:

Table 4.6: LibRadar Detections Confirmed by Ordol

Library	**Confirmation Rate**
Android Support v4	100,00%
Bolts Base Library	100,00%
Bump pay	100,00%
ChartBoost	100,00%
Dagger	100,00%
Facebook	100,00%
Flurry	100,00%
Jsoup	100,00%
OkHttp	100,00%
OkHttp okio Framework	100,00%
retrofit RESTful Library	100,00%
Nine Old Androids	99,21%
Apache Http	98,75%
Fabric	96,88%
ActionBarSherlock	96,72%
Google Gson	96,39%
Google Ads	95,61%
ZXing ('Zebra Crossing')	94,74%
Google Mobile Services	85,28%
Inmobi	76,47%
Parse.com	59,38%

- Older versions of *ActionBarSherlock* contain an Android Support Library v4. This results in cases where the Android Support Library v4 is wrongly detected as ActionBarSherlock.
- Parts of the *okhttp* library contain caching-related instruction sequences, which also occur in other libraries.
- The *ZXing* library contains specific instruction sequences for creation and handling of barcodes which are also used by other libraries.
- Both *Google Gson* and *Apache Http* contain instruction sequences that are often confused with similar libraries for JSON and HTML processing.
- Parts of the *Bolts Base Library* are included in the *Parse.com* library. As only few versions of the *Parse.com* library were available for creation of Ordol's database, apps containing missing versions can be detected wrongly as *Bolts Base Library*.

4.6.4.3 Resilience Against Obfuscation

To show the resilience of Ordol's approach against obfuscation, the Open-Source mail app 'K-9 Mail' was chosen for evaluation. The code of the whole app is available, which allows us to compile the app locally with different obfuscation techniques. As all included

Table 4.7: Ordol Detections Confirmed by LibRadar

Library	**Concordance Rate**	**Error Rate**
ActionBarSherlock	98,33%	14,29%
Nine Old Androids	76,69%	0,00%
Apache Http	75,96%	4,59%
Google Ads	67,59%	0,00%
Dagger	65,22%	0,00%
Android Support v4	64,68%	0,34%
Google Gson	64,52%	3,88%
ZXing ('Zebra Crossing')	64,29%	6,67%
Bump pay	61,18%	0,00%
Google Mobile Services	56,69%	0,00%
ChartBoost	50,94%	0,00%
Parse.com	48,72%	0,00%
Fabric	47,94%	0,00%
OkHttp	44,90%	12,50%
Jsoup	42,86%	0,00%
retrofit RESTful Library	39,06%	0,00%
Inmobi	31,71%	0,00%
Facebook	22,63%	0,00%
Flurry	18,00%	0,00%
Bolts Base Library	14,29%	6,67%
OkHttp okio Framework	2,04%	1,34%

libraries in their exact versions are known, this enables us to show Ordol's library version detection capability.

ProGuard was used to obfuscate the app. ProGuard is available to all Android developers, as it is included in the default Integrated Development Environment (IDE), and can be easily enabled. For this evaluation, ProGuard was enabled with maximum settings. These include code merging, inlining, code removal, identifier renaming and package name renaming.

Table 4.8 and Table 4.9 show the detected libraries before and after the obfuscation.

The results show that Ordol is capable of detecting libraries even after obfuscation. 3/5 library versions were detected correctly after obfuscation. For the remaining two libraries, the version was not detected correctly. Reason for the wrongly detected library version can be, that ProGuard removes parts of the code which are not needed and even changes the structure of classes. After this transformation, the library can be very similar to a different library version, therefore producing a wrong library version.

K-9 Mail includes the library *squareup.moshi*. This library was not contained in Ordol's library database at the time of evaluation, and therefore not detected. The streaming and binding mechanisms of *squareup.moshi* is similar to *google.gson*, which resulted in the wrong detection of *google.gson*.

Table 4.8: Ordol Detections in Unobfuscated App

Library	**Version**	**Correct Version**
apache-mime4j-core	0.7.2	0.7.2
apache-mime4j-dom	0.7.2	0.7.2
bumptech.glide	3.6.1	3.6.1
google.gson	2.3.1	false positive
squareup.okio	1.10.0	1.11.0
android.support-v4	23.1.1	23.1.1

Table 4.9: Ordol Detections in Obfuscated App

Library	**Version**	**Correct Version**
apache-mime4j-core	0.7.2	0.7.2
apache-mime4j-dom	0.7.2	0.7.2
bumptech.glide	3.6.1	3.6.1
google.gson	2.3.1	false positive
squareup.okio	1.7.0	1.11.0
android.support-v4	21.0.2	23.1.1

4.6.5 Conclusion

Libraries in Android apps are packaged into the code base of the app. This is problematic for analysis of apps, as it is not known if found issues arise from the core code of the app, or from included libraries.

This section showed two approaches for distinguishing library code from core code. The simple approach of detecting the library based on its package name has the downside, that this information can be easily changed by obfuscation. A more sophisticated approach – Ordol – was therefore designed to cope with such obfuscation and not only detect the library after obfuscation, but also the version of the included library.

Ordol uses flow-based k-grams to match methods contained in the app with methods from the library, to determine if a library is contained in the app. To do so, a library database has to be created beforehand, which includes all detectable library versions. If libraries or certain library versions are missing from this database, this will result in imprecise results of Ordol, as shown in the evaluation of Ordol against the publicly available tool LibRadar.

Once the library database contains a large number of libraries, Ordol can be used to detect libraries inside apps with a lower false positive and lower false negative rate than the most advanced tool LibRadar.

4.7 Data Leak Analysis by Injecting Targeted Taint Tracking

An issue prevalent in several Android apps is data leaks. In the past, several incidents of data leakages in apps have come to light which show the criticality of the problem. A study of 10.000 apps of the most popular Android Apps from the Google Play Store was conducted in December 2013 which showed that every twentieth app sends the phone's IMEI to a server at startup of the app [12].

Approaches to detect data flows in apps exist in two flavours: dynamic taint analysis and static taint analysis. Those approaches monitor data coming from a so-called source, being processed by the app, and passed to a sink. A source can e.g., be the retrieval of the IMEI, and a sink the sending of data to the Internet. The most prominent examples are TaintDroid [83] for dynamic analysis and FlowDroid [65] for static analysis.

This section focusses on dynamic taint analysis, i.e., the app will be executed during analysis, and data flows will be observed as they occur. Whilst dynamic analysis is dependant on an execution strategy (see Section 4.3.3.6), all found issues are true positives and indicate a data flow contained in the app. Previous work, i.e., TaintDroid requires a modified OS to perform its data flow analysis, which makes it harder to deploy to real or virtual environments, as each update of the OS potentially requires adaptations of TaintDroid.

Our approach presented in [12] overcomes this limitation by adding taint analysis directly to the app itself, which allows the analysis to be performed on a unmodified Android OS. The approach – called AppCaulk – analyses an application statically to find all possible data flows through the app. The app is then modified to keep track of the variables along such data flows. If tainted data (i.e., data read from a source) reaches a sink, a message can be written to the log indicating the data leak. In addition, [12] introduced a policy language to make the approach extensible and allow the specification of different data flows which should be detected during runtime.

> *AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps* [12] was published in *Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Bejing, China, 2014*, co-authored by Julian Schütte, Dennis Titze, and J. M. de Fuentes.

To evaluate the approach, AppCaulk is applied to the most popular Android apps and its effectiveness is compared against dynamic taint analysis using TaintDroid.

4.7.1 Approach Overview

The approach shown in this section is based on AppCaulk and aims at discovering data leaks in Android apps. Differing the the original approach published in [12], the technique is slightly modified: whilst the original goal of AppCaulk was to perform a static data flow analysis and include a dynamic data flow analysis to *block* data leaks, the main goal in this section is to *detect* such data leaks. This can be handled by AppCaulk, as

it includes a policy engine to specify which actions to take once a data flow is detected during runtime.

As static data flow analysis can only analyse which data flows *can* occur in the app, executing the app afterwards with a dynamic analysis shows which data flows *actually* occur. Benefit from the beforehand static data flow analysis is the minimization of app modification: only paths where data can potentially flow have to be instrumented, all other paths remain unchanged. This results in much less instructions needing instrumentation.

An Android app can be seen as a directed graph P of basic blocks, each containing n instructions $s \in S$ (S denoting the set of instructions). A potential data leak is contained in the app, if a directed, non-cyclic subgraph of P, starting at a *source* statement (e.g., reading sensitive information from a contact data provider) to a *sink* statement (e.g., sending data to a server), exists.

To discover data leaks at runtime, monitoring code will be inserted to track the status of all variables which are used in the data flow. To do so, a static data flow analysis detects all potential data leaks in the app. This analysis result is then used to insert as little instructions as possible to follow the data as it is transferred from variable to variable through the app.

The first step is to define what constitutes a data leak. To do so, a policy can be defined which specifies the sources, sinks and the propagation of the taint status through the app. This policy is essential for the following data flow analysis, as it specifies in detail how the analysis has to be performed.

To analyse the app for potential data leaks, the app's code is first disassembled into Smali instructions and subsequentially into basic blocks. Based on these basic blocks, the inter- and intraprocedural CG and CFG are calculated. These graphs specify how the execution is performed, i.e., which instruction is executed after one another. To perform the actual data flow analysis, the code is traversed twice: in the backward slicing step, all execution paths leading to a sink are analysed, and all paths not leading to a sink are eliminated. The next step performs a forward slicing, which traverses the code starting at all sources and eliminating all paths not starting at such a source. Essential for performing both backward and forward slicing it the taint propagation logic which specifies how the data is propagated from one variable to another. This propagation logic is based on the well-known *taint propagation* concept by Schwartz et al. [137]. It specifies in detail how registers are *tainted* or *untainted* by certain instructions. To offer more flexibility, user-defined propagation rules can be added to this policy. These rules can contain different taint levels, which enable classification of the data, e.g., classifying data as 'personal information' or 'confidential'.

After static data flow analysis, the app is instrumented to track all data flows and perform certain actions once a data leak is detected.

As a final step, the app is run on a virtual or real device and interacted with. Once a data leak is detected, the instrumented action is performed, and the user can decide how to act on this action. Possible actions are the displaying of a dialogue to the user to ask if the data leak should be performed. Another important action which can be used to analyse apps is to simply log the data leak.

4.7.2 Policy Definition

Data leak analysis produces different assertions about an app, depending on what is considered a data leak. The assertions e.g., depend on the definition of sources and sinks, but also on the definition of the propagation logic. As these definitions can depend on the use-case of the app, the definitions should not be set globally, but should be configurable. To do so, the following policy-based approach is introduced, to allow use-case-specific data leak analysis.

Sources are instructions of the app indicating either program entry points or method invocations (e.g., reading of sensitive information). If the source is a method invocation, the return value is considered tainted using the given tag. The example source definition in Listing 4.20 is the reading of the device's IMEI via `getDeviceId()`. The resulting string is tainted using the tag `TAG_IMEI`.

```
1 Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;[TAG_IMEI]
```

Listing 4.20: Example Source Definition

Sinks are instructions of the app where tainted data must not leak. Sinks are method invocations with additional information to which parameter the data must not leak. Listing 4.21 shows an exemplary sink definition where the 0^{th} parameter `HttpUriRequest` should not be tainted once the method is executed.

```
1 LHttpClient;->execute(LHttpUriRequest;)LHttpResponse; [0]
```

Listing 4.21: Example Sink Definition

Tags are a set of string constants to classify tainted data. Example tags are: `TAINT_IMEI`, `TAINT_GPS`, etc.

Propagate defines a set of propagation logic rules which propagate the taint status. For each possible instruction of the Smali bytecode, it defines how a tag is propagated from input registers to output registers. The example in Listing 4.22 shows how the taint status for the instruction `aget` is propagated. The instruction `aget vx, vy, vz` retrieves the entry vz of array vy and stores the result in vx. The propagation definition example states that if either vy or vz are tainted with tag `L`, the result vx is tainted with the same tag.

```
1 aget: 0:L ← 1:L|2:L
```

Listing 4.22: Example Propagation Definition

Untaint defines a set of propagation logic rules which remove the taint status from the result variable. The definition is similar to `Propagate` and setting the resulting tag to ∅.

APIPropagationRules define propagation rules not on a instruction level, but on method level. Whilst `Propagate` and `Untaint` specify the logic for each Smali instruction. `APIPropagationRules` can specify logic for method invocations. The example in Listing 4.23 shows how such a propagation can be defined for the function substring: the function retains its taint status if the first parameter is tainted. If a further parameter is tainted, the taint status is not propagated to the return value.

```
1  substr: 0:L ← 1:L
```

Listing 4.23: API Propagation Definition

These rules are especially important if the code for certain API-functions is not available during static analysis. This can e.g., be the case if the exact target OS version of the app is not known.

Countermeasure defines the measure to be taken once a data leak is detected at a sink. The measure is a class which has to implement a `CounterMeasureHandler` which handles data reaching a sink. This class receives the relevant data at the sink, including the program counter, tainted variables, taint tags, etc. The example countermeasure definition in Listing 4.24 states that the class `LogDataFlow` is called if the method `<sink>` is called with a taint tag `L`. The functionality of this class is not defined here, but has to be implemented manually and added to AppCaulk. In this example, `LogDataFlow` only logs the data to the Android log and then continues the execution of the app.

```
1  <sink>:L LogDataFlow
```

Listing 4.24: Countermeasure Definition

4.7.3 Static Data Flow Analysis

Using the policy definition, the next step is to perform static data flow analysis.

This is achieved by using a bi-directional inter-procedural data flow analysis. This analysis slices the program code accordingly and returns only those instructions which are potentially included in a data flow from a source to a sink. During the analysis, further propagation channels are considered which reside outside the execution of the dex code, e.g., propagation over files or via the Android framework. These channels are referred to as *external tunnels.*

As AppCaulk uses the static data flow analysis of Apparecium, details about backward and forward slicing are explained in Section 4.4.

4.7.3.1 Backward Slicing

First step of the static data flow analysis is backward slicing. This algorithm starts at all sinks, and iterates the CFG backwards, towards the start of the program.

Goal of backward slicing is to produce a slicing of the program containing all control flows along which data can be propagated to a sink.

4.7.3.2 Forward Analysis

To further reduce the number of instructions which need to be monitored and therefore instrumented, an additional forward data flow analysis is performed. This step is similar to the backward analysis, but starting at all sources and iterating the CFG in a forward direction.

4.7.3.3 Creation of Annotations

After calculating the backward and forward slices, both are merged to create *annotations* for each instruction where a variable is tainted. These annotations guide the subsequent instrumentation.

The merging algorithm takes the result of the forward and backward analysis as input, and only considers instructions which appear in both.
The algorithm then produces annotations called taint or untaint variable, which indicate if the current instruction taints or untaints a variable. Apart from single instructions, the merging algorithm handles the following cases:

- *Method calls:* tainted method parameters are tainted inside the method before the first instruction.
- *Method returns:* the taint tag of the return value inside a method is mapped to the variable which stores this return value in the calling method.
- *Member variables:* tainted member variables are stored and read depending on the current instance.
- *Exception:* taint states are propagated over exception handlers

4.7.4 Addressing External Tunnels

Forward and backward analysis relies on a control flow path through the analysed application. But besides paths residing on the control flow path, additional paths can exist which allow data leakage from a source to a sink. These paths are referred to as *external tunnels*.

External tunnels can propagate the taint tag via files, databases, content providers, SharedPreferences containers, intents and broadcast receivers (see Figure 4.31). In order to perform a data leak via an external tunnel, the tainted data has to be sent to an outside party (e.g., a file), and retrieved again at a later point in the control flow.

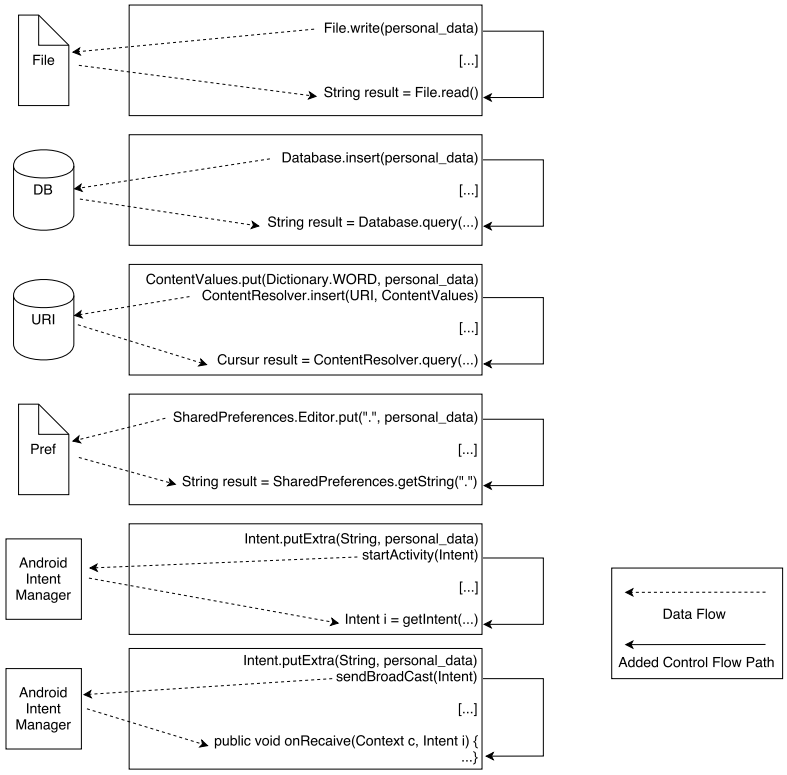


Figure 4.31: Addressed External Tunnels

To address external tunnels, the CFG of backward and forward flow analysis is extended. For instance, in the case of an external tunnel via files, whenever a file write is found in the control flow, an edge is inserted into the control flow graph to all locations where the file is read again. This new edge is then used both during backward analysis and forward analysis as the control flow graph is traversed. The same is done for databases, connecting *insert* and *query* operations, and similarly for the remaining external tunnels.

A special situation are broadcast receivers: the *sendBroadcast* method is the instruction where data is sent to the outside party. But differing to the previous cases, no retrieval operation exists. Instead, once a broadcast is received by the app, the *onReceive* method of the associated class is started. Therefore, the first line of this method is treated as the retrieval operation, and the corresponding edge is added to the control flow graph.

The limitation of this approach is its heuristic nature when addressing certain external tunnels. For instance, precise tracking of file handlers is not possible at bytecode level, as files can be moved or modified outside of the application's bytecode, e.g., by the OS. Therefore, we overapproximate such an external tunnel by connecting every file read with all file writes in the app. This results in false positives, but guarantees that no data flow is missed over an external file tunnel.

4.7.5 Bytecode Instrumentation for Dynamic Data Flow Tracking

The result of the static data flow analysis is a list of annotations containing all instructions potentially responsible for a data flow from a source $i \in S$ to a sink $o \in S$. These instructions are subsequentially instrumented such that the resulting app includes a taint analysis along these paths.

In this section, the procedure for dex bytecode instrumentation and the challenges to overcome are shown.

4.7.5.1 Type Inference

Variables in Smali bytecode do not contain type information. To correctly interpret the registers during taint propagation and to handle them in the countermeasure, the type has to be inferred. Although the exact type is not needed for the instrumented type inference, the distinction between *reference* and *non-reference* has to be made. This is required in order to instrument correct instructions into the app. Otherwise, the instrumented app would not pass Android's bytecode verifier and the app could not be installed and executed.

Inferring the type of a register is trivial in several cases. The instruction in Listing 4.25 trivially shows that $v2$ has to be an Integer and $v3$ has to be a String, as both are passed to a function expecting those types.

```
1  invoke-static {v2, v3}, L/Testing;->test(I L/java/String;)V
```

Listing 4.25: Trivial Type Inference

But this is not the case for all instructions. The example in Listing 4.26 does not reveal any information about the types of $v2$ or $v3$.

```
1  move v2, v3
```

Listing 4.26: Unknown Types in Type Inference

This problem can be solved by performing a type inference during the backward data flow analysis. For each method, the backward analysis will either at some point analyse an instruction where the the types are trivially shown (as in Listing 4.25), or reach the start of the method where the remaining parameters can be inferred from the method signature. In this analysis, no variable can remain with an unknown type, as all variables will either be parameters, return values, or variables used at some point in the method.

4.7.5.2 Runtime Stack Tracing

To achieve precise context-sensitive data flow tracking, information about the instance of a method and its caller at runtime are needed.

This information is not available in the Dalvik VM. Although the call stack can be inspected, references to the instances (e.g., memory addresses or unique identifier) cannot be retrieved. Therefore it is not possible to determine if the calling instance of a method is instance A or instance B of the same class.

To counteract this problem, a custom call stack can be built and instrumented into the app. This custom call stack writes the instance of the current method to a globally accessible stack before a method is called. Inside this method, the instance of the caller can then be retrieved to check if the parameters of a method are tainted for the caller's instance.

4.7.5.3 Injecting Dynamic Taint Analysis

To track the taint tag of each register, all tainted registers are tracked in a global tainting table. This is achieved by adding custom taint tracking classes to the app during instrumentation.

For each instruction in the annotations, a call to this taint tracking code instrumented, which includes all tainted variables, and the instance of the caller. Inside the taint tracking class, each register is assigned a globally unique identifier and its taint status is set as indicated by the static data flow analysis. Additionally the taint tag is stored for each register.

4.7.5.4 Handling Policy Violations

Before the call to a sink method, the handler configured in the policy is inserted into the app. This handler checks if the current taint status of the parameter of the sink is configured to trigger the countermeasure.

One possible countermeasure useful for dynamic taint analysis is to output the taint tag to the Android Log. Doing so allows the taint analysis to determine if a certain data leak is actually contained in the app.

4.7.5.5 Example Instrumentation

```
1 invoke-virtual source_function()
2 move-result v0
3 if-gt v0, 0x10, :end
4 const/16 v0, 0
5 :end
6 invoke-virtual sink_function(v0)
```

Listing 4.27: Before Instrumentation

```
1 invoke-virtual source_function()
2 move-result v0
3 __track_now_tainted("v0", "TAG1",
      ↪ this)
4 if-gt v0, 0x10, :end
5 const/16 v0, 0
6 __track_now_untainted("v0", this)
7 :end
8 __track_countermeasure("v0", this)
9 invoke-virtual sink_function(v0)
```

Listing 4.28: After Instrumentation

An example for the instrumentation is shown in Listing 4.27 and Listing 4.28.

The unmodified app (Listing 4.27) reads a register $v0$ from a source, and checks if it is greater than $0x10$. If it is, it jumps to label : *end* where the register is passed to a sink function. If not, the tainted register $v0$ is overwritten by the constant 0, and this constant is passed to the sink function.

After performing the static data flow analysis, three locations are contained in the annotations and have to be instrumented as shown in Listing 4.28. After the result is stored in $v0$, `__track_now_tainted` is called which adds $v0$ to the list of currently tainted registers (in *this* instance). The taint tag is set to 'TAG1', which would be configured by the policy for the source `source_function`. After the register is overwritten by a constant, a call to `__track_now_untainted` is instrumented, which clears the taint tag for the register $v0$. Finally, before the sink `sink_function` is reached, the call to `__track_countermeasure` is instrumented. If the execution reaches this instruction, `__track_countermeasure` checks if the register is tainted, and if so, performs the configured countermeasure.

In this example, the countermeasure would only be triggered, if the `if-gt` is true, i.e., the `source_function` returned a value greater than $0x10$. Otherwise, the `sink_function` would be called with a constant value, which is no data leak.

During static analysis it is not known if the app contains an actual data leak. But during dynamic analysis, only one of the paths is taken, either resulting in a data leak or not leaking any data.

4.7.6 Evaluation

As first evaluation, a simple app with exemplary data leaks was implemented. This example implements reading the IMEI and telephone number and leaking them to a server. The data propagation path includes methods of the app itself, the Android framework, exception handlers, and file accesses.

```
1  source {
2          Landroid/tele/TelephonyManager;->getDeviceId()Ljava/lang/String;
3                  [TAG_IMEI]
4          Landroid/tele/TelephonyManager;->getLine1Number()Ljava/lang/String;
5                  [TAG_PHONE]
6  }
7
8  sink {
9          Lorg/apache/http/impl/client/DefaultHttpClient;-><init>[0]
10 }
```

Listing 4.29: Example Policy

The policy has been set up specifically to prevent such data leaks, and includes the sources and sinks shown in Listing 4.29. The propagation rules are configured for all dex instructions in the policy. As a countermeasure, `LogDataFlow` is used to show the data flow in the Android Log.

Since taint tags are used to track the type of data being leaked, the type of data leaked can be given as additional information.

	Tested apps			AppCaulk	Taintdroid
Package name	Sources	Sinks	Methods	Data leakage?	Data leakage?
com.rhmsoft.fm	0	62	14514	No (no sources or sinks)	n/a
cn.wps.moffice_eng	1	29	39152	*	Not achieved
com.facebook.pages.app	1	4	36171	Yes	Yes
com.mobisystems.office	2	30	48308	Yes	Yes
com.allesklar.job	1	26	3492	No	Not achieved
de.arbeitsagentur.jobboerse	0	0	7587	No (no sources or sinks)	n/a
com.dataviz.docstogo	2	22	17748	Yes	Not achieved
com.estrongs.android.taskmanager	1	10	1647	No	Not achieved
com.netqin.ps	7	43	14355	Yes	Yes
com.indeed.android.jobsearch	0	8	2777	No (no sources or sinks)	n/a
at.tomtasche.reader	0	19	8260	No (no sources or sinks)	n/a
com.splashtop.remote.pad.v2	0	9	13560	No (no sources or sinks)	n/a
com.box.android	1	53	37023	No	Not runnable in emulator (crash)
com.dynamixsoftware.printershare	3	55	13170	Yes	Not achieved
com.allesklar.lehrstellen	1	34	6152	No	Not achieved
com.monster.android.Views	0	10	13101	No (no sources or sinks)	n/a
com.mobisystems.editor.office_with_reg	2	30	48709	*	Not achieved
mobi.infolife.smsbackup	0	3	1265	No (no sources or sinks)	n/a
de.joergjahnke.documentviewer.android.free	0	14	2436	No (no sources or sinks)	n/a
cn.wps.moffice_i18n	1	27	39649	*	Not achieved
com.olivephone.edit	1	43	48811	No	Not achieved
com.scout24.jobs	0	6	4925	No (no sources or sinks)	n/a
com.tux.client	0	7	2564	No (no sources or sinks)	n/a
com.rhythm.hexise.task	0	9	1128	No (no sources or sinks)	n/a
com.threebirds.wordreader	1	39	5148	No	Not runnable in emulator (crash)
com.stoik.mdscanlite	2	25	2508	Yes	Not runnable in emulator (not installed)
de.aok.gehaltsrechner	0	14	830	No (no sources or sinks)	n/a
im.ecloud.ecalendar	6	72	20327	Yes	Yes
com.intsig.BCRLite	7	39	11531	Yes	Not runnable in emulator (crash)
com.hi.applock	2	12	9353	No	Not achieved
com.netqin.mm	7	49	9615	Yes	Not runnable in emulator (crash)
com.threebirds.excelreader	1	39	5080	No	Not runnable in emulator (crash)
mobi.infolife.installer	0	30	9981	No (no sources or sinks)	n/a
com.sic.android.wuerth.wuerthapp	1	22	3742	Yes	Not achieved
de.interaid.heizoel24	0	7	4329	No (no sources or sinks)	n/a
com.citrix.Receiver	17	32	27562	Yes	Not achieved
de.sandnersoft.Arbeitskalender_Lite	0	9	3514	No (no sources or sinks)	n/a
com.samapp.excelcontacts.excelcontactslite	4	5	6913	No	Not runnable in emulator (crash)
com.tf.thinkdroid.amlite	1	35	31604	No	Not achieved
soma.de	0	12	3309	No (no sources or sinks)	n/a
com.nitrodesk.droid20.nitroid	5	47	35735	*	Yes
com.cisco.anyconnect.vpn.android.avf	2	2	4229	No	Not runnable in emulator (crash)
com.wyse.pocketcloudfree	0	35	13370	No (no sources or sinks)	n/a
com.zero8timetracking.timesheeter	0	27	16254	No (no sources or sinks)	n/a
com.IQBS.android.app2sd	0	11	1395	No (no sources or sinks)	n/a
com.ups.mobile.android	1	16	20125	No	Not achieved
com.infraware.polarisoffice.entbiz.gd.viewer	1	21	16493	No	Not runnable in emulator (requires PIN)
com.threebirds.officereader	1	43	5088	No	Not runnable in emulator (crash)

Table 4.10: Effectiveness Comparison of AppCaulk and TaintDroid (taken from [12])

4.7.6.1 Discussion of Performance Impact

One reason for combining dynamic taint analysis with a static data flow analysis was to limit the overhead caused by the instructions added to the app. A naive approach would have to instrument each statement with the taint tracking code, and instrument every method call with the custom stack tracking. Adding taint tracking code to each instruction would increase the size of the resulting app considerably which will very likely result in performance problems.

Limiting the instrumentation to potential data flows results in a drastic improvement which is shown in our example app. Although the app is small, it serves well as a benchmark, as it does not contain much functionality besides leaking data, and can therefore be considered a worst-case app. The example app contains 371 instructions which would have to be instrumented if no information about potential data flows would be used. After the static data flow analysis, only 39 instructions have to be instrumented, which is less than 11% of the total number of instructions. Instrumenting the app resulted in no noticeable performance impact when executing the app.

Besides the instrumentation of existing instructions, AppCaulk adds a fixed overhead of 1100 instructions for taint tracking and countermeasures.

4.7.6.2 Effectiveness Analysis

To show the practical applicability and the effectiveness of AppCaluk, it is tested against TaintDroid [83] with the 48 most popular Android apps from the Google Play Store as of July 2013.

To compare both techniques, AppCaulk's policy was configured with reading of the IMEI as source and all API calls which need the INTERNET permission as sinks. To find all methods requiring this permission, the permission map from [125] was taken as input. Doing so, all data leaks of the IMEI to the Internet are considered. This configuration was chosen, as the same configuration can be enabled for TaintDroid, which is the case in this test.

Table 4.10 summarizes the results of the evaluation. The comparison was composed of two steps: in a first step, all apps were analysed using AppCaulk. Four apps could not be analysed with AppCaulk due to timeout in the static data flow analysis (noted by * in Table 4.10).
In the second step, each app was executed manually in an environment with active TaintDroid. If the app did not have sources or sinks (shown in a manual static analysis), the apps were not executed with TaintDroid (shown as n/a in the table). This was the case for 19 of the 48 apps.

For the remaining 29 apps only 19 could be installed and run successfully in the TaintDroid environment. This is due to the fact that TaintDroid changes to the platform, which can result in a runtime error in the app (e.g., if specific parts of the framework are accessed directly). A second reason why not all apps can be analysed by TaintDroid is the fact that TaintDroid is always specific to one Android version. Since TaintDroid is not updated regularly to newer versions of Android, all apps which require the new

version of the OS cannot be analysed by TaintDroid. Four of the 19 apps analysed by TaintDroid could not be analysed by AppCaulk, so only 15 apps can be compared effectively.

Of all apps which can be analysed by AppCaulk and TaintDroid, 12 out of 15 returned the same result for both approaches. Due to the technique TaintDroid uses, its results can be seen as ground truth. Therefore the remaining 3 apps were false positives by AppCaulk.

All data leaks detected by TaintDroid were also detected by our approach if the analysis finished without timeout, i.e., AppCaulk did not miss any data flows which were detected by TaintDroid.

Table 4.10 also shows that AppCaulk was able to analyse all 10 apps which cannot be analysed by TaintDroid.

4.7.7 Conclusion

This section presented AppCaulk, a tool to instrument Android apps with data flow analysis. The instrumentation is performed along potential data flow paths identified by a static data flow analysis, therefore minimizing the modification of the app. Contrary to existing dynamic data flow analysis techniques, AppCaulk only modifies the app and does not require any modification of the platform.

AppCaulk is able to find data flows over external tunnels, e.g., files and databases. For this step, AppCaulk overapproximates data flows, as the context (e.g., the name of the file) is not taken into consideration. This was improved in later work, e.g., by Schütte et. al [138].

The practical evaluation showed that AppCaulk can effectively detect data flows in Android apps. The evaluation also showed that many apps (10 out of 29) could not be run using the TaintDroid environment, showing its limited applicability.

On the other hand, even though AppCaulk requires a static data flow analysis before the dynamic analysis, which can be time and resource consuming, the evaluation showed that the approach does not have the limitations of TaintDroid, and can be applied to all apps. The only limitation of AppCaulk stems from a limitation of computation-time during the analysis which is no general limitation of the approach.

For analysis, AppCaulk is therefore a very useful technique to dynamically check which data leaks actually occur during runtime of an app. As the required modifications are limited to the app itself, it can be easily used inside any execution environment, and easily transferred to newer versions of the Android OS.

4.8 Summary

This chapter introduces a framework for the analysis of apps, which can be comprised of different analysis techniques working together. Each analysis technique produces raw analysis data, which is subsequentially used to evaluate if the app adheres to specific requirements. The framework is designed to be easily extensible, both for new analysis techniques, as well as to new requirements.

The chapter further details, which analysis techniques are possible. In general, analysis techniques can be categorized as static or dynamic analysis: static analysis techniques analyse the app without running any part of the app and can further be distinguished into meta-data analysis and code analysis. Meta-data analysis analyses all parts of the app except the code itself. The code is analysed during code analysis.
Conceptually different to static analysis is dynamic analysis, which always executes an app for analysis. Dynamic analysis techniques can work with an unmodified app, or the app can be modified before analysis. For each analysis technique, distinguishing characteristics and their requirements to the app and the platform are shown in this chapter.

For each technique examples in the respective chapters show what each technique can achieve and how each analysis technique benefit from properties of the Android OS.

In the course of this thesis, several approaches to improve analysis techniques have been published which solve shortcomings of the current state of the art. These publications tackle the problem of static and dynamic data flow analysis, reconstruction of component transitions, and detection of libraries in apps.

CHAPTER 5

Automated Issue Mitigation

The previous chapter explained in detail which analysis techniques can be performed on apps. Once any analysis technique identifies an issue inside an app the question arises how to handle this issue.

In this chapter, different strategies will be introduced to perform automated mitigation. Depending on the issue, this mitigation can either be automatic, or it might only be solvable in a manual fashion. Goal of this chapter is to show the different possibilities to solve issues, and how issues can be classified according to the ability to mitigate them automatically.

Different mitigation classes exist which will be explained in detail in this chapter:

Automated Mitigation - Approved Generally: a developer has to approve mitigation for the class of issues once, and all such issues can then be fixed in an automated manner without further interaction.

Automated Mitigation - Approved Individually: a developer has to approve mitigation on a per issue basis. I.e., for each issue of this class and for each app containing this issue, the developer has to approve or deny the fix. If the fix is approved, it is automatically fixed in this instance.

Fixed Manually: if a fix is not possible in an automated manner, the developer can be given details about the issue, with possible fixes. The fixes have to be applied to the app manually.

The following sections explain the necessary requirements for automated issue mitigation (Section 5.1), and the different classes of issue mitigations, including examples for each class (Section 5.2). Section 5.3 introduces a process for issue mitigation for Android apps. Finally, Section 5.5 and Section 5.6 show the specific usage of issue mitigation in two examples, before Section 5.7 concludes this chapter.

5.1 Requirements for Automated Issue Mitigation

Performing automated issue mitigation requires the preceding analysis to detect all occurrences of a specific issue. This is especially important if an issue has to be mitigated at several locations. E.g., if files are stored in an insecure location, a possible fix is to encrypt the files before writing them, and decrypting the files as they are read. The resulting app will only perform as expected, if all such reading and writing locations are found during analysis. If one location is missed in the analysis, the resulting issue mitigation will likely produce a broken app.

A further requirement is the need for an exact location of the issue. If this location is not known, an automated issue mitigation cannot be performed. This can e.g., be the case for dynamically detected issues. During runtime of the app, communication with a server can be recorded in the dynamic environment which might not be linked to an exact location in the code where the communication was initiated. This can e.g., be the case if several threads are running in parallel during the communication.

Being able to modify the app is necessary to perform any automated issue mitigation. On Android this is possible, as the app can be transformed into a higher level representation (e.g., to Smali as shown in Section 2.2.5.1), and back to binary format.

5.2 Issue Mitigation Classification

Whether a fix exists which can be applied automatically depends on two things:

- if one or more fixes are available, and
- and if the parts of the app responsible for the issue interact with the outside.

If no fix for the issue is available, it is obvious that no automated mitigation can be performed.
Additionally, if the app interacts with the outside, the response from the outside is not known by definition. If this response is part of the issue, no fix can exists, as the response is not known.
Interaction with the outside does not necessarily mean with entities outside the device (e.g., a server), but can also mean interaction with other components (e.g., other apps) on the device itself. Such interaction is only considered as *real* outside interaction, if the other component (regardless if inside or outside of the device) further works with the data supplied by the app. If data is stored on the device outside the app, but retrieved only by the app later, this is not considered real outside interaction.

An example of such an issue is unencrypted communication with an outside server. As the issue mitigation might not know if the server is capable of encrypted communication, an automated changing of unencrypted to encrypted communication can very likely make the app unusable.

To perform automated mitigation, fixes have to be created beforehand for each issue which can potentially be found in the analysis step. Such a fix specifies exactly how an issue can be fixed, and under which circumstances it can be applied. The list of fixes has to be specified once, and can then be applied to apps.

Figure 5.1 shows how the decision is made if a fix can be applied in a manual or automated manner:

If no specific fix is available, the issue has to be fixed manually.

If a fix is available, *and* the app interacts with the outside, the issue has to be approved individually, since the response from the outside is not known. Since the fix can depend on the response from the outside, automated mitigation is not possible.

If a fix is available, and the app does not interact with the outside for this specific app and issue, the issue can be fixed in an automated manner, as all functionality is contained in the app, and can therefore be analysed and modified. If more than one fix is available, the process can either use any of the possible fixes, or delegate the decision which to use to the developer.

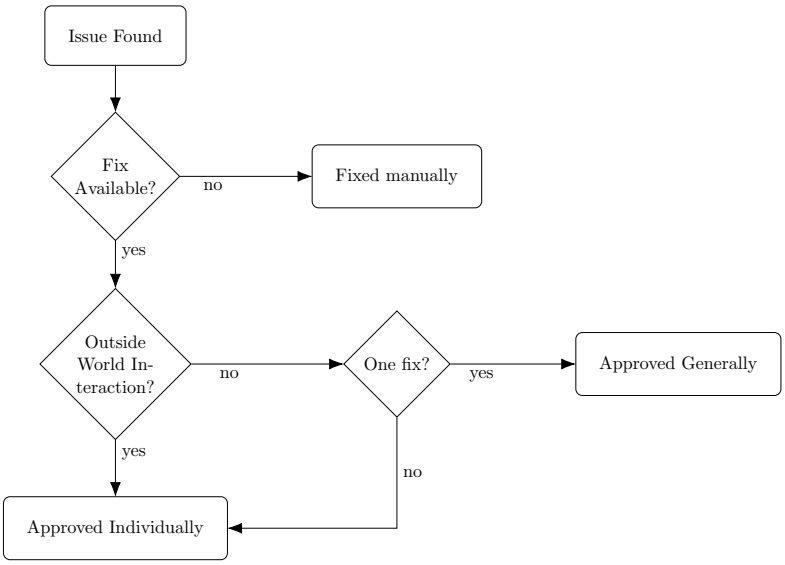


Figure 5.1: Issue Mitigation Classification

5.2.1 Manual Fix

If no fix is available, no automated process for mitigation can be applied. Instead, the issue can be shown to the developer with additional information to easily fix the problem manually. This information can include information about the exact location of the issue (if available), and how the issue can be fixed.

Examples for issues where a manual fix is required are:

Hard-coded Credentials: if credentials are left in the code, these can be a severe issue as they might allow other parties to access the services secured by those credentials. As these credentials might be needed by the app at some point, the developer needs to implement measures to remove them from the app. As possible fixes depend on the use-case of the credential, the developer has to address it manually. Possible fixes are to store the credentials in a secure location on the device (e.g., Android's keychain), or use techniques to make retrieval of the credentials difficult (e.g., Whitebox cryptography).

Insecure Library Version: if the app includes an older version of a library, this can result in vulnerabilities for the whole app, as both app and library have access to the same resources. As newer libraries might not be easily available, or the API of the library can change, this cannot be done automatically, but has to be done manually by the developer.

Data Flows with Unknown Location: if the dynamic analysis records a data flow, it might not be known where the flow originated. This can e.g., be the case, if only behavioural analysis of the app is performed, where the execution environments communication is monitored. In this case the developer can be given additional information (including the currently executed methods and content of the data flow), to help understand and remove the data flow.

5.2.2 Automated Fix and Individual Approval

Once fixes exist, these can be applied automatically. As recommended in the introduction of this chapter, no mitigations should be applied without the approval of a developer.

If either more than one fix exists for an issue, or outside services are involved in the issue, an automated mitigation algorithm cannot decide if the issue should be fixed, as the fix very likely would break the functionality of the app. In that case, the developer can be presented with the issue and the details about this specific issue. If the developer then decides that the issue can be removed in the proposed way, it can be mitigated automatically.

Examples where issues have to be individually approved before mitigation are:

Encrypted Communication: if the app communicates with a server, the analysis can determine if this is done encrypted or unencrypted. Adding encryption (e.g., standard algorithms like SSL), can be done in an automated manner, but since the server also has to be able to handle this protocol, the developer has to approve such a

fix. Adding encryption is a non-trivial change to the app, as several choices have to be addressed. This includes the exact configuration of encryption parameters (key size, encryption algorithm, signature algorithm, etc.) and the specification how the verification of server and potentially the client has to be performed. This can be done automatically, by determining which parameters the server accepts and using the strongest configuration, and a default verification mechanism (e.g., relying on Android to perform server verification). The alternative is to request this information from the developer and only apply the fix according to developer specification.

Data Leakage in Core Functionality: data leaks to files or to servers can occur in the core functionality of an app. As it is not known if such a leak is intended behaviour or a programming error, this can only be removed after developer approval. Removing such a data leak can happen by overwriting the data with benign values before the sink is reached. To distinguish core functionality from functionality in third-party libraries, the technique shown in Section 4.6 can be used.

Data Leakage in Third-Party Libraries: third-party libraries allow developers to include functionality into their code without the need for developing it themselves. When libraries are included into the app, developers might not analyse in detail which functionality they perform. As the libraries have access to the same data on Android as the core functionality of the app, this also includes access to sensitive data like contacts, and the IMEI. Whilst such data access might be needed, e.g., for libraries helping a developer to debug the app if it crashes, it might not be known to the developer that this data is accessed and e.g., sent to a server. If such a data leak is found in a library, the developer can decide if this is required and wanted behaviour. If not, such data leaks can be removed automatically, by exchanging data read from a source with benign data, or removing the call to the sink. Section 4.6 showed how core code of the app can be distinguished from included libraries.

5.2.3 Automated Fix and General Approval

If the issue is limited to the current app, and a fix exists, the issue can be fixed automatically, if a developer previously allowed this class of issues to be fixed. An example for this class of issues are SQL injections. If allowed, these can be automatically removed by replacing the problematic calls with secure variants.

The approval of the developer is needed, as the analysis system can not know if the functionality is intended, even if it is detected as an issue. The developer might e.g., access the database in an insecure manner, but requires it to perform a certain task. If the fix would be applied automatically, this intended task would be patched as well, and the app would not perform the same functionality any more.

Examples for issues which can be automatically fixed if approved are:

SQL injection: once input to a database query is not sanitized, this can lead to a SQL injection, e.g., if an attacker can control the input to this function. Such input

needs to be properly sanitized, or database access methods need to be used which are not susceptible to SQL injection. Removal of such an issue can e.g., be done by replacing calls to *SQLiteDatabase.rawQuery* with prepared statements. Listing 5.1 shows a SQL statement which is prone to SQL injections which can be replaced by a prepared statement as shown in Listing 5.2. Such a fix can be applied to the app automatically at all locations where a SQL injections is possible.

```
1 // possible SQL injection if str is not sanitized
2 db.rawQuery("INSERT␣INTO␣session␣(user)␣VALUES␣('" + str + "')");
```

Listing 5.1: SQL Statement Prone to SQL Injection

```
1 String query = "INSERT␣INTO␣session␣(user)␣VALUES␣(?)";
2 SQLiteStatement statement = db.compileStatement(query);
3 statement.bindString(1, str); // matches the ? in the query
```

Listing 5.2: SQL Statement not Susceptible to SQL Injection

Encrypt to Disc: data persistence is used for most apps, e.g., for storage of media data, or preferences. This storage can be done in an insecure manner, e.g., unencrypted in publicly accessible locations. As the static analysis can find all locations in the code where data is written or read from such locations, these operations can be modified so that the data is encrypted before writing data, and decrypted again as soon as the data is read. For such a fix, it is crucial that all reading and writing instructions are found in the app. If one is missed, this will lead to unencrypted data written to a file, or data not being decrypted before using it in the app, which will very likely break the functionality of the app.

Unvalidated Third-Party Code: apps can use functionality of the platform for critical functionality like encryption (e.g., using encryption algorithms from *javax.crypto* included in the Android framework). As the app does not generally have to trust such third-party libraries, functionality can be added to check the signature of these libraries. Such checks could either be signature-based checks of the library, effectively only allowing well known versions, or – as all code on Android is signed – the app can simply verify the signature of the library to only allow well known developers. Section 5.5 shows this mitigation in detail.

5.3 Mitigation Process

Mitigating issues in apps can be separated into two steps: detection of issues and correction of issue. The flowchart in Figure 5.2 shows this process.

First, the app is analysed for any contained issue (e.g., using the framework introduced in Section 4.2). The result of this is a list of issues found in the app. For each issue, a description of the problem, its exact location (if known), and one or more possible mitigations are output.

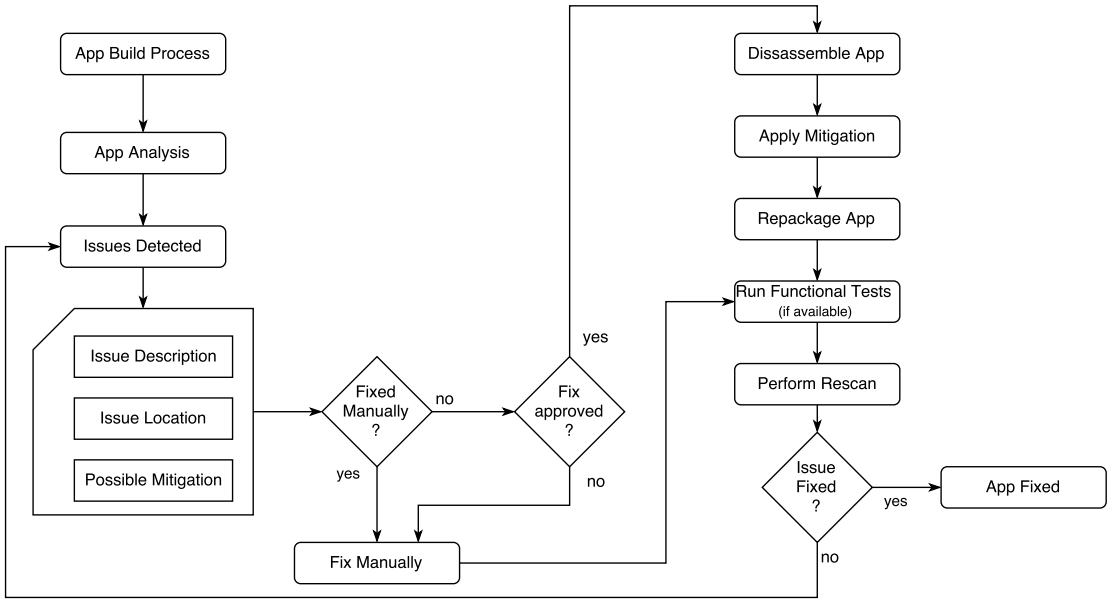


Figure 5.2: Issue Mitigation Process

After automatic of manual mitigation of the problem, the app is scanned again to ensure the issue is removed by the fix. If the issue is not fully mitigated, the process can be restarted until the issue is mitigated.

This mitigation process can be applied at different locations in the lifecycle of an app. It can e.g., be included in the build process of the app: after each build, an analysis and following issue mitigation process is attached. Another possible location is to apply the process once an app is published to an app store for distribution. This app store can analyse the app, and only allow the distribution if all issues are fixed (either manually or automated).

5.4 Limitations

The following two limitations remain for the automatic mitigation of an issue:

If no fix exists which can be applied to the specific issue, no mitigation can be performed automatically, and the developer has to fix the issue manually. No fix can exist for all issues which require a response from an outside system, as this response is not known by the mitigation process.

If a fix exists, it can be applied to the app to remove the issue. As the app is modified during this mitigation process, it cannot be guaranteed that the semantics of the app is maintained: e.g., as modifying statements changes the signature, any algorithm depending on this signature will produce different results after modification. It is therefore important to verify the functionality of the app after the mitigation is applied.

5.5 Issue Mitigation - Preventing Library Spoofing on Android

This section will show a typical issue of Android apps, which can be fixed in an automated manner with the help of a developer, namely library loading on Android. In particular, loading code from other applications also installed on the device.

> *Preventing Library Spoofing on Android* [10] was published in *Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Helsinki, Finland, 2015*, main author Dennis Titze, co-author Julian Schütte.

Dynamic loading of libraries is a widely used technique in Android applications. But including and executing external library code does not only have benefits, it can have severe detrimental security implications for the application and the user.

In this section we explain the mechanisms of loading external library code into an Android application and discuss resulting security implications. Since the app executes arbitrary code if the application does not perform the necessary verification, loading such code can introduce severe security problems. As a remedy, we present how external code can be verified and since currently available application often do not perform such verification, we introduce a novel way to enforce this verification. Such verification can be added in an automated manner to the app, e.g., after the build process of the app.

A prototype of this system has been published (`https://github.com/titze/android-library-caller` and `https://github.com/titze/android-library`) which can be easily integrated into existing apps and libraries.

For mobile app developers, reaching an audience is challenging. They need to create well-designed, innovative applications in a timely manner, which is no trivial task. Due to the short development and update cycles, many application developers rely on already existing libraries for various tasks, such as embedding advertisement, cryptographic functionality, ensuring compatibility with older devices, and many more.

But embedding library code directly into the code base of an application comes at the price that either the app has to be updated whenever any of the included libraries is updated, or the update is ignored and bugs remain unfixed.

As an alternative, external code can be dynamically loaded from already installed applications or external library files. Android allows to load bytecode via reflection from other code bases and execute it in the context of the calling app. Currently, many applications use this technique as a basis for plugin mechanisms (e.g., Google Analytics) or for sharing a common framework base among multiple apps (e.g., Adobe AIR).

However, while dynamic loading in such a way may be convenient and reduces the footprint of individual applications by externalizing shared code, it can have severe security problems. This is due to the fact that Android does not provide any means for verification of the loaded code by default, which enables an attacker to trick an application into loading the attackers' own code.

Typically the loaded code is only identified by the name of the library, i.e., either by the package name of the application providing the code, or by the path of a library file. If an app loads a library without proper verification, an attacker thus can simply set the name of her library to the one of the spoofed library and implement methods matching the headers of the original library. If the malicious library is then installed on the phone, applications will load and execute code from them without any further verification. Currently, Android does not provide any mechanisms to distinguish benign libraries from spoofed ones and consequently such attacks are not detected.

This section explains the problems of this commonly used programming pattern in detail and proposes remedies at different layers: the app itself, from within the library, or by the operating system. The section further shows how loading of external code can be used to enhance the security of an app.

5.5.1 Code Loading

Code loading is a common programming pattern on Android. It is done for example to dynamically load the appropriate code depending on the exact hardware. Another use case for dynamic code loading is to share a common code base among apps, for example in the case of frameworks executing cross-platform apps written in HTML5, .NET or any other language different from the natively supported Dex bytecode format. This section focusses on the technique of loading external code from library applications. A library application is similar to a normal app on an Android phone, but with the sole purpose of being included as a library by other apps. Such library applications typically do not contain a user interface and can not be started by the user.

Loading native code which is executed outside the Dalvik virtual machine is thus out of context of this work.

Various techniques exist to load code. The two most common techniques are to load it from a library file via *dalvik.system.PathClassLoader* or from another application using *dalvik.system.DexClassLoader*. An example for loading code from the external storage of the device is shown in Listing 5.3, where the external code is loaded into a new class loader *pcl*, which is subsequently used to get a class (com.example.External) using the Java Reflection API. This class is further searched for the specific method *doSth(String)*, which is then executed.

```
1 PathClassLoader pcl = new PathClassLoader("/sdcard/mylib.jar",
2                 ClassLoader.getSystemClassLoader());
3 Class cl = Class.forName("com.example.External", true, pcl);
4 Method me = cl.getMethod("doSth", String.class);
5 Object result = me.invoke(null, new String("test"));
```

Listing 5.3: Loading and Executing Code from an External Library

The code (*mylib.jar*) itself does not have to be available during compile time of the application, but the developer has to know the exact names of classes and method signatures which should be executed during runtime.

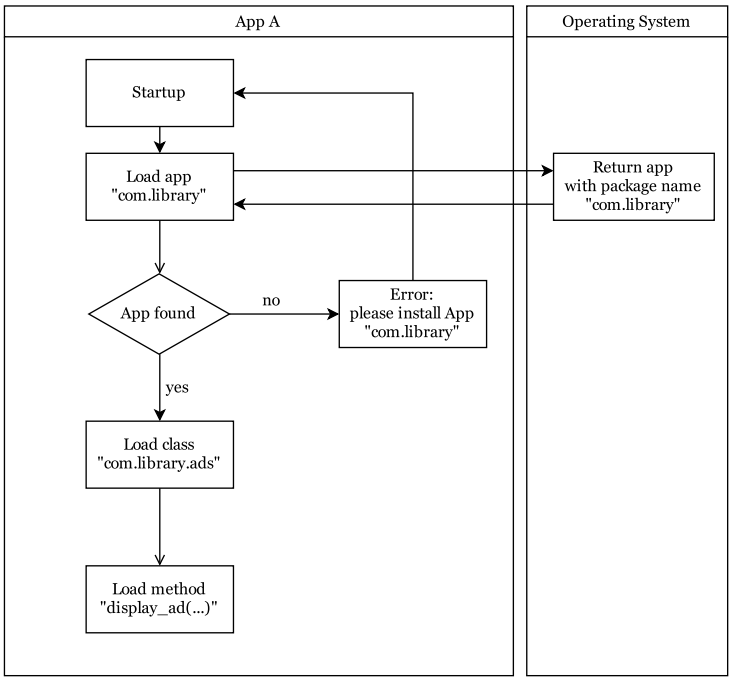


Figure 5.3: Loading External Code from another App

The second technique works very similar. Here, code is not loaded from a library file, but from a third-party app installed on the phone. A legitimate use case for this is again sharing a common code base among applications. A big benefit of requiring a separate library application is the decoupled update process. If a security update of the library is issued, it can immediately be deployed via the Google Play Store, rather than having to recompile all apps including that library. This improves the time to market and dramatically reduces development effort.

For this technique to work, the user must first install the library application and only afterwards, she is able to use any app relying on that library application. Figure 5.3 illustrates the process.

Loading external code does not require any verification, but rather relies on a simple identification by package name. This package name is unique on a phone and in the Google Play Store, but it is not bound to any secret of the legitimate developer. That is, whoever is the first to claim a certain package name either in the Google Play Store or by installing it on the user's device reserves the respective namespace.

5.5.2 Vulnerabilities Introduced by External Code Loading

Loading and executing code from external sources can have severe security implications. As long as the simple identification by package or path name is used instead of strong authenticity verification, the calling app might load and execute malicious code. This code is executed in the context of the calling application, i.e., it gets access to the

application's process memory and files, is granted all its permissions and may leverage them to access all resources of applications signed with the same developer key.

This is a major flaw in the Android security model, which actually aims at isolating apps from each other and requires user consent for an application to access specific resources. In the case of dynamic code loading however, the library application is treated as if it were part of the loading application's code base and is thus not subject to the isolation mechanisms, despite being a separate app which has been installed by the user and possibly not granted any permissions at all. While the user may grant permissions to an application A, the code actually using those permissions may be a combination of A along with all other applications installed on the phone, no matter which permissions have been assigned to them.

The mechanism of dynamically loading code does not require any authentication of the library app. Although apps have to be signed, Android does not perform any signature checks when external code is loaded. The *dalvik.system.PathClassLoader* accepts a jar or apk file which can be loaded from any where on the device. This location can for example be the external storage, a remote site, or an already installed app. Thus, any code from any installed application can be dynamically loaded and executed, requiring only the package name of the application. Listing 5.4 shows how a package location can easily be retrieved[1], without the need for any permission. The classes in the file can then be loaded as shown in Listing 5.3.

```
1  String apkPath = ctx.getApplicationContext().getPackageManager()
2                  .getApplicationInfo("com.ex.library", 0).sourceDir;
```

Listing 5.4: Location of a Package on Android

None of the possible install locations for such library apps are protected from tampering by attackers. Files on the external storage are typically world write- and readable so that benign library apps can easily be replaced by malicious code. As for remote sites, library download are susceptible to man in the middle attacks, as also explained in [101]. When loading code from already installed applications, there is no mechanism ensuring the origin and authenticity of that application. Thus, dynamic code loading is a prevalent attack vector for mobile applications, considering the simplicity of an attack, and the wide-spread usage in current applications.

5.5.3 Enhancing App Security by External Code Loading

One conclusion would be to regard dynamic code loading as a general issue and to abandon it right away. This would take away an easy to use programming pattern for plugin mechanisms, which can also have security benefits.

To enhance the security, an application can for example include a security library, which can analyse the integrity of the app and the system. This library can be called before

[1] Calling the package manager requires the caller to have a context object. But even if the context is not available, the location of the app can be retrieved, e.g., by issuing the shell command *pm list packages -f*.

critical functions, and check the checksum of the callee, e.g., against the checksum calculated by the app store. If these checksums differ, the app has been tampered with, and the library can for example inform the user about the security breach.

The library could also check that no known malware is running, e.g., by comparing the running processes against a blacklist of known malware, or check the integrity of the whole smartphone, i.e., checking if the phone is rooted, or has been tampered with.

The security of the library can be enhanced by linking it with a Secure Element (SE), which is a hardware element in the smartphone used for example for storing cryptographic keys. The attached SE can be used to secure the library itself against tampering.

When such security functionality is moved into the library, the application developers do not need to have the respective security expertise, they only have to include the corresponding library.

If such a library is available on the system, the app itself does not need to implement this security functionality itself, and can concentrate on its core functionality. The security library therefore enhances the security of apps on Android beyond the capabilities of the operating system. The library has to be well secured, since a compromised security library will again introduce all shown problems.

5.5.4 Securing External Code Loading

As Android does not provide any means to support secure loading of external libraries or apps, we present alternative techniques which developers can use to work around this missing security feature of Android. The techniques we discuss allow the app to verify the library on the one hand, and the library to enforce such verification on the other hand. Furthermore, we describe how the operating system could implement the verification by itself.

5.5.4.1 From the App's Perspective

Securing external code loading could be easily done from inside the app. Since the app knows which library should be loaded, it can either directly check the hashsum of the library, or check the signature of the app developer. Checking the hash of the app is trivial: since the app's location has to be known to load code, this location can be passed to a hashing function (e.g., to `java.security.MessageDigest`), and can then be compared against a fixed stored hash. But this technique removes the advantage of the external library, since any library update would require the app to change the comparing hash.

```
1 PackageInfo packageInfo =
    ↪ context.getPackageManager().getPackageInfo("com.example.library",
    ↪ PackageManager.GET_SIGNATURES);
2 for (Signature sig : packageInfo.signatures) {
3         MessageDigest md = MessageDigest.getInstance("SHA256");
4         md.update(sig.toByteArray());
5         if (!valid_signature(md.digest())) {
6         throw new LibraryVerificationException();
7         }
8 }
```

Listing 5.5: Signature Verification of an Android App

The more flexible solution is to check the signing certificate of the app. This certificate will stay valid for all future updates of the library, since 'you must always sign all versions of your app with the same key.' [139]. The signature can be checked as shown in Listing 5.5, where each signature of the application is validated against a stored signature. An Android app does not need any additional permissions to perform such verification.

5.5.4.2 From the Library's Perspective

In order for a library spoofing attack to succeed, an attacker would try to get the malicious library installed *before* the real library can get installed in order to reserve the package name or file name of a real library. Since the legitimate library is not installed at all in this case, there exists no direct way to improve the security and for example, warn the loading app, that this is not the legitimate library.

However, one solution is that the legitimate library forces any application to perform correct verification. This is done by forcing all apps which want to call functions of the library to implement a verification of the library. Before calling any function of the library, the library will require the calling application to implement a certain verification code immediately before the call (e.g., using code similar to Listing 5.5). The presence of this verification code can in turn be validated by the library before executing the actual functionality. Whenever the app does not perform such verification, the library will either warn the user or refuse execution.

Although a library does not have a direct benefit from including such enforcement, it might do so to force a certain level of security of all apps that include the library. Another reason can be to prevent reputation loss. If a certain library is used in many applications, it can be an attractive target for an attacker. This is the case for the library Adobe AIR, which is used in many Google Play Store apps. As discovered by Schütte et al. in the publication *ConDroid* [94], Adobe AIR is loaded without any verification, so an attacker can attack all apps that execute code from it. As shown, this is done by installing a malicious library with the real library not installed on the phone at all. This can result in a reputation loss for the library developer, since the library is the reason the app could be attacked. If no application would be able to use the library without verification, all apps requiring its functionality would include the verification, and an attacker would not be able to spoof this library any more.

This does of course not help in the case that the app does not perform such a verification, and a malicious library is installed before the valid library. But it will force apps which want to use the library's functionality to include verification. And since the app is intended to use the real library, it is therefore forced to implement such verification.

Figure 5.4 illustrates this process: the app intends to call the function `doSth` of the library 'com.example.external'. But before doing so, it correctly calls the function `Verification.verify`, which was provided by the library developer to check the signature of the library. As the function `doSth` is called (which is contained in the external

App

SomeClass

```
private void a(String lib) {
    do_something();
    something_else();
    if (Verification.verify(lib)) {
        String apk = context.getApplicationContext().getPackageManager()
                    .getApplicationInfo(lib, 0).sourceDir;
        PathClassLoader p = new PathClassLoader(apk,
                                ClassLoader.getSystemClassLoader());
        Class<?> cl = Class.forName("com.example.external", true,
                                                    pathClassLoader);
        Method me = cl.getMethod("doSth", String.class);
        Object result = me.invoke(null, new String("test"));
    } else {
        throw new LibraryVerificationException("Library not be verified");
    }
}
```

Verification

```
private static boolean verify(String lib) {
    PackageInfo pInfo = context.getPackageManager().getPackageInfo(
                            lib, PackageManager.GET_SIGNATURES);
    for (Signature sig : pInfo.signatures) {
      MessageDigest md = MessageDigest.getInstance("SHA256");
      md.update(sig.toByteArray());
      if (!valid_signature(md.digest())) {
        return false;
      }
    }
    return true;
}
```

Library

com.example.external

```
private void doSth(String input) {
    if (perform_caller_verification()) {
        perform_action();
    } else {
        log_error(input);
    }
}

private boolean perform_caller_verification() {
    Caller caller = extract_caller_from_Stacktrace(Thread.currentThread()
                                                    .getStackTrace());
    String apk = context.getApplicationContext().getPackageManager()
                                .getApplicationInfo(caller.app, 0).sourceDir;
    // get_class_signature can e.g. be implemented using baksmali,
    // or by writing a simple dex parser
    String class_signature = get_class_signature(caller, apk);
    if (class_signature == "67d7b4d5a1b7cafebabe1c9181066bfecd3b6280") {
        return true;
    } else {
        return false;
    }
}
```

Figure 5.4: Checking for the Correct Verification Code

library), the library checks if the verification took place. This is done by inspecting the byte code of the caller. The library only performs its functionality if the verification was done by using the exact library verification code provided by the library.

A proof of concept implementing all steps from the library and an example app is available on github[2]. It uses Smali [20] to translate the dex bytecode of the calling app into a readable assembly format which is then checked for the presence of the correct library verification code. This verification code checks the signature of the library against a certain signature, which will still be valid once the library is updated.

The performance overhead for the library code verifying the calling app depends on the processing power of the device. On a Galaxy Nexus S (GT-9023, Android 4.4.4) a successful verification takes around 766 ms, on a Galaxy Nexus (GT-I9250, Android 4.4.4) about 512 ms, and on a Motorola Nexus 6 (Android 5.0) only about 231 ms. A failed verification takes less time, and took 531 ms, 412 ms and 204 on the respective devices. Figure 5.5 shows the performance measurements on the three devices. 80% of the time spent in a successful verification run accounts to bakSmali, which disassembles the calling class. This time could be further improved by leaving out multiple parsing and disassembling steps of bakSmali, which are not required for this use case – either by developing an own parser for the dex file or by optimizing bakSmali for this use case. The total time could be further improved, by checking an app once at an initial call to the library and then again only if the app has been modified.

[2]Available on https://github.com/titze/android-library and https://github.com/titze/android-library-caller

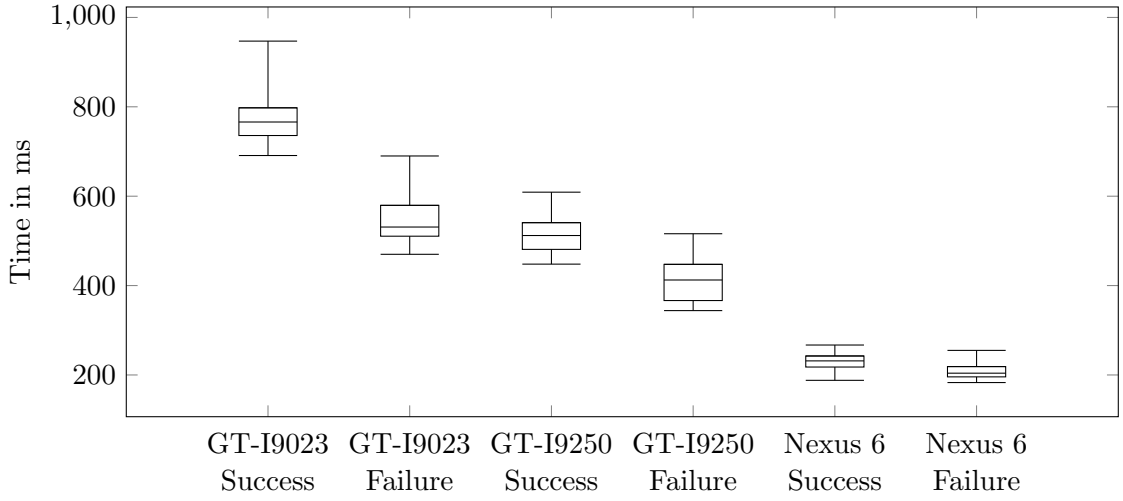


Figure 5.5: Runtimes of Successful and Failed Library Verification

5.5.4.3 From the Operating System's Perspective

Since the operating system has all possibilities for securing the external code loading, various possibilities exist, which are briefly explained here for sake of completeness.

Similar to the approach presented above, the OS could force any app to verify the signature of the library to be loaded. This can be done for example by requiring the app developer to specify the signature of the library developer, thereby strongly binding applications to the libraries they use. Once the app code is executed, the OS can then verify that the library is indeed signed with the specified developer key and only then permit the loading.

Another approach would be to treat the installation of shared code differently from the installation of a normal application, thereby taking into account the higher sensitivity of such code. As libraries are identified by package names or file names, it would be possible to have the user confirm whether an application named `X` shall be allowed to install a library `Y`, or whether `X` should be granted access to embedding code from `Y`. While this protection is certainly not equivalent to a certificate validation, it is a plausibility check which prevents library spoofing attacks from remaining unnoticed by the user.

A third approach is to only permit the loading of system libraries. This would remove the issue of loading malicious third-party code, but would also restrict app development.

The last approach is discussed in detail in [101], whose authors propose a operating system modification which only allows the loading of verified code, where the verification happens by different verification providers outside the smartphone.

Although these approaches would secure newer Android versions, it would not increase the security of older OS versions which are still used. Securing external code loading independently of the OS is therefore necessary to prevent library spoofing even in outdated OS versions.

5.5.5 Automatic Mitigation

Adding library verification can be done by the app developer, but this could also be done in an automated way.

If the app is analysed, and locations are found where external code is loaded and accessed, an automated mitigation process can instrument these locations. As neither the hash of the library, not the signature of the library's developer are known during mitigation, a fully automated process is not possible. Instead, the developer can be asked to input either of those values which will subsequentially be used for verification.

Using this information, a library verification code can be added to the app which checks the authenticity of the library (e.g., as shown in the method $perform\_caller\_verification$ in Figure 5.4). At each location where code of the library is actually executed, the automated process can instrument a call to this method, and only execute the library's code if the verification if successful.

5.5.6 Discussion

A prerequisite for securing external code loading is the absence of malicious apps with root privileges on the phone. Such apps have full access to all libraries and applications and can thus simply tamper with the verification code, thereby rendering all verification of the external code obsolete. Therefore the case of a malicious attacker having root privileges on the phone is out of scope for this work.

However, as seen in the previous sections, it is easy to spoof dynamically loaded libraries even without root privileges. Although it would not require much work for app developers to include application-layer verification, this is not a common pattern and we have not seen such verification in the wild yet. Since app developers are either not aware of this issue, or do care about it, we proposed an approach to enforce such verification from within the library.

Several reasons exist for a library to force such verification onto the app. This can be done to prevent reputation loss, if an attacker can impersonate the library, which could have been prevented by the library. Another reason can be that the library only wants apps to use its functionality which are developed in a secure manner, which includes correct verification of external code. If the library would offer security functionality as described in this section, it would only improve the security of the app, if the app adheres to the development requirements stipulated by the library. Checking these from the library ensures the compliance of the app to these requirements.

5.5.7 Conclusion

In this section we showed how external library code can be loaded into an Android application. Loading such code introduces severe security problems for the app, since it is possible for an attacker to impersonate a library which is not verified correctly.

Since most apps currently do not perform such verification, we introduced a way to force the verification from the library side. If a library developer decides to force such verification, any app including and using this library has to perform the verification before being able to use any of its functionality.

Since we have not seen any apps perform library verification, this gives the library developers a chance to force secure loading which increases the security of external code loading drastically.

A proof-of-concept showing all necessary steps to prevent library spoofing has been developed and published on github. It shows the feasibility of such verification and verification enforcement, and can be easily integrated into existing applications.

Adding library verification to an app is an example for a semi-automated mitigation process. The verification can be added to the app e.g., after its build process. The only manual step a developer has to perform is to supply the hashsum of the library which should be loaded, or the signature of the library's developer.

5.6 Issue Mitigation - Data Leak Prevention

Another issue which can be mitigated in an automated manner is presented in the following publication:

> *AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps* [12] published in *Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Bejing, China, 2014*, co-authored by Julian Schütte, Dennis Titze, and J. M. de Fuentes.

This technique has been shown in detail in Section 4.7. Whilst the technique is used for analysis in Section 4.7, it can also be used to mitigation of issues.

AppCaulk analyses an app statically for possible data leaks and adds a so-called countermeasure to all possible sinks, i.e., to locations where sensitive data can leak. In the previous chapter, this countermeasure is tailored to analysis. The alternative – as proposed our original publication – is to add a countermeasure to the app which informs the user about any data leak which is about to happen.

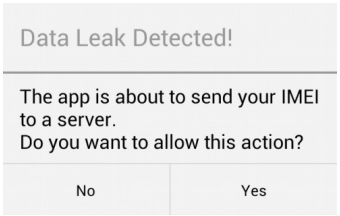


Figure 5.6: Injected Confirmation Message

In this example, the issue found in the app is the data leak to an external source. As such a leak can be undesirable to a user of the app, automated mitigation can be applied to react to these data leaks. As it is not known to the mitigation strategy, why the app sends the data, i.e., if it a legitimate use-case of the app, or unwanted behaviour, the responsibility of accepting or denying the leak is handed to the user. At each location where data is sent to an external source the mitigation strategy adds a dialogue box into the app which asks the user if she wants to accept the sending of data to an external source. Figure 5.6 shows the injected dialogue box. The data is sent to the sink only if the user accepts this data leak. Otherwise, the sink-function is skipped, and the execution of the app continues without sending any data.

As the user can choose not to send data to the external source, this changes the functionality of the app. If the data is required by the external source for the app to work correctly, skipping this step results in a broken application. In this example, it is the responsibility of the user to make this decision. If the app does not work as expected, the alternative is to restart the app and accept the data leak.

AppCaulk can be applied automatically to the whole app, thereby effectively allowing a user to choose which data leaves the app. This technique is an example for the mitigation class ‘Automated Mitigation - Approved Generally’.

5.7 Summary

This chapter shows under which circumstances issues in apps can be mitigated in an automated manner. Depending on the type of the issue, it can either be fixed automatically if a fix for the issue is available, or has to be fixed manually. Fixes for each issue have to be configured once beforehand, e.g., by the developer of the analysis technique before any automated approach can mitigate it. A fix for an issue can be to remove the problematic instructions from the app, or modify them in such a way that they are not problematic any more.

Two issue mitigation instances are shown in this chapter: in the first, the problem of library verification is tackled. As apps can use libraries installed on the system, an attacker can be able to exchange these to change the functionality of the app. The remediation for this issue is to verify the library which is used. If this is missing from an app, it can be added in an automated manner by the mitigation process.
The second mitigation shows how data leaks can be prevented in an automated manner. A general prevention of all data flows is not desirable in an app, as the sending of data to an external source can be part of the benign functionality of the app, e.g., to authenticate a user. Therefore the decision if a specific data flow is allowed or should be prevented is handed to the user who has to make this decision. If a data leak is found during analysis, the mitigation process automatically adds a dialogue box asking the user to accept or deny the data flow.

The mitigation process shown in this chapter is essential to mitigate the detected issues in an app. The process can be applied at different stages of the app development cycle, either during the build process or as the app is published to an app store.

CHAPTER 6

Conclusion

This thesis answers the question how to analyse apps on Android, and subsequentially how to fix the found issues in an automated manner. Analysing apps for mobile devices is important for finding privacy violations, security related issues or programming errors.

Analysis techniques can analyse apps either statically or dynamically. During static analysis, all content of the app, including contained files and the contained bytecode is analysed. The app is not executed during static analysis, therefore all issues are potential issues which can be executed during runtime of the app, or might not be executed, depending on the execution path that is taken during runtime. Conceptually different, dynamic analysis actually runs the app. Issues found during this analysis are true positives, as a call path is executed and the behaviour during this execution is analysed.

In the course of this thesis, several techniques benefiting from the proposed properties have been published to improve existing analysis techniques. These publications improve static and dynamic data flow analysis, reconstruction of component flows, and detection of libraries in apps.

Previously published data flow analysis techniques require a time and resource intensive entrypoint analysis of the app which is used as starting point of the data flow analysis. In contrast to this, our approach calculates the data flows directly from all sources to the sinks of the app. This allows faster analysis results, and can potentially find more data flows, if these do not occur on the control flow from an entry point of the app. The evaluation of Apparecium showed that the approach is able to analyse more apps in a real world analysis scenario where the analysis is performed on commodity hardware.

Current publications concerning Android Activity Flow Reconstruction focussed on a precise reconstruction by relying on time and resource intensive data flow analysis. Our approach chooses a different approach of reconstructing transitions in a data flow agnostic manner. Although this leads to a decreased accuracy, it is not as time and resource

consuming as a data flow analysis, and is therefore a viable approach if runtime is more important than accuracy.

Differentiation of Core Code and Library Code is important e.g., to determine if found issues originate in a library, or in the core of the app itself. Previous approaches do not work reliably with obfuscated code, which is solved by our approach. This approach is able to not only detect the name of the library with high accuracy, but also the version of the library with a high accuracy.

Data Leak Analysis by Injecting Targeted Taint Tracking focusses on dynamic taint analysis, i.e., data flows will be observed as they occur. Previous work required modifications of the Android OS to perform its analysis. As this is specific to one Android OS version, transferring it to a newer version or different hardware is difficult. Contrary to this, our approach adds taint analysis directly to the app itself, which allows the analysis to be performed on a unmodified Android OS.

Following the analysis of apps, the question arises how the found issues can be mitigated. This thesis shows for which class of issues an automated mitigation is possible, and under which circumstances manual mitigation is needed. Automated mitigation can be performed once solutions to the detected issues are available. Therefore, solutions to fix each issue have to be included into the automated mitigation process, e.g., by the developer of the analysis technique.
The mitigation process can then be integrated in the build process or the publishing process of an app, and remove issues in an automated manner.

Two instances of automated issue mitigation are shown in this thesis. The first improves library verification to prevent library spoofing on Android. This is done by adding code to the app which verifies the signature of the loaded library before it is executed.

The approach to perform dynamic data flow analysis shown in the analysis chapter can also be used for automated mitigation of data leaks. In this case, the app is enriched with a data flow analysis and a countermeasure. The data flow analysis tracks sensitive data across the app. Once such data reaches a sink function, e.g., a function which would send the data to a remote server, the countermeasure is executed, which prevents the data leak completely, or asks the user if the data leak is allowed.

During this thesis, several areas have been identified which are interesting to investigate in future work:

In the proposed framework, the selection which detector has to be run for which analysis has to be done manually by the developer of the analysis technique. Future work could improve this to automatically determine which analysis results are required to check for a certain issue.

The approach shown in Section 4.4 determines component transitions without data flow analysis. As shown, this results in a high imprecision, which could be mitigated in the future by allowing data flow analysis on only small parts of the code. This could lead to similar speed improvements as our work, but potentially achieve higher accuracy.

Both proposed approaches for data flow analysis do not take native code into account. Whilst it is theoretically possible to apply the approach also to native code, further

research has to be conducted to realize this. It is especially interesting to determine how the data flow analysis can handle the transition from bytecode to native code and back.

One problem during dynamic analysis is the dependency on the outside world once apps e.g., communicate with remote servers. Further research could try to solve this problem by simulating the outside world to some extent, to allow dynamic analysis even without outside communication.

For automatic fixing of found issues, a solution for each issue has to be developed and configured before it can be applied. Further research has to be conducted to evaluate if such a solution can also be determined by a self-learning system.

Finally, the approaches shown in this thesis fix issues in the app itself. Further research should be conducted to determine if classes of issues can be fixed in the operating system itself, thereby allowing all apps running on the system to benefit from the fix.

List of Figures

List of Tables

Listings

Acronyms

ART Android Runtime

AFG Android Component Flow Graph

AOSP Android Open Source Project

API Application Programming Interface

BM Birthmark

CFG Control Flow Graph

CA Certificate Authority

DFG Data Flow Graph

DVM Dalvik Virtual Machine

GUI Graphical User Interface

ICC Inter Component Communication

IDE Integrated Development Environment

IMEI International Mobile Equipment Identity

IPC Inter Process Communication

IR Intermediate Representation

JVM Java Virtual Machine

MWM Maximum Weight Matching

OS Operating System

RPC Remote Procedure Call

SE Secure Element

UUID Universally Unique Identifier

UID User-ID

AWS Amazon Web Services

SMS Short Message Service

DoS Denial Of Service

mTAN Mobile Transaction Authentication Number

Bibliography

[1] Hecht, Matthew S. ; Ullman, Jeffrey D.: Analysis of a Simple Algorithm for Global Data Flow Problems. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1973 (POPL '73), 207–217

[2] Android Developers Blog: *Announcing the Android 1.0 SDK, release 1.* https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html, accessed 09.03.2017

[3] Statista: *Number of mobile phone users worldwide from 2013 to 2019 (in billions).* https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/, accessed 09.03.2017

[4] IDC: *Smartphone OS Market Share, 2016 Q3.* http://www.idc.com/promo/smartphone-market-share/os, accessed 09.03.2017

[5] Statista: *Number of apps available in leading app stores as of June 2016.* https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, accessed 09.03.2017

[6] IDC: *Smartphone Vendor Market Share.* http://www.idc.com/promo/smartphone-market-share/vendor, accessed 09.03.2017

[7] Viennot, Nicolas ; Garcia, Edward ; Nieh, Jason: A Measurement Study of Google Play. In: *SIGMETRICS Perform. Eval. Rev.* 42 (2014), Juni, Nr. 1, 221–233. `http://dx.doi.org/10.1145/2637364.2592003`. – DOI 10.1145/2637364.2592003. – ISSN 0163–5999

[8] Titze, D. ; Lux, M. ; Schuette, J.: Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications. In: *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, S. 618–625

[9] Titze, Dennis ; Weiss, Konrad ; Schütte, Julian: Ariadnima - Android Component Flow Reconstruction and Visualization. In: *Proceedings of International Conference on Advanced Information Networking and Applications*, 2017 (AINA)

[10] Titze, Dennis ; Schütte, Julian: Preventing Library Spoofing on Android. In: *Proceedings of IEEE International Workshop on Trustworthy Software Systems*. Helsinki, Finland, 2015

[11] Titze, Dennis ; Schütte, Julian: Apparecium: Revealing Data Flows in Android Applications. In: *Proceedings of the International Conference on Advanced Information Networking and Applications (AINA)*, 2015

[12] Schütte, Julian ; Titze, Dennis ; Fuentes, J. M. d.: AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps. In: *Proceedings of the International Conference on Trust, Security and Privacy in Computing and Communications*, 2014 (TrustCom)

[13] Titze, Dennis ; Stephanow, Philipp ; Schütte, Julian: A Configurable and Extensible Security Service Architecture for Smartphones. In: *Int'l Symposium on Frontiers of Information Systems and Network Applications (FINA)* (2013)

[14] Bornstein, Dan: Dalvik VM Internals. In: *Google I/O*, 2018

[15] *ART and Dalvik*. https://source.android.com/devices/tech/dalvik/, accessed 20.03.2017

[16] Apple: *Maintaining Your Signing Identities and Certificates*. `https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html`

[17] Google Inc.: *Activity-lifecycle concepts*. https://developer.android.com/guide/components/activities/activity-lifecycle.html, accessed 20.03.2017

[18] Google Inc.: *Intents and Intent Filters*. https://developer.android.com/guide/components/intents-filters.html, accessed 20.03.2017

[19] Google Inc.: *App Manifest*. https://developer.android.com/guide/topics/manifest/manifest-intro.html, accessed 20.03.2017

[20] Gruve, Ben: *Smali - An assembler/disassembler for Android's dex format*. https://github.com/JesusFreke/smali, accessed 20.03.2017

[21] Google Inc.: *Dalvik bytecode*. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html, accessed 15.03.2017

[22] Sable Research Group: *Soot: A framework for analyzing and transforming Java and Android Applications*. https://sable.github.io/soot/,

[23] Bartel, Alexandre ; Klein, Jacques ; Le Traon, Yves ; Monperrus, Martin: 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*. New York, NY, USA : ACM, 2012 (SOAP '12). – ISBN 978–1–4503–1490–9, 27–38

[24] Vallee-Rai, Raja ; Hendren, Laurie J.: *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998

[25] Arnatovich, Yauhen ; Tan, Hee Beng K. ; Ding, Sun ; Liu, Kaiping ; Shar, Lwin K.: Empirical Comparison of Intermediate Representations for Android Applications. In: *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute Graduate School, 2014, S. 205–210

[26] Rastogi, Vaibhav ; Chen, Yan ; Enck, William: AppsPlayground: Automatic Security Analysis of Smartphone Applications. In: *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. New York, NY, USA : ACM, 2013 (CODASPY '13). – ISBN 978–1–4503–1890–7, 209–220

[27] Spreitzenbarth, Michael ; Freiling, Felix ; Echtler, Florian ; Schreck, Thomas ; Hoffmann, Johannes: Mobile-sandbox: Having a Deeper Look into Android Applications. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2013 (SAC '13). – ISBN 978–1–4503–1656–9, 1808–1815

[28] Bläsing, T. ; Batyuk, L. ; Schmidt, A. D. ; Camtepe, S. A. ; Albayrak, S.: An Android Application Sandbox system for suspicious software detection. In: *2010 5th International Conference on Malicious and Unwanted Software*, 2010, S. 55–62

[29] Lindorfer, M. ; Neugschwandtner, M. ; Weichselbaum, L. ; Fratantonio, Y. ; Veen, V. v. d. ; Platzer, C.: ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In: *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014, S. 3–17

[30] Lindorfer, M. ; Neugschwandtner, M. ; Platzer, C.: MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis. In: *2015 IEEE 39th Annual Computer Software and Applications Conference* Bd. 2, 2015, S. 422–433

[31] Xia, Mingyuan ; Gong, Lu ; Lyu, Yuanhao ; Qi, Zhengwei ; Liu, Xue: Effective Real-time Android Application Auditing. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2015 (SP '15)

[32] Portokalidis, Georgios ; Homburg, Philip ; Anagnostakis, Kostas ; Bos, Herbert: Paranoid Android: versatile protection for smartphones. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. New York, NY, USA : ACM, 2010 (ACSAC '10). – ISBN 978–1–4503–0133–6, 347–356

[33] Zheng, M. ; Sun, M. ; Lui, J. C. S.: Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013. – ISSN 2324–898X, S. 163–171

[34] Faruki, Parvez ; Laxmi, Vijay ; Bharmal, Ammar ; Gaur, M.S. ; Ganmoor, Vijay: AndroSimilar: Robust signature for detecting variants of Android malware. In: *Journal of Information Security and Applications* 22 (2015), 66 - 80. `http://www.sciencedirect.com/science/article/pii/S2214212614001471`. – ISSN 2214–2126. – Special Issue on Security of Information and Networks

[35] Faruki, P. ; Laxmi, V. ; Ganmoor, V. ; Gaur, M. S. ; Bharmal, A.: DroidOLytics: Robust Feature Signature for Repackaged Android Apps on Official and Third Party Android Markets. In: *2013 2nd International Conference on Advanced Computing, Networking and Security*, 2013. – ISSN 2377–2506, S. 247–252

[36] Huang, Chun-Ying ; Tsai, Yi-Ting ; Hsu, Chung-Han: Performance Evaluation on Permission-Based Detection for Android Malware. In: Pan, Jeng-Shyang (Hrsg.) ; Yang, Ching-Nung (Hrsg.) ; Lin, Chia-Chen (Hrsg.): *Advances in Intelligent Systems and Applications - Volume 2: Proceedings of the International Computer Symposium ICS 2012 Held at Hualien, Taiwan, December 12–14, 2012*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–35473–1, 111–120

[37] Sato, Ryo ; Chiba, Daiki ; Goto, Shigeki: Detecting Android Malware by Analyzing Manifest Files. In: *2013 Proceedings of the Asia-Pacific Advanced Network*, 2013

[38] Aung, Zarni ; Zaw, Win: Permission-based android malware detection. In: *International Journal of Scientific and Technology Research* 2 (2013), Nr. 3, S. 228–234

[39] Sanz, Borja ; Santos, Igor ; Laorden, Carlos ; Ugarte-Pedrero, Xabier ; Bringas, Pablo G. ; Álvarez, Gonzalo: Puma: Permission usage to detect malware in android. In: *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions* Springer, 2013, S. 289–298

[40] Sanz, Borja ; Santos, Igor ; Laorden, Carlos ; Ugarte-Pedrero, Xabier ; Nieves, Javier ; Bringas, Pablo G. ; Álvarez Marañón, Gonzalo: MAMA: manifest analysis for malware detection in android. In: *Cybernetics and Systems* 44 (2013), Nr. 6-7, S. 469–488

[41] Zhou, Yajin ; Wang, Zhi ; Zhou, Wu ; Jiang, Xuxian: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: *NDSS* Bd. 25, 2012, S. 50–52

[42] Book, Theodore ; Pridgen, Adam ; Wallach, Dan S.: Longitudinal Analysis of Android Ad Library Permissions. In: *CoRR* abs/1303.0857 (2013)

[43] Crussell, Jonathan ; Gibler, Clint ; Chen, Hao: Attack of the Clones: Detecting Cloned Applications on Android Markets. In: *Proceedings of ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–33167–1, 37–54

[44] Grace, Michael C. ; Zhou, Wu ; Jiang, Xuxian ; Sadeghi, Ahmad-Reza: Unsafe Exposure Analysis of Mobile In-app Advertisements. In: *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. New York, NY, USA : ACM, 2012 (WISEC '12). – ISBN 978–1–4503–1265–3, 101–112

[45] Narayanan, Annamalai ; Chen, Lihui ; Chan, Chee K.: AdDetect: Automated detection of Android ad libraries using semantic analysis. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*, 2014, 1–6

[46] Yang, Wenbo ; Li, Juanru ; Zhang, Yuanyuan ; Li, Yong ; Shu, Junliang ; Gu, Dawu: APKLancet: Tumor Payload Diagnosis and Purification for Android Applications. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. New York, NY, USA : ACM, 2014 (ASIA CCS '14). – ISBN 978–1–4503–2800–5, 483–494

[47] Aafer, Yousra ; Du, Wenliang ; Yin, Heng: DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*. Cham : Springer International Publishing, 2013. – ISBN 978–3–319–04283–1, 86–103

[48] Arp, Daniel ; Spreitzenbarth, Michael ; Gascon, Hugo ; Rieck, Konrad: *Drebin: Effective and explainable detection of android malware in your pocket*. 2014

[49] Steve Hanna, Edward Wu Saung Li Charles C. Ling Huang H. Ling Huang ; Song, Dawn: Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In: *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012

[50] Wang, Haoyu ; Guo, Yao ; Ma, Ziang ; Chen, Xiangqun: WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. New York, NY, USA : ACM, 2015 (ISSTA 2015). – ISBN 978–1–4503–3620–8, 71–82

[51] Ma, Ziang ; Wang, Haoyu ; Guo, Yao ; Chen, Xiangqun: LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. New York, NY, USA : ACM, 2016 (ICSE '16). – ISBN 978–1–4503–4205–6, 653–656

[52] Backes, Michael ; Bugiel, Sven ; Derr, Erik: Reliable Third-Party Library Detection in Android and Its Security Applications. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2016 (CCS '16). – ISBN 978–1–4503–4139–4, 356–367

[53] Grace, Michael ; Zhou, Yajin ; Wang, Zhi ; Jiang, Xuxian: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012

[54] Yang, Shengqian ; Yan, Dacong ; Wu, Haowei ; Wang, Yan ; Rountev, Atanas: Static Control-flow Analysis of User-driven Callbacks in Android Applications. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA : IEEE Press, 2015 (ICSE '15). – ISBN 978–1–4799–1934–5, 89–99

[55] Cao, Yinzhi ; Fratantonio, Yanick ; Bianchi, Antonio ; Egele, Manuel ; Kruegel, Christopher ; Vigna, Giovanni ; Chen, Yan: EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015

[56] Sounthiraraj, David ; Sahs, Justin ; Greenwood, Garret ; Lin, Zhiqiang ; Khan, Latifur: Smvhunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In: *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014

[57] Bartel, Alexandre ; Klein, Jacques ; Monperrus, Martin ; Traon, Yves L.: Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. In: *CoRR* abs/1408.3976 (2014)

[58] Au, Kathy Wain Y. ; Zhou, Yi F. ; Huang, Zhen ; Lie, David: PScout: Analyzing the Android Permission Specification. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2012 (CCS '12). – ISBN 978–1–4503–1651–4, 217–228

[59] Desnos, A.: Android: Static Analysis Using Similarity Distance. In: *2012 45th Hawaii International Conference on System Sciences*, 2012. – ISSN 1530–1605, S. 5394–5403

[60] Gascon, Hugo ; Yamaguchi, Fabian ; Arp, Daniel ; Rieck, Konrad: Structural detection of android malware using embedded call graphs. In: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security* ACM, 2013, S. 45–54

[61] Suarez-Tangil, Guillermo ; Tapiador, Juan E. ; Peris-Lopez, Pedro ; Blasco, Jorge: Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. In: *Expert Systems with Applications* 41 (2014), Nr. 4, S. 1104–1117

[62] Dienst, Steffen ; Berger, Thorsten: Static Analysis of App Dependencies in Android Bytecode. 0 (2014), S. 1–10

[63] Junaid, Mohsin ; Liu, Donggang ; Kung, David: Dexteroid: Detecting Malicious Behaviors in Android Apps Using Reverse-Engineered Life Cycle Models. In: *arXiv preprint arXiv:1506.05217* (2015)

[64] Octeau, Damien ; McDaniel, Patrick ; Jha, Somesh ; Bartel, Alexandre ; Bodden, Eric ; Klein, Jacques ; Traon, Yves L.: Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In: *USENIX Security Symposium* (2013), S. 543–558. ISBN 978–1–931971–03–4

[65] Arzt, Steven ; Rasthofer, Siegfried ; Fritz, Christian ; Bodden, Eric ; Bartel, Alexandre ; Klein, Jacques ; Le Traon, Yves ; Octeau, Damien ; McDaniel, Patrick: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2014 (PLDI '14). – ISBN 978–1–4503–2784–8, 259–269

[66] Lam, Patric ; Bodden, Eric ; Lhotak, Ondrej ; Hendren, Laurie: The Soot framework for Java program analysis: a retrospective. In: *CETUS Users and Compiler Infrastructure Workshop*, 2011

[67] Grace, Michael C. ; Zhou, Yajin ; Wang, Zhi ; Jiang, Xuxian: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: *NDSS*, 2012

[68] Lu, Long ; Li, Zhichun ; Wu, Zhenyu ; Lee, Wenke ; Jiang, Guofei: CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2012 (CCS '12). – ISBN 978–1–4503–1651–4, 229–240

[69] Chan, Patrick P. ; Hui, Lucas C. ; Yiu, S. M.: DroidChecker: Analyzing Android Applications for Capability Leak. In: *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. New York, NY, USA : ACM, 2012 (WISEC '12). – ISBN 978–1–4503–1265–3, 125–136

[70] Klieber, William ; Flynn, Lori ; Bhosale, Amar ; Jia, Limin ; Bauer, Lujo: Android Taint Flow Analysis for App Sets. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. New York, NY, USA : ACM, 2014 (SOAP '14). – ISBN 978–1–4503–2919–4, 1–6

[71] Fritz, Master-thesis Von C.: FlowDroid : A Precise and Scalable Data Flow Analysis for Android. (2013)

[72] Feng, Yu ; Anand, Saswat ; Dillig, Isil ; Aiken, Alex: Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2014 (FSE 2014). – ISBN 978–1–4503–3056–5, 576–587

[73] Barros, Paulo ; Just, Rene ; Millstein, Suzanne ; Vines, Paul ; Dietl, Werner ; dAmorim, Marcelo ; Ernst, Michael D.: Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Washington, DC, USA : IEEE Computer Society, 2015 (ASE '15). – ISBN 978–1–5090–0025–8, 669–679

[74] Chin, Erika ; Felt, Adrienne P. ; Greenwood, Kate ; Wagner, David: Analyzing inter-application communication in Android. In: *Proceedings of the 9th international conference on*

Mobile systems, applications, and services. New York, NY, USA : ACM, 2011 (MobiSys '11). – ISBN 978–1–4503–0643–0, 239–252

[75] Gibler, Clint ; Crussell, Jonathan ; Erickson, Jeremy ; Chen, Hao: AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: *5th international conference on Trust and Trustworthy Computing (TRUST)*, 2012. – ISBN 978–3–642–30920–5

[76] Jinyung Kim, Kwangkeun Y. Yongho Yoon Y. Yongho Yoon: ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In: *Mobile Security Technologies (MoST)*, 2012

[77] Zhongyang, Yibing ; Xin, Zhi ; Mao, Bing ; Xie, Li: DroidAlarm: An All-sided Static Analysis Tool for Android Privilege-escalation Malware. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. New York, NY, USA : ACM, 2013 (ASIA CCS '13). – ISBN 978–1–4503–1767–2, 353–358

[78] Jeon, Jinseong ; Micinski, Kristopher K. ; Foster, Jeffrey S.: SymDroid: Symbolic Execution for Dalvik Bytecode, 2012

[79] Micinski, Kristopher ; Fetter-Degges, Jonathan ; Jeon, Jinseong ; Foster, Jeffrey S. ; Clarkson, Michael R.: Checking interaction-based declassification policies for android using symbolic execution. In: *European Symposium on Research in Computer Security* Springer, 2015, S. 520–538

[80] Mirzaei, Nariman ; Malek, Sam ; Păsăreanu, Corina S. ; Esfahani, Naeem ; Mahmood, Riyadh: Testing android apps through symbolic execution. In: *ACM SIGSOFT Software Engineering Notes* 37 (2012), Nr. 6, S. 1–5

[81] Grace, Michael ; Zhou, Yajin ; Zhang, Qiang ; Zou, Shihong ; Jiang, Xuxian: RiskRanker: scalable and accurate zero-day android malware detection. In: *10th international conference on Mobile systems, applications, and services (MobiSYS)*, 2012. – ISBN 978–1–4503–1301–8

[82] Yang, Zhemin ; Yang, Min ; Zhang, Yuan ; Gu, Guofei ; Ning, Peng ; Wang, X. S.: AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. New York, NY, USA : ACM, 2013 (CCS '13). – ISBN 978–1–4503–2477–9, 1043–1054

[83] Enck, William ; Gilbert, Peter ; Chun, Byung-Gon ; Cox, Landon P. ; Jung, Jaeyeon ; McDaniel, Patrick ; Sheth, Anmol N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: *9th USENIX conference on Operating systems design and implementation*, 2010

[84] Sun, Mingshen ; Wei, Tao ; Lui, John C.: TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2016 (CCS '16). – ISBN 978–1–4503–4139–4, 331–342

[85] Zhang, Yuan ; Yang, Min ; Xu, Bingquan ; Yang, Zhemin ; Gu, Guofei ; Ning, Peng ; Wang, X. S. ; Zang, Binyu: Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. New York, NY, USA : ACM, 2013 (CCS '13). – ISBN 978–1–4503–2477–9, 611–622

[86] Yan, Lok K. ; Yin, Heng: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA : USENIX, 2012. – ISBN 978–931971–95–9, 569–584

[87] Tam, Kimberly ; Khan, Salahuddin J. ; Fattori, Aristide ; Cavallaro, Lorenzo: CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In: *NDSS*, 2015

[88] Blaesing, T. ; Batyuk, L. ; Schmidt, A.-D. ; Camtepe, S.A. ; Albayrak, S.: An Android Application Sandbox system for suspicious software detection. In: *5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010, S. 55–62

[89] Min, Luo X. ; Cao, Qing H.: Runtime-based behavior dynamic analysis system for android malware detection. In: *Advanced Materials Research* Bd. 756 Trans Tech Publ, 2013, S. 2220–2225

[90] Bierma, Michael ; Gustafson, Eric ; Erickson, Jeremy ; Fritz, David ; Choe, Yung R.: Andlantis: Large-scale Android Dynamic Analysis. In: *CoRR* abs/1410.7751 (2014)

[91] Shabtai, Asaf ; Elovici, Yuval: Applying Behavioral Detection on Android-Based Devices. In: *MOBILWARE*, 2010, S. 235–249

[92] Yang, Yubin ; Wei, Zongtao ; Xu, Yong ; He, Haiwu ; Wang, Wei: DroidWard: An Effective Dynamic Analysis Method for Vetting Android Applications. In: *Cluster Computing* (2016), 1–11. http://dx.doi.org/10.1007/s10586-016-0703-5. – DOI 10.1007/s10586–016–0703–5. – ISSN 1573–7543

[93] VirusTotal: *VirusTotal - online file analysis.* https://www.virustotal.com, accessed 15.03.2017

[94] Schütte, Julian ; Fedler, Rafael ; Titze, Dennis: ConDroid: Targeted Dynamic Analysis of Android Applications. In: *Proceedings IEEE 28th International Conference on Advanced Information Networking and Applications*, 2015 (AINA)

[95] Zhang, Mu ; Yin, Heng: Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security.* New York, NY, USA : ACM, 2014 (ASIA CCS '14). – ISBN 978–1–4503–2800–5, 259–270

[96] Ascia, Giuseppe ; Catania, Vincenzo ; Natale, Raffaele D. ; Fornaia, Andrea ; Mongiovì, Misael ; Monteleone, Salvatore ; Pappalardo, Giuseppe ; Tramontana, Emiliano: Making Android Apps Data-Leak-Safe by Data Flow Analysis and Code Injection. In: Reddy, Sumitra (Hrsg.) ; Gaaloul, Walid (Hrsg.): *WETICE*, IEEE Computer Society, 2016. – ISBN 978–1–5090–1663–1, 205-210

[97] Bugiel, Sven ; Davi, Lucas ; Dmitrienko, Alexandra ; Fischer, Thomas ; Sadeghi, Ahmad-Reza: *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks.* http://www.trust.informatik.tu-darmstadt.de/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2011-0127, 2011

[98] Felt, Adrienne P. ; Wang, Helen J. ; Moshchuk, Alexander ; Hanna, Steven ; Chin, Erika: Permission re-delegation: attacks and defenses. In: *Proceedings of the 20th USENIX conference on Security.* Berkeley, CA, USA : USENIX Association, 2011 (SEC'11), 22–22

[99] Jing, Y. ; Ahn, G. J. ; Zhao, Z. ; Hu, H.: Towards Automated Risk Assessment and Mitigation of Mobile Applications. In: *IEEE Transactions on Dependable and Secure Computing* 12 (2015), Sept, Nr. 5, S. 571–584. – ISSN 1545–5971

[100] Batyuk, L. ; Herpich, M. ; Camtepe, S.A. ; Raddatz, K. ; Schmidt, A. ; Albayrak, S.: Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In: *Malicious and Unwanted Software (MALWARE)*, 2011, S. 66 –72

[101] Poeplau, Sebastian ; Fratantonio, Yanick ; Bianchi, Antonio ; Kruegel, Christopher ; Vigna, Giovanni: Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: *NDSS* Bd. 14, 2014, S. 23–26

[102] Hu, Wenhui ; Octeau, Damien ; Mcdaniel, Patrick ; Liu, Peng: Duet: Library Integrity Verification for Android Applications. In: *WiSec* (2014). ISBN 9781450329729

[103] Stevens, Ryan ; Gibler, Clint ; Crussell, Jon ; Erickson, Jeremy ; Chen, Hao: Investigating User Privacy in Android Ad Libraries. In: *IEEE Mobile Security Technologies (MoST)* (2012)

[104] Open Web Application Security Project: *Mobile Top 10 2016.* https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10, accessed 15.03.2017

[105] Trendmicro: *2016's Mobile Threat Landscape.* http://blog.trendmicro.com/trendlabs-security-intelligence/2016-mobile-threat-landscape/, accessed 18.01.2017

[106] Knuth, Donald E.: *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 1997. – ISBN 0–201–89683–4

[107] Open Web Application Security Project: *OWASP Mobile Top 10 2015 Call for Data*. https://www.owasp.org/images/6/68/MobileTopTen2015-CallForData.pdf, accessed 15.03.2017

[108] Kornblum, Jesse: Identifying almost identical files using context triggered piecewise hashing. In: *Digital Investigation* 3 (2006), S. 91–97. – ISSN 17422876

[109] Fedler, Rafael ; Kulicke, Marcel ; Schütte, Julian: *On the Effectiveness of Malware Protection on Android. An Evaluation of Android Antivirus Apps*. Fraunhofer AISEC TechReport, #apr 2013

[110] ClamAV: *ClamAV - an open source anti-virus engine*. https://www.clamav.net, accessed 15.03.2017

[111] Schmidt, Aubrey-Derrick ; Bye, Rainer ; Schmidt, Hans-Gunther ; Clausen, Jan ; Kiraz, Osman ; Yüksel, Kamer A. ; Camtepe, Seyit A. ; Albayrak, Sahin: Static analysis of executables for collaborative malware detection on android. In: *Proceedings of the 2009 IEEE international conference on Communications*. Piscataway, NJ, USA : IEEE Press, 2009 (ICC'09). – ISBN 978–1–4244–3434–3, 631–635

[112] Yeh, Chao C. ; Lu, Han L. ; Chen, Chun Y. ; Khor, Kee K. ; Huang, Shih K.: Craxdroid: Automatic android system testing by selective symbolic execution. In: *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on* IEEE, 2014, S. 140–148

[113] Allen, Frances E.: Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA : ACM, 1970, 1–19

[114] Grove, David ; DeFouw, Greg ; Dean, Jeffrey ; Chambers, Craig: Call Graph Construction in Object-oriented Languages. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 1997 (OOPSLA '97). – ISBN 0–89791–908–4, 108–124

[115] Machiry, Aravind ; Tahiliani, Rohan ; Naik, Mayur: Dynodroid: An Input Generation System for Android Apps. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA : ACM, 2013 (ESEC/FSE 2013). – ISBN 978–1–4503–2237–9, 224–234

[116] Bodden, Eric ; Sewe, Andreas ; Sinschek, Jan ; Oueslati, Hela ; Mezini, Mira: Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA : ACM, 2011 (ICSE '11). – ISBN 978–1–4503–0445–0, 241–250

[117] Arzt, Steven ; Rasthofer, Siegfried ; Bodden, Eric: *Susi: A tool for the fully automated classification and categorization of android sources and sinks*. Technical Report TUD-CS-2013-0114, EC SPRIDE, 2013

[118] Google Inc.: *monkeyrunner*. https://developer.android.com/studio/test/monkeyrunner/index.html, accessed 15.03.2017

[119] Zeng, Xia ; Li, Dengfeng ; Zheng, Wujie ; Xia, Fan ; Deng, Yuetang ; Lam, Wing ; Yang, Wei ; Xie, Tao: Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case? In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2016 (FSE 2016). – ISBN 978–1–4503–4218–6, 987–992

[120] Mahmood, Riyadh ; Mirzaei, Nariman ; Malek, Sam: EvoDroid: Segmented Evolutionary Testing of Android Apps. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2014 (FSE 2014). – ISBN 978–1–4503–3056–5, 599–609

[121] Azim, Tanzirul ; Neamtiu, Iulian: Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In: *SIGPLAN Not.* 48 (2013), #oct#, Nr. 10, 641–660. http://dx.doi.org/10.1145/2544173.2509549. – DOI 10.1145/2544173.2509549. – ISSN 0362–1340

[122] Bhoraskar, Ravi ; Han, Seungyeop ; Jeon, Jinseong ; Azim, Tanzirul ; Chen, Shuo ; Jung, Jaeyeon ; Nath, Suman ; Wang, Rui ; Wetherall, David: Brahmastra: Driving Apps to Test the Security of Third-Party Components. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, 2014. – ISBN 978–1–931971–15–7, 1021–1036

[123] Weiser, Mark: Program Slicing. In: *Proceedings of the 5th International Conference on Software Engineering*. Piscataway, NJ, USA : IEEE Press, 1981 (ICSE '81). – ISBN 0–89791–146–6, 439–449

[124] Rasthofer, Siegfried ; Arzt, Steven ; Bodden, Eric: A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In: *2014 Network and Distributed System Security Symposium (NDSS)*, 2014

[125] Felt, Adrienne P. ; Chin, Erika ; Hanna, Steve ; Song, Dawn ; Wagner, David: Android permissions demystified. In: *Proceedings of the 18th ACM conference on Computer and communications security*. New York, NY, USA : ACM, 2011 (CCS '11). – ISBN 978–1–4503–0948–6, 627–638

[126] *D3 Data-Driven Documents*. https://d3js.org/, accessed 20.03.2017

[127] Google Inc.: *Android Developer Documentation*. https://developer.android.com/develop/index.html, accessed 20.02.2016

[128] Google Inc.: *Android Open Source Project*. https://source.android.com/, accessed 20.02.2016

[129] Weiss, Konrad: *Android Activity Flow Reconstruction and Visualization*, Technische Universität München, Bachelorthesis, 2016

[130] *vis.js - A dynamic, browser based visualization library.* http://visjs.org/index.html

[131] Tamada, Haruaki ; Nakamura, Masahide ; Monden, Akito ; Matsumoto, Ken-Ichi: *Detecting the Theft of Programs Using Birthmarks*. Japan, 2012

[132] Duan, Ran ; Su, Hsin-Hao: A Scaling Algorithm for Maximum Weight Matching in Bipartite Graphs. In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2012 (SODA '12), 1413–1424

[133] Myles, Ginger ; Collberg, Christian: K-gram Based Software Birthmarks. In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2005 (SAC '05). – ISBN 1–58113–964–0, 314–318

[134] Lim, Hyun-il ; Park, Heewan ; Choi, Seokwoo ; Han, Taisook: A Method for Detecting the Theft of Java Programs Through Analysis of the Control Flow Information. In: *Inf. Softw. Technol.* 51 (2009), Nr. 9, 1338–1350. http://dx.doi.org/10.1016/j.infsof.2009.04.011. – DOI 10.1016/j.infsof.2009.04.011. – ISSN 0950–5849

[135] Hanna, Steve ; Huang, Ling ; Wu, Edward ; Li, Saung ; Chen, Charles ; Song, Dawn: Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In: *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg : Springer-Verlag, 2013 (DIMVA'12). – ISBN 978–3–642–37299–5, 62–81

[136] AppTornado: *AppBrain Android library statistics*. http://www.appbrain.com/stats/libraries/. Version: 15.02.2017

[137] Schwartz, Edward J. ; Avgerinos, Thanassis ; Brumley, David: All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. Washington, DC, USA : IEEE Computer Society, 2010 (SP '10). – ISBN 978–0–7695–4035–1, 317–331

[138] Schütte, Julian ; Titze, Dennis ; Küchler, Andreas: Practical Application-Level Dynamic Taint Analysis of Android Apps. In: *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, S. 17–24

[139] *Signing Your Applications*. http://developer.android.com/tools/publishing/app-signing.html, accessed 23.01.2015