AveDroid: Modeling the Side Effects of the Android SDK

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Michael Appel
Studiengang: M. Sc. IT-Security

Masterarbeit
Thema: AveDroid: Modeling the Side Effects of the Android SDK

Eingereicht: 16. Dezember 2016

1. Gutachten: Prof. Dr. Mira Mezini
2. Gutachten: Prof. Dr. Karim Ali

Betreuer: Dr. Michael Eichberg und Prof. Dr. Karim Ali

Prof. Dr. Mira Mezini
Fachgebiet Softwaretechnik
Fachbereich Informatik
Technische Universität Darmstadt
Hochschulstr. 10
64289 Darmstadt

– Zusammenfassung –

Statische Programmanalyse ist ein großes Forschungsthema mit vielen Herausforderungen. Eine dieser Herausforderungen ist die effiziente Erstellung eines vollständigen und präzisen Aufrufgraphen – eine für die meisten Programmanalysen notwendige Datenstruktur. Um sicherzustellen, dass alle zur Laufzeit möglichen Methodenaufrufbeziehungen (Vollständigkeit) im Aufrufgraphen vorhanden sind wird bei der statischen Programmanalyse meistens ein Aufrufgraph des gesamten Programms – inkl. aller abhängigen Bibliotheken – erzeugt. Bibliotheken sind oft größer als die Applikation selbst, wodurch sich die Komplexität der Erstellung eines Aufrufgraphen so erhöhen kann, dass eine Analyse nicht mehr in angemessener Zeit durchgeführt werden kann. Alternativ kann ein Aufrufgraph berechnet werden, der nur die Applikation abdeckt und die Seiteneffekte der Bibliotheken ignoriert. Da ein solcher Aufrufgraph jedoch nicht vollständig ist, stellt dies keinen idealen Kompromiss dar.

Averroes [18] ist ein Programm zur Transformation von Bibliotheken mit dem Ziel das Verhalten der zugrunde liegenden Bibliothek in Hinblick auf mögliche Methodenaufrufe zu modellieren. Die transformierte Bibliothek kann dann im Kontext von statischer Programmanalyse die Bibliotheken, mit denen das zu analysierende Programm kompiliert wurde, ersetzen. Diese Ersatzbibliotheken sind kleiner als die ursprünglichen Bibliotheken und approximieren alle Seiteneffekte, d.h. die Approximation wird so gewählt, dass möglichst ein vollständiger Aufrufgraph entsteht. Auf diese Weise kann eine effiziente statische Programmanalyse des gesamten Programms durchgeführt werden. Aufrufgraphen, die mit Ersatzbibliotheken erzeugt werden, weisen aufgrund der Approximation eine geringere Präzision auf.

Android birgt spezielle Herausforderungen, die von Averroes nicht berücksichtigt werden. Im Gegensatz zu klassischen Java-Programmen haben Android-Apps keinen zentralen Einstiegspunkt. Dadurch wird die Erzeugung eines Aufrufgraphen erschwert, weil existierende Algorithmen auf einen zentralen Einstiegspunkt angewiesen sind. Des Weiteren sind Apps näher mit der Bibliothek verknüpft, d.h. der Kontrollfluss des Programms wird zu einem großen Teil von der Bibliothek gesteuert. Apps überschreiben eine Teilmenge vordefinierter Methoden der Bibliothek, die von Android bei bestimmten Ereignissen aufgerufen werden. Eine statische Analyse, die das Verhalten der Bibliothek approximiert, muss also diese und weitere Seiteneffekte modellieren.

In dieser Arbeit präsentieren und evaluieren wir AveDroid, eine Erweiterung für Averroes, die spezielle Seiteneffekte von Android-Bibliotheken modelliert, sodass Ersatzbibliotheken auch für Android erzeugt werden können.

Unsere Evaluation zeigt, dass Ersatzbibliotheken bis zu 3-mal weniger Speicherplatz bei einer Taint-Analyse mit FlowDroid [1] benötigen. Allerdings skaliert der Bedarf exponentiell bei steigender Applikationsgröße.

– Abstract –

Static analysis is a wide-ranging research topic with many challenges. One of these challenges is to efficiently build a sound and precise call graph—a required data structure for most static analyses. To ensure soundness, static analyses often choose to build a call graph of the whole program, i.e. a call graph of the application and the libraries that the application depends on. Depending on the size of the library, constructing a call graph of the whole program might be a resource-intensive task for the client analysis. In this case, client analyses have the option to build an application-only call graph which ignores the side effects of the library altogether. This however, leads to unsound results and is therefore not an ideal compromise.

Averroes [2] is a tool that builds Java libraries that can be used as a replacement for the original libraries in the context of static analysis. These placeholder libraries are much smaller than the original library and over-approximate all potential side effects. Smaller libraries can help client analyses to efficiently build a whole-program call graph. Call graphs created with replacement libraries have lower precision due to the over-approximation.

Android introduces domain-specific challenges that are not considered by Averroes. Unlike traditional Java applications, Android apps do not provide a main entry point. This complicates call graph construction because existing algorithms rely on a main entry point. Further, Android apps are tightly coupled with the library, i.e. the control flow of the app is mostly controlled by the library. Android apps override a subset of predefined library methods that are called back by the operating system on certain events. Hence, a static analysis that over-approximates library behavior must model these and other side effects.

In this work, we present and evaluate AveDroid, an extension for Averroes that models side effects of Android libraries, such that replacement libraries can also be created for Android.

Our evaluation shows that replacement libraries require up to 3 times less memory in conjunction with a taint analysis by FlowDroid [1]. However, replacement libraries require exponentially more resources with increasing application sizes.

Acknowledgements

First and foremost, I want to express my gratitude towards my advisor Karim Ali. He taught me many things and patiently answered all my questions. Karim, thank you for everything, I truly appreciate it. Also, I am convinced that you will be a great professor.

I want to thank Michael Eichberg for the feedback, answering my questions and advising this thesis.

Finally, I want to thank Steven Arzt for answering my questions.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
APK	Android Application Package
App	Application
ART	Android Runtime
CFG	Control Flow Graph
CG	Call Graph
CHA	Class Hierarchy Analysis
DVM	Dalvik Virtual Machine
GB	Gigabyte
GPS	Global Positioning System
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
ICC	Inter-Component Communication
IDC	International Data Corporation
MB	Megabyte
NDK	Native Development Kit
OS	Operating System
RTA	Rapid Type Analysis
VM	Virtual Machine
VTA	Variable Type Analysis
XML	Extensible Markup Language

1 Introduction

Android[1] has emerged as the leading operating system among smartphone platforms. According to the International Data Corporation (IDC) [3], Android has a global market share of 87.6% in the second quarter of 2016. In the period of October 2015 to October 2016, the number of available apps on Google Play[2] has grown by approximately 38% and exceeded a total of 2.4 million as of October 2016 [4]. Due to the popularity of Android, it is subject to more malicious apps. The anti-virus vendor G DATA has reported that 1,723,265 new malware samples have been recorded in the first half of 2016 which is an increase of 29% compared to the second half of 2015 [5].

To detect malicious apps, the research community has developed and investigated many program analyses. Most of the analyses are *static* which means that the program being analyzed is not executed. The reasons for this are scalability and to ensure that all execution paths of a given program are traversed. Static analyses are diverse and include techniques to, for example, unfold permission misuse [6], find sensitive data leaks [7, 8, 1], discover vulnerabilities [9, 10] and detect semantic bugs [11]. All of these analyses have to deal with specific challenges. Some of these challenges, such as resolving reflective calls or native methods, originate from Java. With Android, new challenges were introduced. Analyses for Android applications (apps) need to support or translate the Dalvik bytecode, consider the Android Inter-Component Communication (ICC) scheme and callback registration via Extensible Markup Language (XML) files, reason about the lifecycle of components, as well as deal with the absence of a main entry point.

The Dalvik Virtual Machine (DVM) is a virtual machine running on Android devices. Since the Dalvik bytecode is fundamentally different from Java bytecode, static analyses need to provide means to either process Dalvik bytecode directly or translate it to a representation that is already supported by the respective framework. Android defines various components (e.g., *activities* and *services*) that are used by developers to implement apps. Each of these components has a distinct *lifecycle*. The lifecycle of a component refers to certain methods that are called by the Android framework as an app is executed. Thus, a static analysis must reason about the order these methods are invoked by the framework, or else the results will be imprecise. ICC is a mechanism that enables apps to communicate data between components and also between different apps. If a static analysis tracks data flows of Android apps, it needs to consider potential data exchanges between components and apps, otherwise the analysis is potentially unsound. Graphical User interfaces (GUIs) are built with XML resources in Android. Within these resources, a developer can register new callback handlers (e.g., a handler for an `onClick` event). Hence, it is not sufficient to merely analyze the code of an app. Finally, unlike programs written in Java, Android apps do not have a `main` method. The `main` method can be referred to as a *main entry point*. It is called a main entry point because it is the only point in the program where the execution can start. A main entry point makes it easier to build a call graph because call graph construction algorithms have a dedicated starting point from which the call graph for the whole program can be built. Android apps, however, have multiple entry points that are invoked when, e.g., an app is started, restarted, or sent to the background. State of the art analysis frameworks, such as `Soot` [12] and `WALA` [13] require a main entry point to build a precise *call graph*. A naive approach could build an individual call graph for each entry point in an Android app. However, isolated graphs do not model the real call hierarchy of a program and it is not clear how these graphs could be interconnected.

Call graph construction is an integral part of static analysis because analysis results directly depend on the soundness and precision of a call graph. A precise call graph can map any call site to the appropriate receiving object. However, in general that is not possible because the receiver of a call is decided during runtime (*dynamic dispatch*). Because of that, call graph construction algorithms need to approximate possible receiving types. Basic call graph construction algorithms like Class Hierarchy Analysis (CHA) [14] and Rapid Type Analysis (RTA) [15] are fast, but too imprecise for most analyses. More advanced approaches, e.g. Variable Type Analysis (VTA) [16] and SPARK [17] achieve higher precision but are more expensive to compute. The reason is that

[1] `https://www.android.com/`
[2] `https://play.google.com/`

a *points-to analysis* like SPARK keeps track of object flows through the *whole program*. By whole program we mean the application and the libraries that the application uses (in the remainder of this document we use the singular "library" to refer to all libraries an application depends on). Often, the library is larger than the application itself and since the library can instantiate and assign objects, call graph construction takes much time, even for small programs. Works like StubDroid [18] try to mitigate this problem by creating stubs for the library. However, these stubs are tailored for a specific analysis and therefore their benefits are limited to this specific analysis domain.

Ali and Lhoták propose `Averroes` [2], a Java bytecode generator that builds a placeholder library for any given Java application. A placeholder library created by `Averroes` has a flat structure and contains only methods and fields referenced directly and transitively by the application. To compensate for the missing fields and methods, `Averroes` models all potential library actions that could influence call graph construction. This process is described in more detail in Section 2.4. Analyses can use the placeholder library as a replacement for the original library to make the analysis more efficient in terms of runtime and memory consumption.

In this thesis, we lay the groundwork to deal with Android specific challenges such that placeholder libraries for Android can be created with `Averroes`. Further, we aim to compare the precision of resulting call graphs generated with Android libraries shipped with the Android SDK and placeholder libraries created with our novel extension for `Averroes` called `AveDroid`.

We found that `FlowDroid`'s [1] taint analysis is more efficient up to a certain application size in conjunction with placeholder libraries. However, placeholder libraries require exponentially more resources with increasing application size.

1.1 Motivational Example

A whole program analysis considers the code from the application and library. Dependent on the size and structure of a library, the call graph can be fairly large. Even in a simple program as depicted in Listing 1.1, the call graph is complex due to the deep call structure of the Android library. Usually, only a small fraction of the information processed in the library is important for a client analysis.

```
1  public class Button1 extends Activity {
2      private static String imei = null;
3
4      protected void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          setContentView(R.layout.activity_button1);
7
8          TelephonyManager telephonyManager = (TelephonyManager)
               getSystemService(Context.TELEPHONY_SERVICE);
9          imei = telephonyManager.getDeviceId(); //source
10     }
11
12     public void sendMessage(View view){
13         Toast.makeText(this, imei, Toast.LENGTH_LONG).show();
14         SmsManager sms = SmsManager.getDefault();
15         sms.sendTextMessage("+49", null, imei, null, null);  //sink
16     }
17 }
```

Listing 1.1: Example Android application

Table 1.1 shows call graph data for the program in Listing 1.1. In total, there are 43,379 methods in the whole program, while the application defines only 2. Since call graph construction requires a main entry point,

	Whole-program CG	Application-only CG
Total methods	43379	2
Edges	1963	6
Reachable methods	1043	2
Memory usage (in MB)	179	61

Table 1.1: Comparison between the whole-program call graph and the application-only call graph for the application shown in Listing 1.1

we have built a dummy main method with `FlowDroid` (see Section 3.1) to be able to construct the call graph. The data for the whole-program call graph was obtained by running the SPARK algorithm in `Soot`. In comparison, the application-only call graph records only edges that have a connection to the application code. The application-only call graph replaces the library subtree with one node, i.e. all edges that lead or originate from the library are connected to the "library node". We obtained the memory requirements of the application-only call graph by using a replacement library computed with `AveDroid`. To construct an application-only call graph, the library must be analyzed, such that edges from the library to the application can be found. For that reason, client analyses sometimes choose to ignore the effects of the library, including callbacks. In the case of Android, however, this is not an option due to the tight coupling of Android apps with the library, i.e. most of the control flow is controlled by the library by invoking callbacks.

1.2 Contribution and Outline

We present and evaluate `AveDroid`, an extension for `Averroes` that creates replacement libraries for any given Android application. The main benefit of a replacement library created with `AveDroid` is efficiency, i.e. a replacement library is much smaller than the original library which reduces the required time to construct a call graph. The tradeoff for faster call graph construction is decreased precision; replacement libraries must over-approximate library behavior, leading to spurious edges from the library to the application.

The remainder of this thesis is structured as follows. Section 2 provides background information about call graph construction, challenges in the context of static analyses, functionality of `Averroes` and important aspects of the Android platform. Section 3 describes and compares existing approaches for challenges we focus on in this work. Section 4 presents the general workflow of `AveDroid` as well as implementation details and limitations. Section 5 evaluates the scalability and precision of `AveDroid`. Section 6 contains related work, Section 7 indicates aspects for future work and concludes this thesis.

2 Background

In this chapter we provide the necessary information to get a better understanding of existing approaches and our approach; Section 2.1 describes the architecture of the Android OS and the structure of Android applications. Next, we define the term call graph and point out important properties of call graphs (Section 2.2). Moreover, we discuss the concept of partial program analysis. Section 2.3 summarizes challenges for static analyses that target Android apps. We conclude this chapter by giving details about `Averroes` (Section 2.4).

2.1 Android

2.1.1 Architecture

Android is the most predominant operating system in the mobile market and thus it is a popular target for malware. To prevent certain kinds of threats, Android implements various security features in its system architecture. The architecture itself consists of multiple layers, as depicted in Figure 2.1. On the lowermost layer, Android runs a Linux kernel to facilitate access control and process isolation. The Hardware Abstraction Layer (HAL) offers hardware functionality to the upper layers. Native libraries which handle access to databases, the Secure Sockets Layer (SSL), media and other services reside on top of the HAL. Moreover, the Android Runtime (ART) is located on this layer. Although Android applications are mostly written in Java, they are compiled to Dalvik bytecode instead of Java bytecode like traditional Java applications. ART is essentially a newer version of the DVM, i.e. Dalvik bytecode is translated to native instructions on this layer. The Android framework provides different Application Programming Interfaces (APIs) for developers. Functionality of these APIs include, e.g., accessing the device's location, storage and WiFi. Most of this functionality allows to directly access privacy sensitive information (e.g., the location). To protect this information, Android uses a permission system, i.e. each app must request permissions to access sensitive information. The topmost layer comprises all apps. Apps can be pre-installed by Android (e.g., the contacts app) or installed by the user via the Google Play Store or third party stores.

<table>
<tr><td>APPLICATIONS</td><td colspan="2">ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • SMS • MEDIA PLAYER • PHOTO ALBUM • VOICE DIAL</td></tr>
<tr><td>ANDROID FRAMEWORK</td><td colspan="2">CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM</td></tr>
<tr><td colspan="2">NATIVE LIBRARIES</td><td>ANDROID RUNTIME</td></tr>
<tr><td colspan="2">AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT</td><td>CORE LIBRARIES • DALVIK VM</td></tr>
<tr><td>HAL</td><td colspan="2">AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV</td></tr>
<tr><td>LINUX KERNEL</td><td colspan="2">DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT</td></tr>
</table>

Figure 2.1: Android software stack, reproduced from [19]

Android's architecture is constantly evolving and new security features have been added over time. For example, Android employs Security-Enhanced Linux (SELinux) since Android 4.3 [20]. Further, the permission system was reworked with the release of Android 6.0 [21]. Users can now grant permissions to apps during run time which potentially increases awareness whether the permission in question is really required by the app. If the user decides to deny a permission request, the app can still be used with limited functionality. Before that, users had to grant all necessary permissions to an app at installation time, i.e. the app could not be used if the user did not grant all permissions. Despite all these security improvements, one major problem persists; anyone can upload an app to the Google Play Store without any kind of certification process. A developer who desires to publish an app must merely create a self-signed certificate that adds no security. Instead, this certificate is used for authentication purposes, e.g. in case the developer wants to update their app.

2.1.2 Applications

Unlike Java applications, Android apps do not have a main entry point. In order to implement an app, developers extend specific classes defined in the Android framework. These classes, also known as *components*, declare so-called *lifecycle methods* that are called by the framework on certain events. As a user navigates through an app (e.g., a component is started for the first time), Android calls specific subsets of lifecycle methods. Each component has a unique lifecycle. Figure 2.3 exemplarily illustrates the lifecycle of activities. Activities incorporate the most lifecycle methods among all components with a total of seven methods. Note that only certain callback sequences of lifecycle methods (e.g., `onCreate()->onStart()->...`) are possible within one activity. In total, there are four different components:

Activities represent the main GUI elements of Android apps. They are the only components a user can directly interact with.

Services run in the background to perform tasks. Opposed to activities, services keep running when the user switches between apps.

Content Providers can be used to store and access structured data. Sharing data across multiple apps is a common use case for content providers.

Broadcast Receivers are special listeners that handle system-wide events (e.g., incoming phone calls). Broadcasts do not necessarily originate from the Android system, i.e. any app can initiate global events. The reception of a broadcast could trigger, for instance, the start of a service.

Apart from lifecycle methods, developers can register additional callback handlers that handle *notifications*. Notifications are special events triggered by the Android OS to inform apps about, for instance, incoming phone calls or battery shortage. Updates from sensors like the Global Positioning System (GPS) are also distributed through notifications. Lastly, there are GUI-related callback handlers (e.g., an `onClick` handler for a button). GUI-related callback handlers can be defined imperatively, as in Java applications, or declaratively using layout files. Layout files are written in XML and define the user interface of apps. Elements declared in a layout file, for instance buttons, have certain properties that reflect their appearance in the app. Additionally, they can define properties that register callback handlers for the respective element. The handler itself is implemented in the application code.

All components except content providers can communicate asynchronously by using *intents* (see Figure 2.2). Intents are messages that carry an "intended" action, e.g. a component could request another component to take a photo. This feature is also referred to as ICC and it is most commonly used to start activities and services as well as to deliver broadcasts [22]. ICC is not limited to components within one app, i.e. it can also be used to communicate data between different apps. Intents are always passed through the Android framework to determine potential receivers. To reference receivers, a component can specify the target explicitly by stating the name of the component (e.g., `MyActivity.class`) or implicitly by indicating a certain capability. An implicit intent could, for instance, target all components that are capable of displaying a text document to a user.

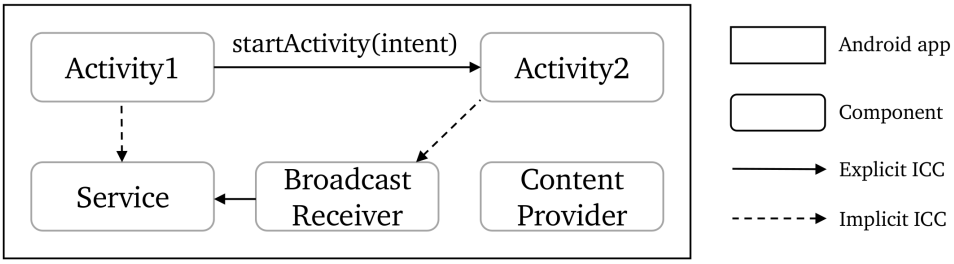

Figure 2.2: Interaction between Android components, reproduced from [23]

In order to compile an app, a developer must have access to the Java class files of the Android library. The Android library is part of the Android System Development Kit (SDK). Since there are many versions of Android, the Android SDK provides a *platforms* folder that contains all versions that have been installed. Each version has its own subfolder with a file called *android.jar* comprising all classes of the respective library. Classes shipped with the Android SDK do not represent a complete implementation of the framework. The Android SDK does not provide a complete implementation (as found on real Android devices) because many modules are dependent on native code. Instead, library methods of the Android SDK contain many stub implementations. Such methods merely throw a *NotImplementedException*. Regarding static analysis, these stub implementations hinder call graph construction due to missing code. Real devices store their library in an optimized file format, making it difficult, but not impossible to extract them [25]. Lastly, there is an open source implementation of the Android framework called Android Open Source Project (AOSP) [26]. This implementation is especially useful for designing static models of the Android framework because it is easily accessible and well-documented. Depending on the client analysis, it might be worthwhile to use the AOSP or a device library instead of a library provided by the Android SDK.

Compiled apps are released in Android Application Packages (APKs). APKs are simple archive files, similar to Java Archive (JAR) files. The contents of an APK are the following:

AndroidManifest.xml. The Android manifest comprises various meta-information about an app, e.g., the Android SDK version the app was compiled with and information about components.

classes.dex. This file contains all compiled application classes in a format that is understandable by the DVM.

resources.arsc. This file comprises binary resources, such as layout files and IDs of resources. IDs are used by the Android OS to access resources during runtime.

lib folder. The lib folder encloses subfolders for platform-specific binary data (e.g., ARM or x86).

res folder. This folder consists of resources (e.g., XML and images) that are not compiled.

assets folder. This folder contains data (e.g., fonts) that can be accessed with the Android class `AssetManager`.

META-INF folder. The META-INF folder contains the certificate created by the developer.

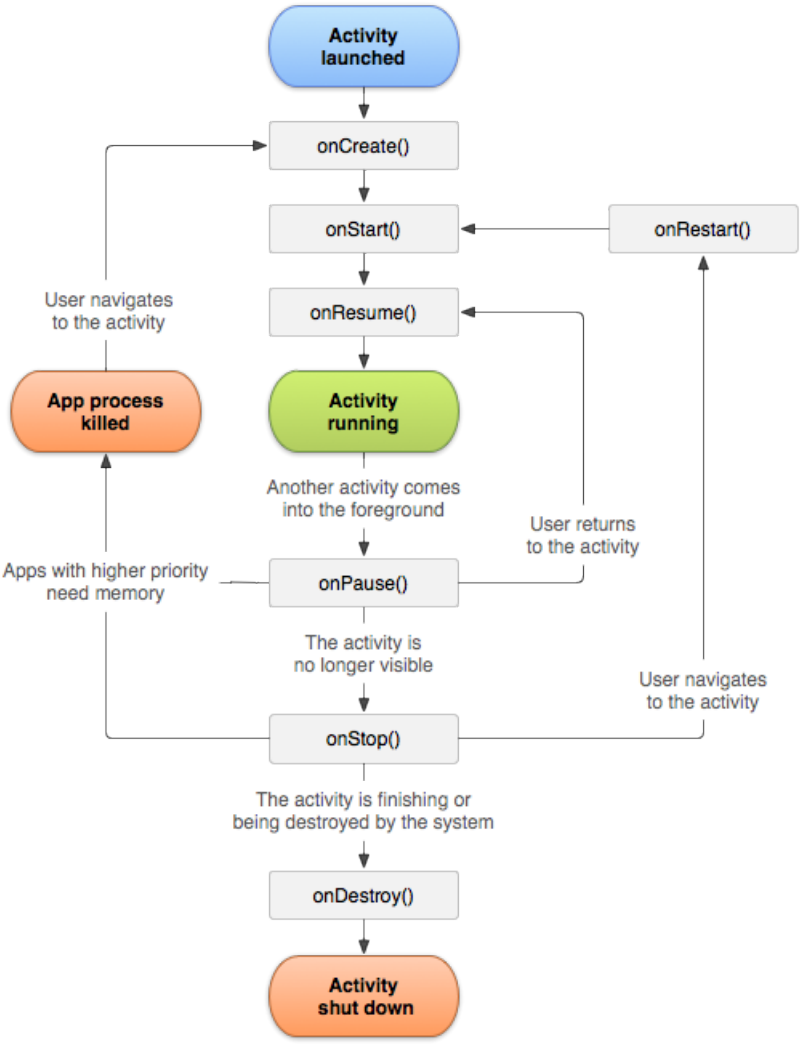

Figure 2.3: Lifecycle of an activity, taken from [24]

2.2 Call Graph Construction

Static call graphs (in the remainder of this document, we use the term "call graph" to refer to static call graphs) are an important data structure for static analyses. They consist of nodes of and edges; nodes symbolize methods in the analyzed program and edges represent the calls between methods. To construct a call graph, algorithms usually start in the main method to look for method invocations. For each newly discovered call site, directed edges to potential targets are added. Iterating this process for all newly discovered methods results in a call graph. Resolving method calls to actual targets is not trivial due to *dynamic dispatch* and some specific Java language features (see Section 2.3).

The most basic approach to construct a call graph is CHA; for each call site encountered in the reachable code, edges to all possible receiving types are added, i.e. all types that extend the static type of the given reference. *Precision* is an important property indicating the quality of a call graph with regard to real program behavior, i.e. a fully-precise call graph comprises only calls that may occur at runtime. Therefore, a call graph built with CHA is very *imprecise* because it contains many edges that are not reachable when the program is executed. More precise results can be achieved with a *points-to* analysis. In addition to reachability, a points-to analysis keeps track of the types, that a reference might point to during runtime. The information of potential runtime types is maintained for each reference and stored in so-called *points-to sets*. These sets aid call graph construction by limiting the number of edges in the call graph because when a call site needs to be resolved to possible receiving types, only the types of the respective points-to set are relevant. This, in turn, decreases the number of reachable methods. On the other hand, reachability also affects the computation of points-to sets. Therefore, unreachable code is not considered by a points-to analysis because it could add new types to the points-to sets resulting in imprecise call graph edges. Computing points-to sets however, is significantly more complex than basic approaches like CHA. Depending on the complexity of the analysed application, a points-to analysis might require too many resources to be a valuable option.

Another important property of call graphs is *soundness*. A call graph is considered sound when it comprises all edges that might be used during runtime. Results of static analyses depend on the soundness and precision of the underlying call graph. For example, assume a client analysis that needs to track data flows through a program to infer certain properties. If this analysis operates on an unsound call graph, it might incorrectly report that a property does not hold (false negative) because a method was not analyzed, although this method might be invoked during runtime. In case the same analysis runs on a sound, though imprecise call graph, results might comprise false positives due to the spurious edges.

In the previous section, we have seen that Android apps do not have a main entry point. That makes it especially challenging to construct a sound and precise call graph. A naive approach would be to build a call graph for each entry point. If these call graphs are built with a points-to analysis, they will miss points-to information from other entrypoints and the Android framework, rendering the resulting call graphs unsound. Constructing call graphs with CHA could work in theory, however these call graphs would be highly imprecise and costly to create because the entire Android framework must be analyzed.

Whole-Program Call Graph Construction. In the motivational example (see Section 1.1), we have seen that constructing a call graph for a whole program is costly, even for small programs. The reason is the complexity of the library; points-to analyses must track potential receiving types through the whole program, otherwise the points-to sets are invalid. Again, although simple approaches like CHA build call graphs faster, usually such call graphs are too imprecise in terms that a client analysis cannot conclude meaningful results. Moreover, even building a whole-program call graph with CHA is expensive because just reading the library dependencies takes a long time [27]. Further, it might be the case that certain libraries are not available for an analysis. All of the previous points raise the need for a possibility to not analyze the whole program, but only certain parts of it.

Partial-Program Call Graph Construction. When an analysis does not have access to the whole program or when analyzing the whole program is too costly, a *partial-program analysis*, i.e. an analysis that uses a *partial call graph*, can be an interesting tradeoff. A partial call graph is a call graph that, in addition to regular nodes, incorporates *summary nodes* for parts of the application that are not analyzed. For instance, a partial call graph can approximate the call hierarchy of the application with regular nodes and the library with summary

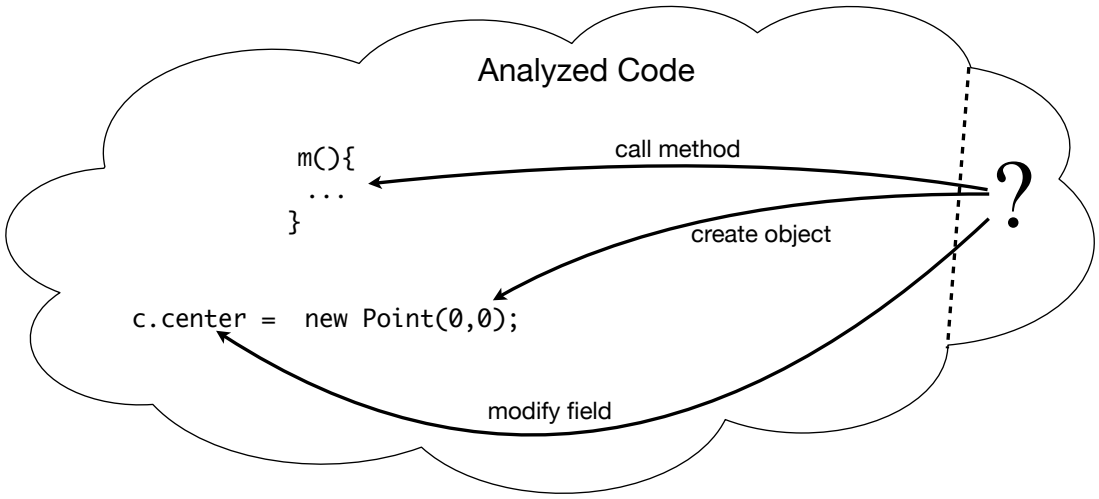

Figure 2.4: Conservative assumptions made by a partial program analysis, taken from [2]

nodes. As the name already suggests, a summary node summarizes program behavior that is relevant for static analyses. Compared to a whole-program call graph, a partial program call graph is much smaller because summary nodes replace entire subtrees of regular nodes. That way, an analysis does not need to analyze deep call chains of methods and instead, only the respective summary node is analyzed. It is important that summary nodes provide an abstraction for all possible side effects the code (e.g., the library) could have, or else the call graph will be unsound. Figure 2.4 illustrates potential side effects of unanalyzed code; it could call any method, create any object as well as load and store any field in the program. All of these side effects have an impact on the computation of points-to sets, hence it is unsound to ignore these effects. The disadvantage of making conservative assumptions about unanalyzed code is that the resulting call graph will be highly imprecise.

In order to gain more precise results with a partial program analysis, more specific assumptions about the unanalyzed parts of the application must be made. For that reason, K. Ali defines the *separate compilation assumption* [27]. The separate compilation assumption states that "the library can be compiled separately without the client application program" [2]. This assumption allows to derive more specific constraints about possible interactions a library could have with an application. For instance, it can be derived that the library cannot extend or implement an application class or interface. That means a partial program analysis does not need to incorporate an abstraction for that behavior in its summary nodes. A complete list of all constraints that follow from the separate compilation assumption is given in [27]. Since Android libraries are JAR files like any other Java library, we investigate whether it can be beneficial to apply the same concept to Android libraries. Nevertheless, due to the challenges presented in Section 2.3, we cannot use `Averroes` out of the box to create placeholder libraries for Android.

2.3 Challenges

In this section we provide specifics about the Java programming language and show why the mentioned features are challenging for static analyses. Since Android applications are implemented in Java, these challenges are relevant for analyses that target Android, too. Moreover, the architecture of Android yields additional challenges which will be discussed here.

2.3.1 Java Inherited Challenges

Reflection is a feature that permits programs to look up and modify its own data structures during runtime. In the general case, reflective method invocations cannot be resolved statically. To see that, we provide an example in Listing 2.3.1. During the execution of line 3, a method with the name `malicious` will be invoked. Since the name of the declaring class is determined dynamically, it is impossible for a static analysis to (precisely) resolve that call. However, the invocation parameters can also be, e.g., hard coded strings, enabling static analyses to reason about possible receiving types. Li Li et al. [28] examined 500 apps and found that 438 apps (87.6%) make use of reflective calls and apps which incorporate reflection, on average, have 138 reflective calls. Moreover, the authors looked for common use cases for reflection in Android apps which we now list.

Generic Functionality. Reflection is sometimes used to implement generic functionality. For example, to instantiate a collection based on user input.

Backward Compatibility. Developers use reflection to detect the Android version that is running on a device. Depending on the version, functionality can be implemented differently or left out.

App Security. Apps that do not implement obfuscation techniques are easier to reverse engineer. Protecting code by loading it dynamically with reflection is a common approach among developers.

Access Internal API. As we have discussed in Section 2.1, developers use an Android library that is different from the libraries used on devices. Libraries in the runtime environment offer more functionality and developers can make use of that by invoking methods which are exclusive to these libraries.

```
1 public static void main(String[] args) {
2     Class c = Class.forName(args[0]);
3     Method m = c.getMethod("malicious");
4     m.invoke(null);
5 }
```

Listing 2.1: Reflective method invocation that is hard to resolve statically

Native code is a feature that is used to implement performance-critical parts of an application or library more efficiently. Moreover, it can help to support legacy code that was composed in other languages than Java. For Android, this feature was adopted in the form of the Native Development Kit (NDK)[1]. Previous studies [29, 30, 31], have shown that the performance gain for Android apps can be significant if the NDK is utilized. Afonso et al. [32] investigated 1,208,476 apps and detected that 267,158 apps (22%) have at least one native method. Further, it is pointed out that apps can make use of native code in other ways, such as including Executable and Linkable Format (`ELF`) data. Considering all means native code can be facilitated by an app, the number of apps that use native code is 446,562 (37%). Native methods are challenging for static analyses because they are written in languages like C, C++ or other languages that are not managed by analysis frameworks. As a result, most analyses treat native code as a black box.

Concurrency or *multi-threading* is a feature to perform multiple actions at the same time. A common use case for multi-threading is to have a thread which controls the GUI and one or more threads to handle functionality. This

[1] `https://developer.android.com/ndk/index.html`

is beneficial because interaction with the GUI will not block operations done in background and vice versa. In the context of Android, the `MessageAPI`[2] is of special interest. The purpose of this API is to send and receive messages across multiple devices, to, e.g., start a new activity on a remote device. Internally, this mechanism uses asynchronous callbacks which need to be considered by a sound analysis. Moreover, Android provides the class `AsyncTask` to perform short background operations and the class `Thread` for more extensive operations. Analyzing programs with multiple threads is challenging because it is hard to outline the interaction between threads and the execution order of statements cannot be predicted.

Dynamic dispatch is a feature that enables applications to decide the target of a method call at runtime. In Java, object references can have a static and a dynamic type. When a method is called with the dynamic dispatch mechanism, the receiver of the call will be the dynamic type of the respective reference. While it is trivial to figure out the static type of a reference, static analysis cannot decide the dynamic type because it is determined during runtime.

2.3.2 Android Specific Challenges

A general challenge for static analyses designed for Android is the evolution of the framework. Framework behavior that is important for analyses might change as new versions are released, i.e. static models that were designed for older versions might become obsolete quickly. Creating static models for Android's library is difficult because the code base is comprehensive and the documentation is incomplete. In fact, the runtime behavior of commonly used features have been changed without any documentation in past versions. For example, Wang et al. [33] have found that there were undocumented changes in the behavior of menu windows and related callbacks.

Dalvik bytecode is a bytecode that was introduced specifically for Android apps. Although Android apps are written in Java, the Dalvik bytecode is fundamentally different from Java bytecode and Android apps that are released on the Google PlayStore are compiled to Dalvik bytecode. As a result, existing analyses for Java bytecode cannot be used for Dalvik bytecode. That forces analyses designed for Android to either convert the Dalvik bytecode first to, e.g., Java bytecode or to provide means to process the Dalvik bytecode directly.

XML resources are used by Android developers to implement the GUI and to define metadata for an app. The metadata of Android apps is encoded in a file named `AndroidManifest.xml` and the contents of this file comprise required permissions, target SDK version, information about components and more. Since this information is valuable for static analyses, it is a necessity for analyses that target Android apps to provide means for parsing XML data. Moreover, developers have the possibility to register callback handlers in XML files. An example for this is given in Listing 2.3.2. This introduces two challenges: (1) the declaring class of the callback (here `android.view.Button`) is potentially only specified in a XML file and never instantiated by the application. An analysis that does not consider this, might conclude that the callback handler is not reachable. (2) The implementation of the specified callback handler (here `submitOnClick()`) is located in an application class file. Thus, a static analysis must find the class that belongs to the respective XML file.

```
1  <RelativeLayout
2  [...]
3      <Button
4          android:id="@+id/submitButton"/>
5          android:onClick="submitOnClick"
6  [...]
7  >
```

Listing 2.2: Callback handler registration in a XML file

[2] `https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi`

As opposed to Java applications, that only have a single entry point, Android apps comprise *multiple entry points*. That is, depending on certain events, such as when an activity is started or brought to the background, the Android library calls distinct methods in the application. In Section 2.2, we showed that multiple entry points complicate call graph construction. In fact, the entry point methods are a subset of the lifecycle methods we have discussed in Section 2.1. The main challenge for static analyses is to reason about the sequence these methods are called by the Android framework. Modeling possible sequences is always an over-approximation because the sequence depends on user actions and operating system's needs. The user could trigger different chains of callbacks by, e.g., starting a new activity or pause the current activity by pressing the back button. A further example is Android closing an app running in the background because of low memory.

ICC is a mechanism in Android that is used for intra- and inter-app communication. The communication is triggered by specific methods (ICC methods). In the previous section we have seen that ICC methods take a parameter of the type `Intent` and that these objects contain information about the targeted components and the desired actions. Similar to lifecycle methods, ICC methods are called by the Android library and thus making it challenging to sequence the invocations of ICC methods. The main challenge in the context of ICC is to find out whether two components are connected by ICC or not. This step requires an analysis to distinguish between explicit and implicit intents. Resolving implicit intents is especially complex because information about the capabilities of a component are located in the Android manifest file, i.e. an analysis must combine information from different resources (components and the Android manifest). Omitting the analysis of ICC methods leads to an incomplete or imprecise call graph.

2.4 Averroes

Since `AveDroid` is based on `Averroes` and hence it shares many concepts with it, we now provide details about the functionality of `Averroes`. In particular, we look at the generation process and contents of replacement libraries. A thorough documentation of `Averroes` can be found in [2].

In Section 2.2 we have pointed out that partial program analyses generally scale better than whole program analyses. A partial program analysis is a tradeoff between scalability and precision; in order to ensure soundness, a partial program analysis must make assumptions about the code that is summarized. If these assumptions are too broad, call graph precision will be too imprecise to be useful. `Averroes` specifically creates summaries for Java libraries. That allows `Averroes` to define more precise assumptions about potential side effects of unanalyzed code (i.e., the library) because of the separate compilation assumption (see Section 2.2).

Whole program analyses need an application and the library that the application depends on as input. The general idea of `Averroes` is to create a replacement library that can be used by a whole program analysis like the original library. By doing so, the analysis effectively becomes a partial program analysis because the replacement library summarizes the entire library behavior in one node (explained later). Constraints that follow from the separate compilation assumption state, for instance, that the library can invoke methods in the application that override a method defined in the library. Without knowledge about the application, `Averroes` could not encode such information in the replacement library. Therefore, `Averroes` creates application-specific replacement libraries. Moreover, `Averroes` tries to keep replacement libraries as efficient as possible. That is, `Averroes` only includes classes, along with their superclasses and superinterfaces, and fields in the replacement library that are directly referenced by the application. In order to achieve that, `Averroes` examines the constant pool of application classes for references to the library. Each library class, method and field that is found must be included in the replacement library. Further, the constant pool of all directly referenced library classes are read to gain knowledge about the class hierarchy. On the contrary to a whole program analysis, `Averroes` does not inspect each class included in the library and it does not analyze any code. By merely analyzing information found in the constant pool, `Averroes` stays very efficient. That is an important requirement because if creating a replacement library was as expensive as a client analysis, it would not be beneficial to create them in the first place.

We now explain the contents of placeholder libraries created with `Averroes`. Placeholder libraries consist of three types of classes: referenced library classes, concrete implementation classes and the `Averroes` library class.

Referenced Library Classes. These classes represent all classes that are directly referenced by the application and their superclasses and superinterfaces. Referenced library classes do not contain the original code. Instead, they only comprise methods and fields that are referenced by the application. The implementation of such methods in the placeholder library follows a template that is illustrated in Figure 2.5. The code is depicted in the language *Jimple* [34], an intermediate representation used by the static analysis framework `Soot`. Identity statements assign each method parameter (and `this` in case of non-static methods) to a local variable and are mandatory in Jimple. In the next step, all parameters are assigned to a special field called `libraryPointsTo`. This field represents the points-to set of the library. That is, it keeps track of all objects that the library could store. Beyond types that are defined in the library itself, the library could store objects of types that are defined in the application. This might occur when the application passes an object to the library by invoking a library method. For that reason, parameter assignments are included in all methods present in referenced library classes. Next, the method `doItAll()` is called. The `doItall()` method is the single summary node of the library, implementing all potential side effects the library could have (explained shortly). Finally, if the method has a return type, the `libraryPointsTo` field is casted to that type and returned. This is done to model that the method could return any object that is present in the library and compatible to the return type.

Concrete Implementation Classes. The application code can also invoke methods of classes that are not directly referenced by it. For example, when the library returns an interface to the application, the concrete implementation is not known. For the purpose of constructing a call graph, the whole program analysis must know about at least one type that implements the interface. If neither the application implements the interface nor does the application reference a concrete implementation type in the library, `Averroes` forges a special type that implements the interface. Methods of such types are implemented like methods in directly referenced library classes.

`Averroes` **Library Class.** The `Averroes` library class serves as the summary node for the library. It consists of the `libraryPointTo` field and the `doItAll()` method. To model all potential library behaviors, the `doItAll()` method contains statements to simulate callbacks, object instantiations and exception handling. In detail, the `doItAll()` method has the following contents:

1. **Class instantiation:** In order to model that the library can create objects of any concrete library class, the `doItAll()` method contains two statements for each concrete library class. These statements instantiate the given class and invoke an accessible constructor on the appropriate object. Moreover, the library can instantiate application classes through reflection (see Section 2.3). `Averroes` also creates the aforementioned statements for an application class, if it has knowledge about a reflective instantiation of that class. To learn about classes that are instantiated reflectively, `Averroes` looks for string constants in the constant pool of library classes. Also, it is possible to provide reflection facts as input to `Averroes`. These facts can be created with, e.g., `TamiFlex` [35].

2. **Callbacks:** The separate compilation assumption states that the library could call any application method that overrides a library method due to dynamic dispatch. Therefore, `Averroes` includes method invocations to all methods that override a library method in the `doItAll()` method. Potential receivers and argument types are determined with the `libraryPointsTo` field. Moreover, if the callback method returns a reference type, the `doItAll()` method will assign the returned object to the `libraryPointsTo` field because it could be a type from the application that is unknown in the library.

3. **Array element writes:** All objects that are known to the library could be stored in an array. To model this, the `libraryPointsTo` field is casted to an array of type `java.lang.Object` and the `libraryPointsTo` field is assigned to the first element of the array.

4. **Exception handling:** Finally, the library could throw any Exception objects that it is aware of. To simulate this, the `libraryPointsTo` field is casted to `java.lang.Throwable` and an exception is triggered with a `throw` statement.

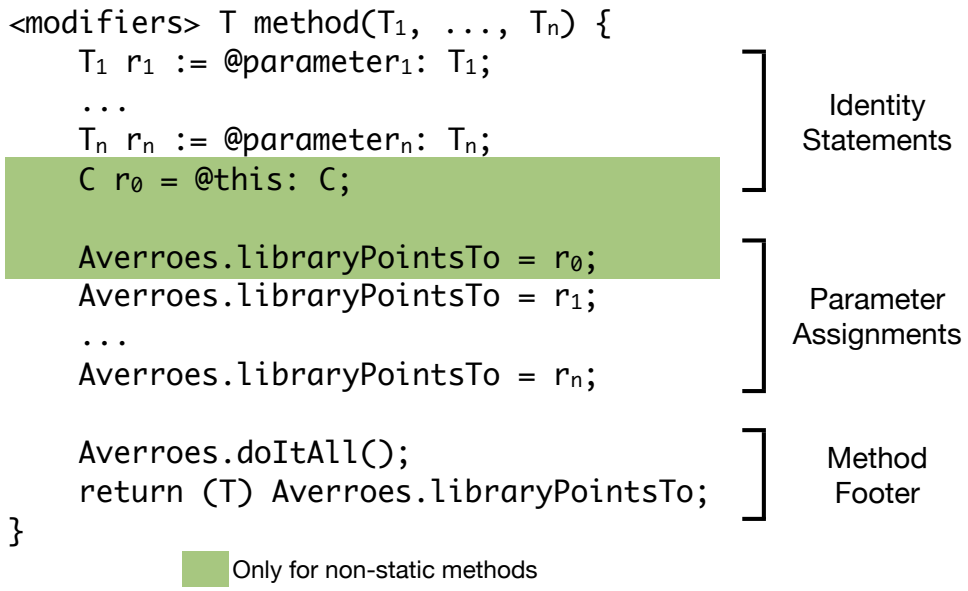

Figure 2.5: Jimple method body of library methods, taken from [2]

All of the aforementioned statements are inserted in the `doItAll()` method without control-flow constraints. The authors based their decision on the fact that all popular analysis frameworks for Java construct flow-insensitive call graphs. This is done to safely over-approximate that statements could be executed in arbitrary order and an arbitrary amount of times.

3 Existing Approaches

In this chapter, we summarize existing approaches that tackle some of the challenges presented in Section 2.3. First, we discuss FlowDroid, a static taint analysis tool for Android apps (Section 3.1). There are other works, for instance, a work by Li Li et. al [8] which cope with Android specific challenges in a similar fashion, i.e. by simulating the lifecycle of components. Since FlowDroid is the first work that incorporates a precise simulation approach and because it is well adopted by the static analysis community, we use it representatively to explain the simulation concept. We then describe StubDroid, a program that creates summaries specifically for taint analyses (Section 3.2). In Section 3.3, we give details about Droidel, an approach that enables static analyses to analyse the behavior of Android libraries rather than abstracting the functionality away. Further, we explain the concepts behind *Window Transition Graphs* (WTGs) created with GATOR (Section 3.4).

3.1 FlowDroid

FlowDroid is a state of the art, precise context, flow, field, object-sensitive and lifecycle-aware static analysis tool for Android apps. The analysis perfomed by FlowDroid is called *taint analysis*. A taint analysis tries to identify sensitive data flows in apps that are potentially malicious or violate a policy. For example, suppose a malicious app accesses a privacy sensitive field (e.g., the phone number) and sends it to a device controlled by the attacker via SMS. Assuming the malicious app does not take any precautions to conceal this action from static analysis, such as loading code dynamically with reflection, FlowDroid will report this data flow. Often, sensitive data flows are not implemented intentionally. This can happen, for instance, when developers incorporate advertisement libraries in their apps [36]. That is, if the library is not transparent in terms of sensitive data usage, the developer will unknowingly accomodate code in their app that leaks sensitive information.

Taint analysis is based on the concept of *sources* and *sinks*. To find sensitive data flows, the meaning of sensitive data and how it is identified must be defined first. Sources are statements that unconditionally cause a variable to become *tainted*, meaning that the variable carries sensitive data. Since Android protects sensitive data with permissions, it makes sense to define all API invocations that require permissions as sources. Nevertheless, it is not trivial to define an accurate set of sources in a sense that each potential privacy breach is found. For instance, previous work has shown that public Android resources (i.e., resources that are accessible without, or with commonly used permissions) can be used to infer privacy sensitive data [37, 38]. Sinks are predefined statements, such as an invocation statement of the API method to send a SMS, that will trigger a taint analysis to report a violation in case a tainted value is involved. FlowDroid provides a sample list of sources and sinks to perform a taint analysis. The analysis itself looks for data flows that lead from sources to sinks and reports all pairs of sources and sinks with at least one connection.

An example for a malicious data flow is depicted in Listing 1.1. In the user-defined method `onCreate()`, the device's IMEI is stored in the field `imei` (line 9). Since the IMEI is a privacy sensitive value because it uniquely identifies a mobile phone, FlowDroid defines statements that store the return value of `<TelephonyManager: String getDeviceId()>` as sources. A sink is located in the method `sendMessage()`. This method first requests the `SmsManager` from the Android system and then sends the IMEI via SMS to a chosen phone number (line 15). Note that the method `sendMessage()` is never explicitly called in the application code. Instead, it is a callback handler invoked by the library when a button in the applicatio nis clicked. Also, the callback handler was not registered in the application code but in a layout XML file. Therefore, in order to find a flow from the sink to the source, FlowDroid must recognize that the method `sendMessage()` is potentially called during runtime.

We now explain how FlowDroid deals with Android specific challenges. In order to be compatible with existing call graph construction algorithms, FlowDroid generates a *dummy main method* which serves as a main entry point. FlowDroid creates application-specific dummy main methods to precisely encode the application-defined methods that are called back by the Android framework. The method body of the dummy

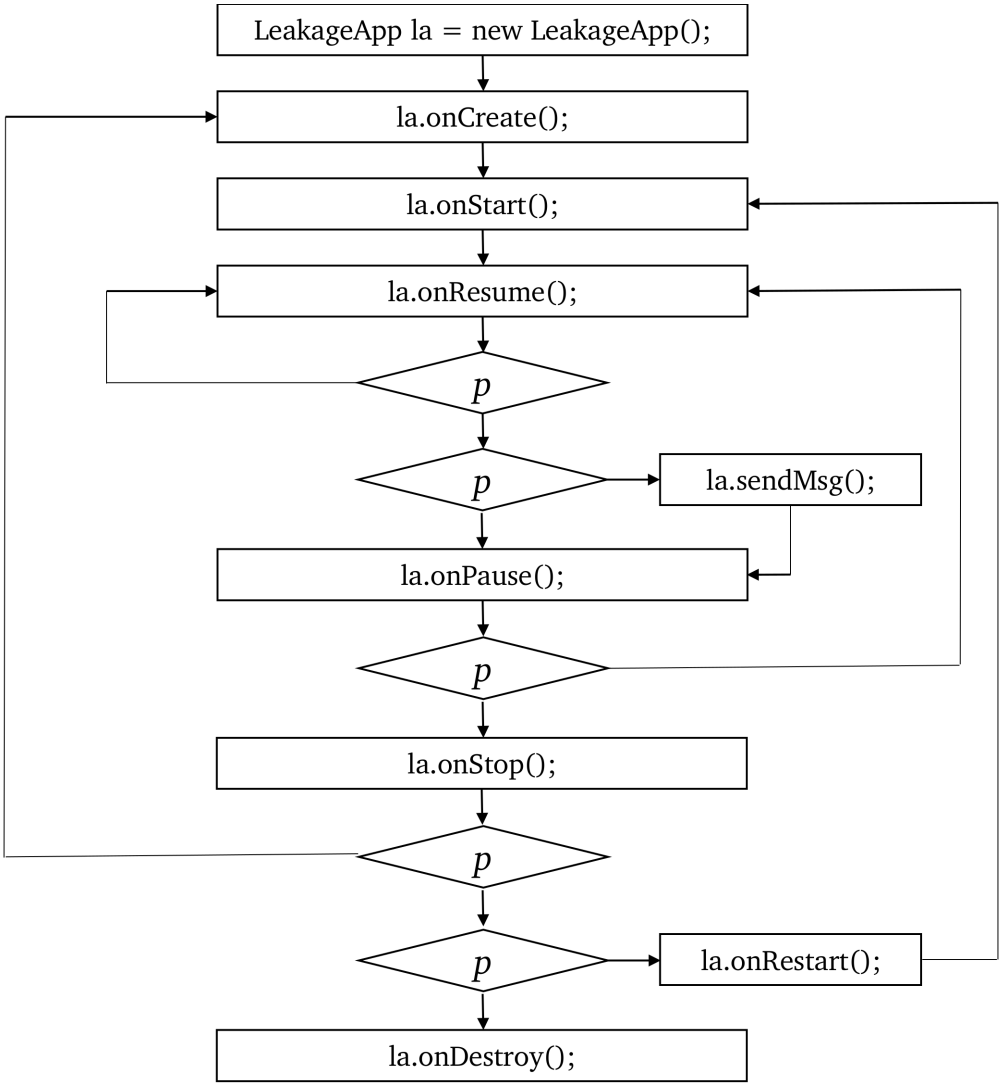

Figure 3.1: Control flow of the dummy main method, reproduced from [1]

main method simulates the lifecycles of all components (see Section 2.1) that are present in the app. Components are found by parsing the Android manifest. Figure 3.1 exemplary illustrates the Control Flow Graph (CFG) of a dummy main method. For the sake of simplicity, the app consists of a single component, namely an activity called *LeakageApp*. `FlowDroid` analyzes the code of all components to learn about implemented lifecycle methods. As can be seen from the provided CFG, `FlowDroid` explicitly invokes all lifecycle methods that an activity defines (see Section 2.1) because the class *LeakageApp* implements all lifecycle methods (the code itself is not shown). To achieve higher precision, `FlowDroid` sequences lifecycle callbacks in accordance with the Android documentation (see Figure 2.3). For that matter, `FlowDroid` inserts opaque if statements (depicted by the rhombuses with *p* in Figure 3.1) in the dummy main method. Opaque means that the condition cannot be evaluated statically. These statements ensure that a flow-sensitive static analysis can consider all possible method orderings.

Beyond lifecycle methods, `FlowDroid` models imperatively and declaratively defined callbacks in the dummy main method. Callback methods that handle GUI-interaction and notifications are always executed in the context of the component that registered the callback [1]. More precisely, `FlowDroid` assumes that callback handlers can only be invoked when the hosting component is running, i.e. between the methods `onResume()` and `onPause()`. We could confirm this assumption for recent Android versions (API versions 23 and 24) with manual testing (see Section 4.2). Therefore, `FlowDroid` only adds call sites for callback handlers that

are associated with the respective component to the dummy main method. In the given example (Figure 3.1), the method `sendMsg()` is registered in *LeakageApp*. In order to find all callback handlers and their hosting components, `FlowDroid` distinguishes between imperatively and declaratively registrations. Details about associating XML-based callback registrations with a component are given in Section 4.3 as we use the same approach. Imperatively registered callback handlers are added to the dummy main method by iteratively constructing call graphs until a fixed point is reached; first, `FlowDroid` constructs a dummy main method that merely comprises entry points that are initially reachable, i.e. all lifecycle methods that are defined by the application. Callback registrations are detected by comparing method signatures with a hardcoded list. If new registrations are found, `FlowDroid` will include calls to the appropriate handlers in the dummy main method and repeat the process. Since callback handlers can register new callbacks, this step is repeated until no new registrations are found.

After `FlowDroid` has finished building the dummy main method, it initiates the taint analysis. The details of the taint analysis are not of further importance for this work, but it is intuitive that `FlowDroid` is able to find data flows from sources to sinks with the dummy main method. For example, assume the code from Listing 1.1. Traversing the CFG of the dummy main method, `FlowDroid` will find that first the method `onCreate()` is called. From analyzing `onCreate()`, `FlowDroid` can infer that the field `imei` is tainted. Since the application code consists of only two methods, the dummy main method contains only two call sites. Therefore, the connection from source to sink is found when `FlowDroid` analyzes the callback handler `sendMessage()`.

3.2 StubDroid

Performing a taint analysis on a model that merely incorporates application code is insufficient. A sound analysis must also track information flows through the library. For better understanding, assume a scenario where the source is not leaked directly, but through a taint propagation due to library code. For example, if the content of the field `imei` in Listing 1.1 is stored in a library field (e.g., with `Exception ex = new Exception(imei)`) after it has been tainted, the client analysis must propagate the taint to the appropriate field or associate the taint with all fields of `ex`. In order to leak the tainted library field, the app could call, e.g., `sendTextMessage("+49..", null, ex.getMessage(), null, null)`. Technically, the original example already involves a tainted library field because the class `String` is part of the library, but we assume that `String` is a primitive type for the sake of argument.

Taint analyses have different options to cope with library code. A simple approach is to treat library code like application code, i.e. to analyse the entire code for each target application [39]. This approach achieves high precision, though it is not scaleable. We have seen earlier that the Android codebase is extensive, hence an analysis based on this approach will spend most of the time analysing library code. However, most of the library code is irrelevant for the client analysis which renders this approach highly inefficient. A second option is to approximate the library behavior by applying general rules for library methods [40]. For example, one of these rules could state that whenever a library method with at least one tainted parameter is called, the returned value is assumed to be tainted as well. In comparison to the first approach, this one loses precision due to the generality of the applied rules.

In its original form, `FlowDroid` makes use of another approach. The idea is to provide manually created summaries for a subset of framework methods. `FlowDroid` refers to these summaries as *taint wrappers*. Before `FlowDroid` analyzes a method for taint propagation, it checks whether a taint wrapper is available for the given method. In case there is a summary available, `FlowDroid` performs the taint propagation based on the summary rather than analyzing the whole method. Taint wrappers are especially efficient for methods and types that are either referenced frequently or complex to analyse because of deep call chains. Another advantage of taint wrappers is that they can be used even if a part of the library that needs to be analyzed is missing. However, taint wrappers also have drawbacks. We mentioned in Section 2.3 that one of the challenges specifically for analyzing Android apps is the evolution of the Android framework. The main drawback of taint wrappers is that they require maintenance, potentially for each newly released version. While some framework constructs, such as the lifecycle of activities, remained unchanged over the course of all Android versions, other parts and especially the concrete implementation of classes is changed and extended frequently. Since these summaries

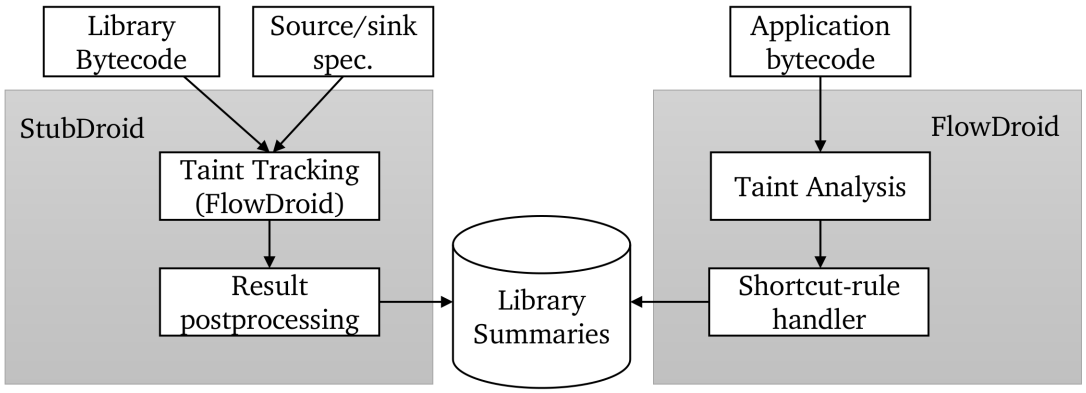

Figure 3.2: Architecture of StubDroid, reproduced from [18]

are created through manual inspection, this is an unsatisfying solution. In comparison, the approach of defining general rules to summarize library methods is not required to provide this kind of customization for different Android versions because it does not depend on the concrete implementation.

A more sophisticated approach for modeling library behavior, specifically for taint analysis, is `StubDroid`. An overview of `StubDroid`'s architecture is illustrated in Figure 3.2. Summaries created by `StubDroid` are not application-specific. Therefore, `StubDroid` requires only the desired library along with the source and sink specification, i.e. a file that lists what method signatures are considered to be sources, and what signatures are considered to be sinks. Note that the input library does not have to be an Android library. `StubDroid` can create summaries for all Java libraries. It assumes that an app only accesses public library resources, hence it exclusively computes summaries for public methods. Further, `StubDroid` cannot make any assumptions about the state of library fields or call sequences of library methods due to missing knowledge about application code. For that reason, `StubDroid` analyzes methods with all potential parameter and field configurations. `StubDroid` outputs one file per summary in the XML format and each file represents one library class. This is beneficial because the summaries are independent of specific client analyses and can be loaded on demand.

In the given example (Figure 3.2), `FlowDroid` is the client taint analysis. Since `StubDroid` provides summaries for each public method, the client analysis does not need the library anymore. Instead, when the analysis encounters a call site to the library, it can plug in the summary by loading the appropriate XML file and perform the taint propagation more efficiently. We have seen earlier that `FlowDroid` incorporates a proprietary data structure (taint wrappers) to facilitate shortcuts. In order to support summaries created with `StubDroid`, `FlowDroid` simply translates the XML summaries to taint wrappers on demand. In comparison with manually crafted summaries, `StubDroid` achieves similar precision while speeding up analysis times because manually crafted summaries are limited to a subset of library methods. The apps that the authors of `StubDroid` examined, were analyzed 15% faster on average with `FlowDroid` using `StubDroid`'s summaries compared to hand-crafted summaries. A downside of `StubDroid` is the required computation time for creating summaries, though this merely must be done once per library. The authors measured summary generation times ranging from 35 to 960 seconds for individual *classes*.

An advantage of all summary-based models is that, in some cases, they *enable* an analysis because analyzing the application together with the library may fail due to exceeding resource demand. Creating summaries with `StubDroid` is not very memory-intensive. `StubDroid` avoids that by analyzing each library method separately, hence the memory consumption is low in comparison to the taint analysis.

3.3 Droidel

Droidel [41] is a more general approach than FlowDroid's simulation approach, i.e. Droidel does not target specific client analyses, such as a taint analysis. The authors of Droidel motivate their work by providing an example app (see Listing 3.3) that is modeled incorrectly by client analyses that incorporate the simulation approach. The app first instantiates a new AsyncTask (line 5). As the name suggests, objects of the class AsyncTask perform tasks in the background. Next, a dialog is created which, e.g., could display to the user that the app is currently logging in (line 6). In order to provide the user the option to abort this operation, the app registers a new callback handler that cancels the task when the user pushes the hardware back button (lines 7–11).

Existing simulation-based approaches, such as FlowDroid, create a dummy main method that instantiates at least two objects for the given example; one that represents the activity LoginActivity o1 and one that represents the class that implements the interface OnCancelListener. Since the latter class is anonymous, we refer to it as LoginActivity.OnCancelListener o2. We have seen in Section 3.1 that FlowDroid adds method invocations to all lifecycle methods that are overridden by the app. Therefore, the dummy main correctly calls o1.onCreate(). Regarding the OnCancelListener, the authors of Droidel claim that simulation-based approaches merely call o2.onCancel() and nothing else. In fact, we could confirm this behavior for the (as the time of this writing) latest development branch of FlowDroid.

In this case, the framework model created by FlowDroid is unsound. To see this, assume an object-sensitive points-to analysis on the previously described dummy main method. After the call o1.onCreate() has been examined, the points-to analysis will find that points-to(o1.mAuthTask) = {MyAsyncTask} and points-to(o2.mAuthTask) = ∅. The points-to set of o2.mAuthTask is empty because of the object-sensitivity, i.e. the method is analyzed in the context of o1. Since there are no other remaining call sites in the dummy main method besides o2.onCancel(), the points-to analysis concludes that o2.mAuthTask can only hold null. This is unsound because when the program is compiled, LoginActivity.OnCancelListener cannot be instantiated without its enclosing type LoginActivity, i.e. the anonymous class captures the reference of o1.mAuthTask which is different from null when o2 is created.

Unsoundness in the call graph can pollute the results of the underlying client analysis. If, for instance, the abort operation propagates tainted values or leaks a tainted value to a sink, FlowDroid will miss these data flows which renders the analysis unsound, too. Note that this particular bug could be easily fixed, though the authors of Droidel claim that they have discovered many more problems. However, they provide only the one concrete example which we have just presented.

```
1  class LoginActivity extends Activity {
2      AsyncTask mAuthTask = null;
3
4      @Override void onCreate() {
5          mAuthTask = new MyAsyncTask(...);
6          AlertDialog d = ProgressDialog.create(...);
7          OnCancelListener l = new OnCancelListener() {
8          @Override void onCancel() {
9              mAuthTask.cancel();
10         }};
11         d.setOnCancelListener(l);
12     }
13 }
```

Listing 3.1: Sample app that is modeled incorrectly by FlowDroid's dummy main method, reproduced from [41]

```
1  class AppStubs implements DroidelStubs {
2      Activity getActivity(String cls) {
3          if (cls == "ActivityA") {
4              return new ActivityA();
5          } else if (cls == "ActivityB") {
6              return new ActivityB();
7          } else { return new Activity(); }
8      }
9      // Further method implementations
10     [...]
11 }
```

Listing 3.2: Extract of a DroidelStubs implementation, reproduced from [41]

According to the authors of `Droidel`, the root cause for the unsoundness in simulation-based models is the modeling of the framework; the framework is too complex to derive sound and precise models from it. Details of the library that are abstracted away by static library models might be of importance for call graph construction. Less precise call graph generation algorithms like CHA mask the particular problem from Listing 3.3, but they might also be too imprecise to get reasonable results.

`Droidel` is a different approach that transforms Android apps into a form such that analysis tools for Java applications can process Android apps. Contrary to the simulation approach, `Droidel` tries to model as little framework behavior as possible. The key idea of `Droidel` is to replace any library code that is hard to handle for static analyses with more explicit code (*explicating*). For example, Android instantiates application-defined components (e.g. activities) with reflection because it is not aware of the class names at compile time. With the knowledge of application code however, many reflective method invocations in the library can be replaced with more explicit statements.

In detail, the Android framework instantiates activities with:

```
Activity act = (Activity) clazz.newInstance();
```

`Droidel` replaces this statement with:

```
Activity act = droidelStubs.getActivity(clazz.getName());
```

`droidelStubs` is a reference to a special class created by `Droidel`. The implementation of this class is dependent on the application. Similiar to `FlowDroid`, `Droidel` parses the application manifest to learn about components that are implemented by the app. With that information, `Droidel` can implement the `DroidelStubs` interface. A sample implementation of the `DroidelStubs` interface is depicted in Listing 3.3. In this case, the application consists of the two activities `ActivityA` and `ActivityB`. Dependent on the specified parameter `cls`, the `DroidelStubs` implementation returns a new instance of one of the both application-defined activities or an instance of the super class `Activity` if `cls` is unknown. Therefore, a precise points-to analysis can conclude the runtime type of `act` by analyzing the code of the library.

Apart from instantiating activities, the Android framework incorporates reflective calls for more functionality. For example, Android uses reflection to retrieve UI elements that were defined in layout files. We have seen earlier that callback handlers can be registered in layout files. This functionality is also implemented with reflective calls in the library. All of these behaviors are explicated in the `DroidelStubs` implementation.

Moreover, `Droidel` modifies the parameter list of the library method `ActivityThread.main()` which is the entry point in the Android framework to start an activity. The parameter list is extended such that it accepts an implementation of the `DroidelStubs` interface. In order to analyze an app with `Droidel`, the application-specific `DroidelStubs` implementation must be created first. Further, `Droidel` provides a

main entry point for clients analyses. The entry point does not model any library behavior like simulation-based approaches. Instead, the main entry point merely calls `Activitythread.main()` with the appropriate `DroidelStubs` implementation.

The client analysis can use `Droidel`'s main entry point for call graph construction. By doing so, the client analysis effectively traces the entire Android framework for callbacks to the application. Compared to simulation-based approaches, `Droidel` achieves enhanced soundness but the call graph construction takes longer. All of the examined applications (7 total) missed less call graph edges using `Droidel` compared to a call graph generated with `FlowDroid`. `FlowDroid` missed 30% of the edges on average while `Droidel` missed 6% on average. The missing edges were identified by comparing the call graphs to a dynamic instrumentation. Building call graphs with `Droidel` is slower by a factor that ranges from 2 to 11, compared to `FlowDroid`.

3.4 GATOR

`GATOR`[1] is a static program analysis toolkit for Android. The analysis performed by `GATOR` is very comprehensive which is why we only want to provide a high-level view in this work. A thorough documentation of `GATOR` can be found in [42].

The development of the toolkit is motivated by the fact that existing solutions (e.g., `FlowDroid`) do not provide a precise model of potential event and callback sequences as well as potential window transitions that may occur during runtime. In order to get a more precise handling of these behaviors, the authors of `GATOR` propose a model that focuses on the control-flow of GUI elements in Android applications. The key idea is to provide a data structure—a *Window Transtition Graph* (WTG)—that represents the GUI semantics during runtime. By "windows", the authors refer to several GUI elements. All subclasses of `android.app.Activity`, dialogs and menus are considered to be windows. Dialogs are interactive popups that are usually closed after the user provided some input and menus are embedded into activities and consist of multiple items that the user can click on. This abstraction is made because all of these elements usually contain multiple widgets (e.g., buttons) that can trigger a change on the *window stack*, i.e. clicking these widgets can cause that a new window is opened or an existing window is closed.

The window stack is a generalization of the *back stack* [43] that keeps track of activities that are currently alive. Compared to the back stack, the window stack does not merely incorporate activities but all types of windows. `GATOR` uses this data structure to model the effects of different callback handlers. When a new window is opened, it is pushed onto the window stack and whenever a window is closed, an element is popped from the window stack.

Modeling the effects of callback handlers on the window stack is a complicated task because a single handler can trigger multiple actions on the window stack. For example, it can happen that multiple active windows are closed by pressing the hardware back button. Moreover, depending on preceding events, the same callback handler could have different effects on the window stack. In order to precisely model all potential behaviors, `GATOR` performs a context-sensitive analysis. This is required because GUI-based callback handlers receive a widget as parameter. This parameter is used to determine the widget that has been clicked by the user. A context-insensitive analysis would conclude that, regardless of the parameter, all statements in the callback handler might be executed. Since these statements might lead to different chains of callbacks, a context-insensitive analysis is too imprecise.

In addition to analyzing widget related events, `GATOR` examines the effects of *default events*. These events comprise screen rotation, pressing the hardware back, power, menu, and home button. In essence, `GATOR` focuses on analyzing all event handlers that might have an effect on the window stack. Callback sequences of other components, such as services are not modeled by `GATOR`.

The output of `GATOR` is the aforementioned WTG. Client analyses can use this data structure to get precise control-flow information of GUI-related elements. An example of a WTG is depicted in Figure 3.3. In this case,

[1] `http://web.cse.ohio-state.edu/presto/software/gator/`

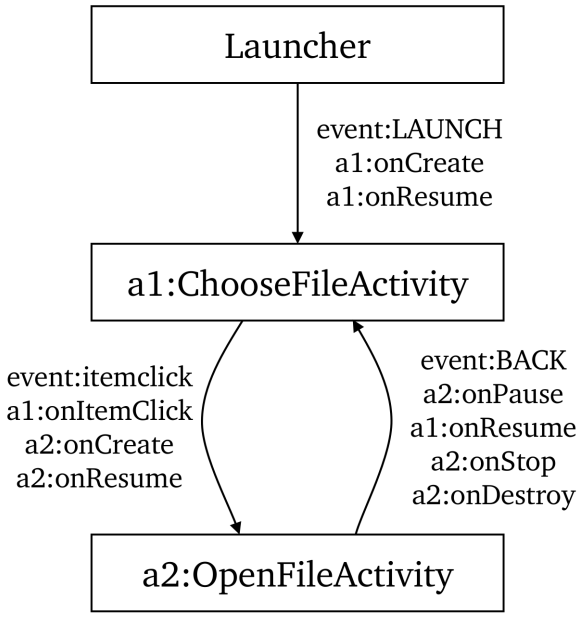

Figure 3.3: Window transition graph, reproduced from [33]

the application consists of two activities `ChooseFileActivity` and `OpenFileActivity`. When the application is launched, the Android framework invokes the lifecycle methods `onCreate()` and `onResume()` of `ChooseFileActivity`. `ChooseFileActivity` has a menu that triggers a window transition to `OpenFileActivity` when an item is clicked. Since menus and dialogues are also windows, `GATOR` allocates distinct nodes for dialogues and windows in the WTG. For simplicity, this is not shown in the example. When the user presses the hardware back button while `OpenFileActivity` is active, a non-trivial interleaving of callbacks occurs; first, the lifecycle method `onPause()` of `OpenFileActivity` is called by the Android framework. Then a lifecycle method of `ChooseFileActivity` is invoked (`onResume()`), followed by further method invocations on the `OpenFileActivity` object. While the first transition (`ChooseFileActivity -> OpenFileActivity`) and the corresponding sequence of callbacks can be derived from `FlowDroid`'s dummy main method, the second one cannot. `FlowDroid` assumes that lifecycle methods of `ChooseFileActivity` will not be invoked until the lifecycle of `OpenFileActivity` has ended (i.e., after `onDestroy()` was called).

A work by S. Yang et al. about WTGs evaluated 20 apps and measured the time to create a WTG for each app [42]. For 19 of these apps, the WTG could be constructed in under 20 seconds. In one case (`FBReader`) the generation took 2086 seconds due to a limitation in their analysis.

There exist some client analyses that incorporate WTGs. The first example is an analysis that tries to spot energy drain effects caused by Android apps [44]. Unnecessary energy drain can happen when developers misjudge the control-flow of their app. For example, if an energy-intensive component (e.g., GPS or WiFi) is still accessed even though it is not needed anymore. Another client analysis focuses on the detection of expensive operations performed on the GUI-thread [45]. Expensive operations on the GUI-thread can compromise the responsiveness of an app. In order to provide a good user experience, these operations should be performed on another thread.

3.5 Summary

We have seen three different approaches that model specific behavior of the Android library and one approach that creates summaries for a specific client analysis. An important observation is that none of the approaches is fully comprehensive. Given the complexity of the Android framework, most approaches try to focus on modeling the parts of the library that are most important for the intended client analysis. `Droidel` is the most general approach and does not focus on a specific client analysis. In order to stay general, `Droidel` provides an entry point and a custom Android library that explicates a subset of reflection uses, such that clients analyses can create a call graph by analyzing the library. There are multiple drawbacks with that approach; the customized library was created with manual inspection, i.e. new framework versions require manual efforts to be compatible with `Droidel`. Further, `Droidel` does not explicate all uses of reflection in the Android library. The biggest drawback, however, is that an analysis of the whole program is expensive and slows down client analyses. `FlowDroid` creates a static model that simulates the lifecycles of components and other callbacks from the library. The design is based on manual investigation of the framework, i.e. the model might require adjustments for newer versions of Android. Also, we have seen that this model yields soundness issues. `StubDroid` provides library summaries that generally improve analysis times. These summaries are specifically designed for client taint analyses. Creating these summaries is costly, though it is a one-time effort per library version. `GATOR` focuses on modeling precisely the control-flow of GUIs. Currently, there exist only a few client analyses that use WTGs. Although WTGs achieve good precision, it is questionable whether this data structure is useful for a broader scope of client analyses because the abstraction lacks important information, such as callback modeling of components other than activities.

4 Implementation

This chapter is divided into three parts: First, we specify the requirements of AveDroid. We then provide a high-level view of the workflow and conclude this chapter by giving details about the concrete implementation.

4.1 Requirements

Currently, Averroes can create replacement libraries for Java applications. We extend Averroes such that it is capable of creating replacement libraries for Android applications. In Section 2.3 we presented challenges for static analyses. In this work, we focus on dealing with a subset of Android specific challenges (see Section 2.3). ICC is not in the scope of this work. As a consequence, we do not model potential side effects of intra-app communication. That also means that we model single applications only, i.e. flows between different applications are not considered in this work.

To summarize, AveDroid has the following requirements:

(i) Users should be able to input an Android application (APK) along with the necessary libraries that the application depends on.

(ii) AveDroid should be able to process Dalvik bytecode to access the app's resources.

(iii) AveDroid should be able to parse XML files to find metadata and XML-based callbacks.

(iv) AveDroid should provide a main entry point for static analyses.

(v) AveDroid should model the lifecycle of Android components.

(vi) AveDroid should create a replacement library that has the same structure as a replacement library created with Averroes (see Section 2.4). That means, the replacement library should only comprise directly referenced methods and fields, a `doItAll()` method that models all possible library behaviors, etc. Addtionally it should also facilitate functionality which models the properties listed in (iii)–(v).

(vii) AveDroid validates and outputs the replacement library for the user. The user can then use the library to perform static analyses for the respective app.

4.2 Design

Figure 4.1 illustrates the general workflow of AveDroid creating a placeholder library. AveDroid creates placeholder libraries that are tailored for specific apps. Hence, it is required that a user provides an app along with the Android library the app was compiled with as well as any external libraries the app depends on. We create application-specific libraries because we aim for an efficient placeholder library, i.e., so we can examine what parts of the original library are used by the app and remove any unnecessary parts.

In the first step, we ensure that the resources we need to build the placeholder library can be accessed easily. The Android library and external libraries are jar archives containing *class* files and the Android application is an APK archive containing a *dex* file and other resources (see Section 2.1). All resources of interest (except for the `AndroidManifest.xml`) are passed to AveDroid in the form of bytecode, i.e. if a resource is required, we transform that resource to a representation that is easier accessible. After the transformation, all classes can

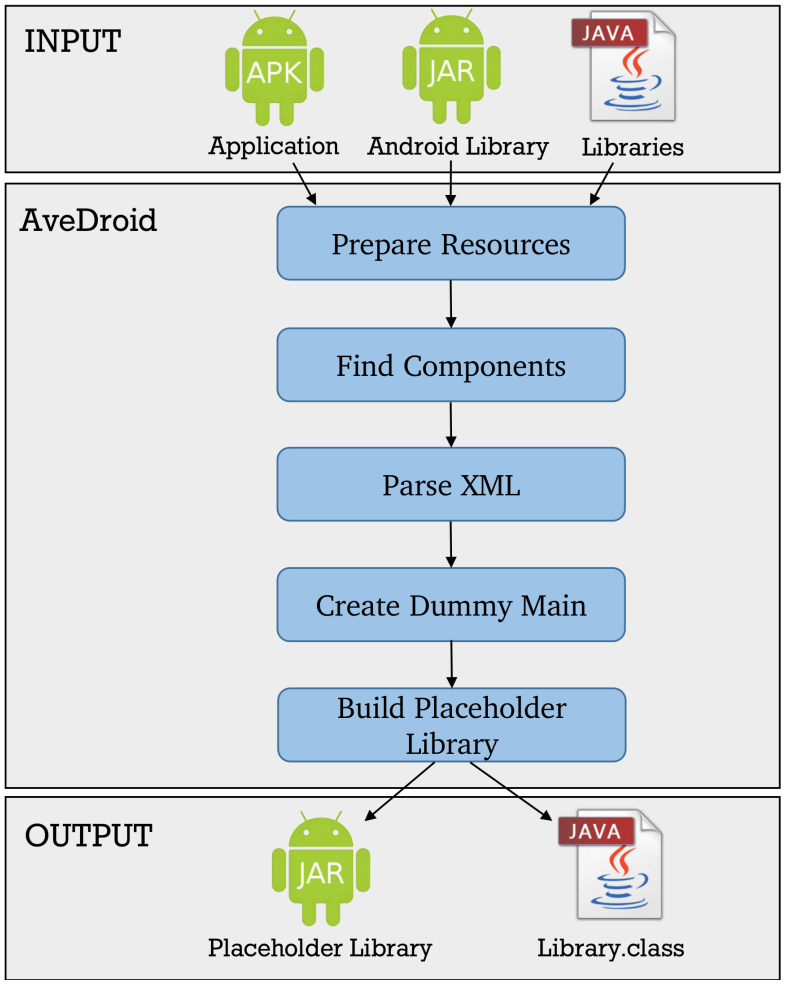

Figure 4.1: AveDroid design

be accessed interchangeably regardless of the source file. Next, we collect all components and their lifecycle methods that are implemented by the application. As we pointed out in Section 2.1, apps are composed of different components (*activities*, *services*, *broadcast receivers* and *content providers*). Each of these components has a distinct lifecycle that we simulate in the replacement library. The idea behind simulating the lifecycle is to have a method (*dummy main method*) that instantiates all components present in the application and invokes all lifecycle methods with appropriate control-flow constraints (see Section 3.1). Since components and their lifecycles are being started dependent on the user interacting with the GUI, we cannot determine their order statically. The control-flow constraints ensure that we safely over-approximate all possible orders [1]. Note that the order of individual lifecycles is encoded precisely by this approach [1, 18] because the sequence in which the Android framework invokes lifecycle methods of individual components is well known (see Section 2.1). Alternatively, we could have used Droidel's explicating approach (see Section 3.3). In fact, the explicating approach is closer to a sound call graph (i.e., call graphs lack less edges in general) than the simulation approach [41], however the explicating approach does not fit our requirements. Similar to `Averroes`, `AveDroid` aims for efficiency, i.e. we want to keep the overhead of the replacement library as low as possible. On the other hand, the authors of Droidel propose the contrary: model as little framework behavior as possible. That is, a client using Droidel analyzes the library to reason about the control-flow of an app. While this approach is more natural because it eliminates assumptions about the library that can lead to unsound results, it also hinders the idea of abstracting away library functionality to improve analyses in terms of runtime and memory consumption.

In addition to lifecycle methods, we also model other callbacks in the dummy main method. As we pointed out in Section 2.1, Android facilitates two ways for an app to register callback handlers; imperatively, by calling the appropriate system method and declaratively, by using XML files. Regarding XML-based callback handlers, we need to make sure that (1) the class declaring the callback registration method (e.g. `android.widget.Button`) will be present in the replacement library, and (2) the matching activity is identified correctly (see Section 2.3). Callback handler registrations in XML files might reference classes that are never referenced by the application code itself. However, such classes might be of importance for a client analysis and since we aim to include only library classes in the replacement library that are directly referenced by the application (explained later), we provide special treatment for such classes (see Section 4.3). In order to find callback handlers that are registered in a XML file, we need to find a mapping between activities and their respective layout XML file. XML files do not contain any information that associate the XML file with an activity class. Thus, we exercise an algorithm of `FlowDroid` that analyzes activities for a certain method invocation (see Section 4.3).

With all this information gathered we can now create the dummy main method. Our dummy main method has a similar structure to `FlowDroid`'s dummy main method (see Section 3.1). The difference is that we use a more general and lightweight, albeit less precise approach to identify and model imperatively defined callback handlers. As we have seen in Section 3.1, `FlowDroid` iteratively creates call graphs from lifecycle methods to find callback handler registrations. Registrations themselves are identified by a manually created list that contains the signatures of all listener interfaces. The advantage of this approach is precision; the list of signatures ensures that irrelevant methods (e.g. `<java.lang.Object hashCode()>`) are not identified as callbacks and the resulting call graphs allow to map the callback handler registrations to an activity. The latter information is useful since callbacks can only be invoked in the context of the activity that registered the callback handler [1]. We do not adopt this concept because constructing multiple call graphs is costly and the list of callbacks needs to be maintained as new Android versions are released. Instead, we use the same approach as `Averroes` (see Section 2.4), i.e. we identify any method that overrides or implements a library method as a potential callback target. The tradeoff is that our dummy main method models imperatively defined callback handlers not as precise as `FlowDroid`'s dummy main method. In our approach, these methods can be called by the library at any time since we invoke these handlers in the `doItAllMethod()`.

We now discuss *when* callbacks can be invoked by the Android framework. The authors of `FlowDroid` make the assumption that both GUI-related as well as other callback handlers (such as a location update handler) will only be invoked by the framework when the activity is running (i.e. between the execution of the lifecycle methods `onResume()` and `onPause()`) that defined these handlers [1]. This assumption seems plausible for GUI-related callback handlers because the user can only interact with an activity when it is visible and ready for input. That behavior is also documented by Android [24]. On the other hand, there exists no documentation (to our best knowledge) for other callback handlers. We undertook manual efforts to get indications whether this assumption is (still) meaningful. Running an emulated device with Android 7.0 (the latest version as of the writing of this document), we triggered events during the execution of the lifecycle methods `onCreate()`, `onStart()` and `onResume()`. According to our observations, all callback handlers are always invoked between `onResume()` and `onPause()`, giving us reason to believe that the assumption is reasonable. Thus, we add method invocations to all callback handlers between `onResume()` and `onPause()`. The order of the callbacks cannot be statically predicted [1], hence we over-approximate all possible orders by using appropriate control-flow constraints.

Constructing the placeholder library involves the same steps as we have seen in Section 2.4. The most important change is the inclusion of the dummy main method in the placeholder library. That is, the placeholder library now provides an entry point for analysis clients in addition to the library summary node. Apart from the placeholder library, the user also gets a file named `Library.class` as output. Technically, this file could be part of the placeholder library but to provide compatibility with `WALA` we chose to exclude this file from the placeholder library. This class is also the place where we encode the dummy main method. Other contents of that file have been discussed in Section 2.4.

Analyzing An App With AveDroid. Static analyses can use the replacement library in two different ways, as illustrated by Figure 4.2. Firstly, an analysis can choose to create its own model of the library and use the placeholder library like a library provided by the Android SDK. An example for this is `FlowDroid`; `FlowDroid` can use the replacement library like a standard Android library without any modifications to the

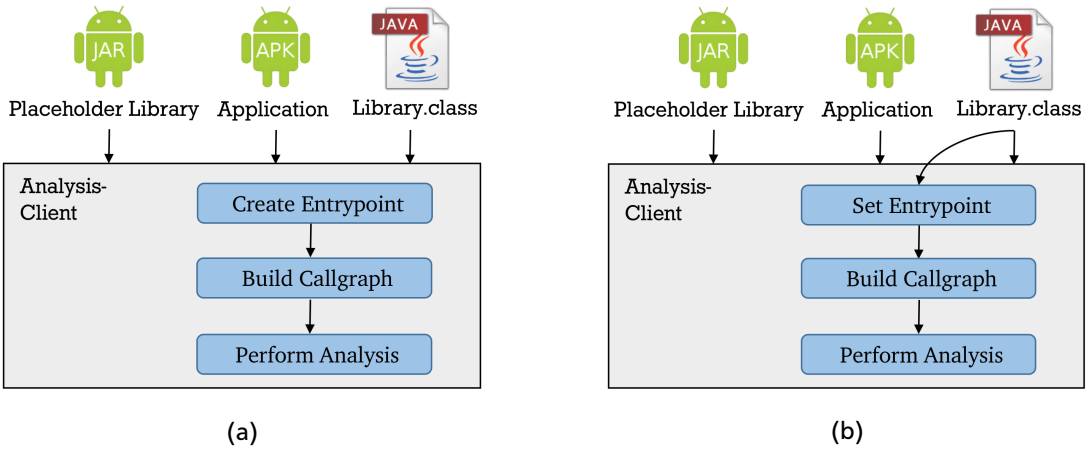

Figure 4.2: Two types of methods to analyze an Android app with AveDroid: (a) the client analysis provides a custom entry point and uses the replacement library to get a summary of the Android library. (b) The client analysis sets the dummy main method in the replacement library as the main entry point and performs the analysis.

taint analysis. However, in order to construct a call graph with the placeholder library, the analysis client must ensure that the library points-to field is initialized. The file `Library.class` provides a static initializer which initializes the library points-to field, i.e. the analysis client can set the static initializer of `Library.class` as an additional entry point. Secondly, an analysis client can use the dummy main method provided by `AveDroid` to initiate call graph construction. In order to do so, the analysis sets the dummy main method inside the class `Library.class` as an entry point. Similarly to the other method, the library points-to field can be initialized with the static initializer of `Library.class`. With the resulting call graph, the analysis client can proceed with its default steps.

Limitations. Since our static model of the Android library is very close to `FlowDroid`'s approach, our approach has similar limitations. We have seen in Section 3 that the simulation model is not sound. For example, our model does not encode the interleaving of callback sequences of multiple components (see Section 3.4). In Section 3.3 we have seen that call graphs derived from the dummy main method miss edges. However, it is not clear to what extend the soundness of call graphs (derived from the dummy main method) can be improved by fixing implementation bugs. Although our dummy main method indirectly encodes the most basic use of ICC, i.e. to start new components, we do not provide means to model data flows via ICC. Further, we do not provide special means to resolve reflective calls. For Java applications, tools like `TamiFlex` [35] can be used to resolve certain kinds of reflection usage, however on the Android platform these tools are not currently supported. Moreover, our abstraction assumes that the code is executed synchronously, i.e. we do not model multi-threading.

4.3 Implementation Details

Our implementation is made on top of the static analysis framework `Soot`. In the first step, we prepare all resources such that `Soot` can access all classes. In order to load the class files from the APK file, we exercise `Dexpler` [46] which is a submodule of `Soot`. In addition to loading classes, `Dexpler` can also transform classes to Jimple.

When all resources are loaded to `Soot`, we parse different files to prepare the construction of the dummy main method. Many parts of this code—including the construction of the dummy main method—are taken from the implementation of `FlowDroid` which is open source and available on `GitHub`[1]. In detail, we first parse the

[1] `https://github.com/secure-software-engineering/soot-infoflow-android`

application manifest file to find components that are implemented by the application. Further, we parse the `resources.arsc` file to learn about mappings between resources and identifiers. At runtime, all Android and application resources (e.g., layout files) have a distinct identifier.

In order to associate layout files with activities, we look for the call `setContentView()` in the Jimple code of activities. The compiled application code invokes this method with an integer which is the identifier of a layout file. For example, the Jimple code of an activity could contain the following statement:

```
virtualinvoke $r0.<package.MainActivity: void setContentView(int)>(42)
```

In this case, the activity `MainActivity` is associated with the resource identifier 42. From parsing the `resources.arsc` file we know what resource is mapped to the identifier 42, therefore we now can associate this activity with a layout file. As mentioned earlier, we use this association to locate and map callback handlers that are registered in layout files. Assume that a layout file registers a callback handler `myHandler()`. In order to find this handler, we look in the Jimple code of the associated activity for a method with the subsignature `void myHandler(android.view.View)`.

The second problem we need to solve regarding layout files is that we need to include the classes that define the callback registration methods (e.g. `android.view.Button`) in the placeholder library. Since layout files only contain the class names without package names (e.g. `Button`), we try to find these class in predefined package names, such as `android.view` and `android.widget`. Each class that is found will be loaded to `Soot` and included in the placeholder library.

We then construct the dummy main method. All components of the application are instantiated and for each component the lifecycle is simulated. By analyzing the Jimple code of implemented components, we can match predefined method signatures to identify lifecycle methods. The lifecycle methods are then invoked on the appropriate objects we allocate in the dummy main method. Moreover, we invoke all callback handlers of components that we have associated with the respective component. As mentioned before, the lifecycles are simulated in accordance with the official Android documentation, i.e. conditional statements and loops are added to encode control-flow information.

As we have seen in Section 2.4, `Averroes` processes the constant pool of class files to find method and field references from the application to the library. Android applications have a different constant pool organization, i.e. we had to adjust this algorithm. Figure 4.3 depicts the constant pools of a Java class and a Dalvik dex file. While each Java class file has a distinct constant pool, Android applications have a *shared* constant pool. The benefit of a shared constant pool is that references must be defined only once and can be accessed from any class, hence a shared constant pool requires less space. We observed that shared constant pools of dex files also contain references from library classes. Since we cannot distinguish between references that are exclusively used in application classes, our algorithm also includes methods and fields in the replacement library that are not directly referenced by the application. In order to access the shared constant pool, we use `dexlib2`, a plugin of `smali` [47].

In Section 2.4 we have seen that `Averroes` encodes callbacks from the library to the application in a single summary node (`doItAll()` method). `Averroes` includes callbacks to all application methods that override a library method or implement a library interface. This property also applies to the lifecycle methods of components. Since we invoke all lifecycle methods in the dummy main method, we do not include callbacks to these methods in the `doItAll()` method.

In order to verify the integrity of replacement libraries, we exercise `BCEL` [48], a Java bytecode analysis tool. The dummy main method contains references to classes of the Android application. Without the referenced Java class files, `BCEL` cannot verify the code because `BCEL` does not support class loading from dex files. Therefore, our solution requires the user to provide a jar file with all application classes. Jar files that contain all classes of an Android app can be obtained by a converter, such as `dex2jar`[2].

[2] `https://github.com/pxb1988/dex2jar`

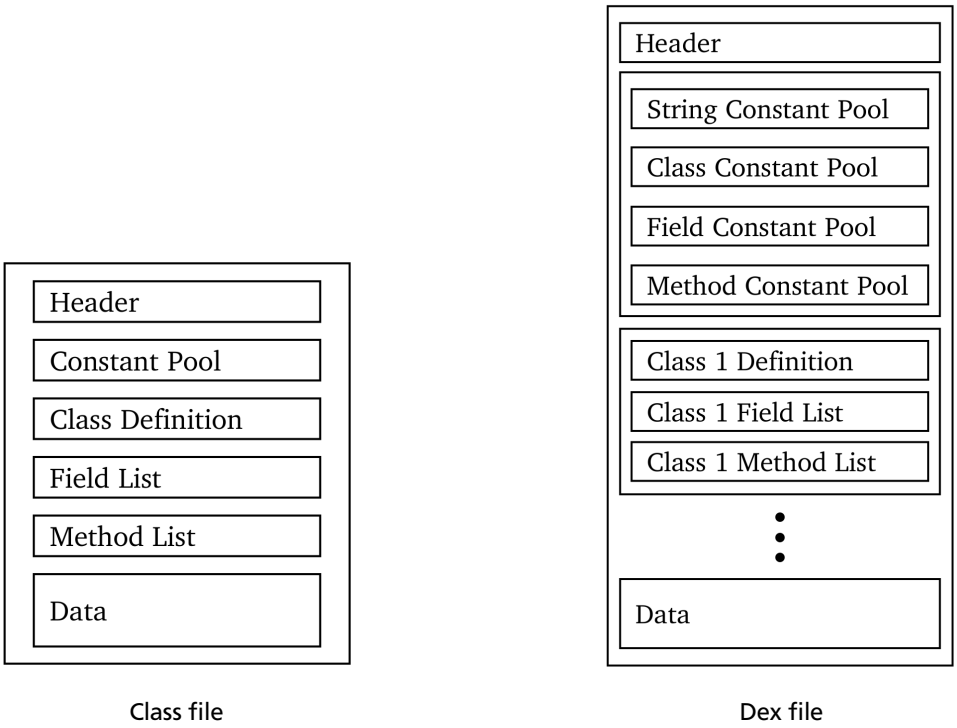

Figure 4.3: Dalvik dex and Java class file, reproduced from [49]

5 Evaluation

In this chapter, we present the results of the experiments we have run. On the one hand, we have created a tool that summarizes library behavior with regards to call graph construction, i.e. the summaries do not target specific client analyses. Therefore, we are interested in comparing the precision of call graphs created with replacement libraries and original libraries. On the other hand, our static model of the Android library is close to FlowDroid's model, and hence we are also interested in comparing the results of FlowDroid running with original Android libraries and replacement libraries. To summarize, we want to answer the following questions in this chapter:

- Can FlowDroid find data flows in conjunction with replacement libraries?
- How do replacement libraries scale in the context of taint analysis with FlowDroid?
- How do call graphs created with AveDroid compare to call graphs created with FlowDroid?

5.1 DroidBench

DroidBench is a by-product of FlowDroid that was developed to evaluate the precision and soundness of client taint analyses for Android. There exist other benchmark suites like SecuriBench [50] that consist of applications with information leaks. However, these benchmark suites were designed for Java client analyses and therefore do not incorporate test cases that consider Android specific challenges, such as XML-based callback registrations or component lifecycles. On the other hand, DroidBench was specifically developed for Android taint analyses and comprises test cases for Java language specifics, such as taint propagations through arrays and fields as well as test cases for Android specifics, such as taint propagation through XML-based callback handlers and ICC. The entire DroidBench test suite is open-source and available on GitHub[1].

Test Setup. In order to perform the tests, we first computed the placeholder library for each tested application. We then ran FlowDroid on the test cases—one time with a standard Android library and one time with the appropriate placeholder libraries. In particular, we utilized an Android implementation from a real device[2] because Android SDK libraries are not ideal for static analysis (see Section 2.1). Since we do not provide special handling for ICC information flows, we merely test the applications that propagate flows through callback methods. All apps that are part of the callbacks tests were compiled with Android 4.2 (API version 17) except one which was compiled with Android 4.4 (API version 19). We left the latter app out because we do not have a real device implementation of Android 4.4.

Table 5.1 depicts the results of the tests we ran. The first column lists the names of the test cases. Under *Android Library* (*AL*), we report the maximum memory usage of the taint analysis and show whether the test case passes (✓) or fails (✗) using a standard Android library. The third column, *Placeholder Libraries* (*PL*), shows the same data with respect to placeholder libraries. In general, the test apps are rather small and contain less than a dozen application methods. For example, the first app, LocationLeak1, implements two activity lifecycle methods and one listener interface to get location updates from the GPS. A benefit of these small applications is that we know the ground truth, i.e. in real-world applications it is not always clear what information flows should be detected or it requires a lot of manual effort to find all leaks.

The results show that placeholder libraries from AveDroid are suited for finding tainted data flows with FlowDroid. In all test cases we receive the same results in terms of sensitive information flows independent of the library we used. One test case (Unregister1) fails with both libraries because FlowDroid does not

[1] https://github.com/secure-software-engineering/DroidBench
[2] https://github.com/Sable/android-platforms

Test Case	AL (Memory, Pass/Fail)	PL (Memory, Pass/Fail)
LocationLeak1	168 MB ✓	114 MB ✓
LocationLeak2	191 MB ✓	86 MB ✓
LocationLeak3	213 MB ✓	70 MB ✓
MultiHandlers1	191 MB ✓	142 MB ✓
AnonymousClass1	133 MB ✓	79 MB ✓
TestOrdering1	182 MB ✓	98 MB ✓
Unregister1	239 MB ✗	140 MB ✗
Button1	179 MB ✓	61 MB ✓
Button2	194 MB ✓	106 MB ✓
Button3	283 MB ✓	176 MB ✓
Button4	170 MB ✓	61 MB ✓
MethodOverride1	177 MB ✓	60 MB ✓
RegisterGlobal1	192 MB ✓	77 MB ✓
RegisterGlobal2	167 MB ✓	154 MB ✓

Table 5.1: DroidBench test results

detect "unregistered" callback handlers that are not reachable during runtime. Moreover, the presented data shows that placeholder libraries perform efficiently with small applications. All taint analyses require up to 3 times less memory using placeholder libraries compared to the Android device library. Regardless of the library, we observed that memory consumption can vary by up to 40% on different runs. Further, we observed that, on average, execution times are 20% faster using replacement libraries.

5.2 Scalability

In order to test the scalability of replacement libraries, we compared results of `FlowDroid` running on real-world applications. Similar to the other tests, we computed the replacement libraries for the applications and ran `FlowDroid` with a standard Android library and appropriate replacement libraries. We ran all experiments on a Debian[3] virtual machine (VM). The VM was set up on an Intel Core i5-2500k and 8 GB RAM. We used a 64-bit JVM with a maximum heap size of 3.5 GB.

When we were looking for suitable applications, we faced some difficulties. Since Android 4.2 is the latest Android version we could find a real device implementation for, we were looking only for applications that were compiled with this version. However, most app stores do not list the Android version a specific app was compiled with, but rather the minimum version that is required to install the app on a device. Newer Android versions usually contain classes that are not available in older versions. `FlowDroid` can compute results in these cases by using *phantom references*, a `Soot` internal representation of classes that are unavailable for the analysis. Missing classes could influence the analysis' results however, thus it is unsound to utilize phantom references. Moreover, `AveDroid` expects the user to provide all referenced classes, otherwise the replacement library cannot be created. This includes any application dependencies. We observed that many apps incorporate additional libraries, such as *Google APIs*[4], *Apache Commons*[5], etc. in their apps. Therefore, it is cumbersome to analyze apps since we must obtain all dependencies first.

Table 5.2 illustrates the results of our scaling experiments. Under *Application*, we list the names of applications we examined. The second column, *Methods*, approximates the number of application methods. In particular, we looked at packages `x` that contain at least one registered application component. We count all methods as application methods for which the following two properties apply: (1) the method is located in a class that resides in `x` or a subpackage of `x`. (2) The method has a reference in the constant pool. This procedure is only an approximation because many developers include components of extensive libraries (e.g., advertisement

[3] `https://www.debian.org/`
[4] `https://developers.google.com/api-client-library/`
[5] `https://commons.apache.org/`

Application	Methods	Android Library		Placeholder Library[6]		
		RAM (MB)	Time (s)	RAM (MB)	Time (s)	Points-to
Button1	5	179	7	61	5	463
PDFViewer	450	330	23	314	21	799
Drupal	433	✗	✗	430	29	1,080
SonyNotes	1,447	427	42	811	70	1,733
WeatherFlow	2,161	482	36	✗	✗	2,696
TVPortal	8,701	1,630	141	✗	✗	3,428

Table 5.2: Placeholder library scaling

libraries) in their application. Moreover, the number of methods is of limited use when describing the application size due to varying method complexity. Under *Android Library* and *Placeholder Library*, we report the maximum memory usage and runtime of the taint analysis with `FlowDroid`. In the case of placeholder libraries, we additionally report the size of the library points-to set (see Section 2.4). For reference, we also included the application *Button1* from the `DroidBench` benchmark suite. The symbol "✗" means that the analysis timed out.

We observed that *none* of the real-world applications could be analyzed with `FlowDroid` using replacement libraries without modifications. That is, when we executed `FlowDroid` with replacement libraries, the analysis ran out of memory. We found that the class `java.lang.Object`$_{\text{AVE}}$ (i.e., the class `java.lang.Object` with `AveDroid` stub implementations) is the reason for the exponentially increasing demand of resources in some cases. In order to obtain results, we decided to replace the class `java.lang.Object`$_{\text{AVE}}$ with the implementation of a standard Android library. The particular reason for the increasing demand of resources remains unclear, though we ruled out that `java.lang.Object`$_{\text{AVE}}$ leads to a significant increase of the library points-to set. For example, in the case of *APVPDFViewer*, the library points-to set had 71 more elements (i.e. 870, instead of 799) using `java.lang.Object`$_{\text{AVE}}$. In the case of *Drupal*, the library points-to set had 250 more elements (i.e. 1,330, instead of 1,080). For the `DroidBench` test apps, we did not modify the placeholder libraries.

The results show that `AveDroid`'s placeholder libraries perform slightly better than the device library for smaller applications. For *Drupal*, the appropriate replacement library even enables the analysis. We did not find any indications that could explain why `FlowDroid` times out when analyzing *Drupal* with the device library. According to our results, replacement libraries require exponentially more memory starting from a certain application size. While *SonyNotes* still works with the placeholder library, the analysis also requires twice as much resources compared to the device library. Regarding the two largest applications we examined, we suspect that the library points-to sets have too many elements for an efficient analysis.

5.3 Call Graph Comparison

We have seen in chapter 3 that call graphs derived from `FlowDroid`'s dummy main method are unsound. Therefore, it is impossible to use these call graphs to infer soundness of other call graphs. Due to static analysis challenges (see Section 2.3), static call graphs are not sound in general. In order to determine the degree of unsoundness of static call graphs, it is a common approach to compare them to dynamic call graphs. A dynamic call graph can be obtained by running the program on a device and logging all methods that are visited. Dynamic call graphs comprise only methods of a specific run, thus they are unsound in general as well. However, if a call edge of a dynamic call graph is not present in the static call graph, it can be concluded that the static call graph is unsound with respect to that call edge.

[6] With the exception of the placeholder library for the app *Button1*, the class `java.lang.Object` was replaced in all placeholder libraries with an implementation of a standard Android library.

Application	Dynamic C.	$\text{SPARK}_{\text{FLOW}}$		$\text{SPARK}_{\text{AVE}}$	
		Unsound edges (dynamic)	Edges (total)	Unsound edges (FlowDroid)	Spurious edges (FlowDroid)
Button1	60%	0	6	0	0
PDFViewer	5%	0	686	7	22
Drupal	14%	0	574	8	59
SonyNotes	✗	✗	1,599	42	252
WeatherFlow	✗	✗	1,393	87	340
TVPortal	9%	0	5,560	650	431

Table 5.3: Application-only call graph comparison

`Tracedroid` [51] is a tool that generates dynamic traces of Android applications. To run applications, `Tracedroid` uses an Android emulator that runs on a modified Android OS to log method invocations. The tool performs the analysis automatically by simulating user input with *Monkey Exerciser* [52] and other events, such as incoming network traffic and low battery status. We exercise `Tracedroid` to construct a dynamic call graph for each tested application, i.e. we parse the dynamic trace and convert it to a representation that is comparable to static call graphs.

Furthermore, we computed two static call graphs with `SPARK` for each tested application. First, we constructed a call graph derived from the dummy main method that we generated with `FlowDroid`. This call graph is built in conjunction with an Android 4.2 device library. The second call graph is based on the appropriate placeholder library, i.e. the placeholder library as well as its entry point is used to construct this call graph. Both call graph analyses are field-sensitive, array-insensitive, flow-insensitive, and context-insensitive.

Table 5.3 compares the application-only call graphs, i.e. call edges that connect two library methods are not considered in our experiments. We analyzed the same applications as in Section 5.2, as listed in the first column. Under the second column, *Dynamic Coverage*, we report the relative amount of application methods that were visited by the dynamic instrumentation with `Tracedroid`. The third column, $\text{SPARK}_{\text{FLOW}}$, lists the number of unsound call edges with respect to the dynamic call graph and the amount of call edges present in the call graph that is derived from `FlowDroid`'s dummy main method. Under $\text{SPARK}_{\text{AVE}}$, we report the number of unsound call edges as well as the number of spurious call edges with respect to the call graph generated with `FlowDroid`. Although this comparison is not ideal, the code coverage of the dynamic analysis is too low to infer soundness of other call graphs. From the low application code coverage we conclude that automated dynamic instrumentation is not suitable for Android applications. In two cases, *SonyNotes* and *Weatherflow*, we could not obtain a dynamic call graph because `Tracedroid` failed to install the app on the emulator.

$\text{SPARK}_{\text{AVE}}$ has approximately 10% or less spurious call edges in 4 of the 6 tested applications with respect to the total amount of call edges in $\text{SPARK}_{\text{FLOW}}$. We count every call edge as spurious that is contained in $\text{SPARK}_{\text{AVE}}$ and absent in $\text{SPARK}_{\text{FLOW}}$. Since $\text{SPARK}_{\text{FLOW}}$ is unsound, call edges that we identify as spurious in $\text{SPARK}_{\text{AVE}}$ might be unsound call edges in $\text{SPARK}_{\text{FLOW}}$. For example, we found that approximately 50% of the spurious call edges in *WeatherFlow* originate from, or target a *fragment*. Fragments are sub-components of activities that are not supported by `FlowDroid`. Although we did not confirm this through dynamic instrumentation, we suspect that some of these call edges are sound. We consider all missing call edges of $\text{SPARK}_{\text{AVE}}$ unsound that are present in $\text{SPARK}_{\text{FLOW}}$. For example, we observed that some call edges from callback handlers to the library are missing in $\text{SPARK}_{\text{AVE}}$. In the case of *TvPortal*, $\text{SPARK}_{\text{AVE}}$ misses call edges from and to entire classes. The reasons for this unsoundness remain unclear and are left for future work.

6 Related Work

In Chapter 3, we have seen a few closely-related works in more depth. These works will not be discussed here. In this chapter, we summarize further works that create static models of the Android framework as well as works that create pre-computed summaries of Android libraries.

6.1 Static Models of the Android Library

Similar to FlowDroid, ScanDroid [53] performs an information-flow analysis on Android applications. ScanDroid's analysis scope is broader than FlowDroid's approach by supporting intra- and inter-application communication. The analysis is less precise because ScanDroid does not craft a special entry point that simulates the lifecycle, but instead connects call graphs from multiple entry points. ScanDroid also provides hand-crafted summaries of specific library methods to reduce analysis times.

IccTA [8] is a client taint analysis that extends the functionality of FlowDroid. It combines the analysis of FlowDroid with Epicc [9]. Epicc is a tool to analyze intra- and inter-application communication. IccTA builds a dummy main method with a modified version of FlowDroid and facilitates the ICC data obtained with Epicc in its analysis.

DroidSafe [54] is also an information-flow analysis tool that incorporates ICC information in its analyses. The static abstraction is based on a proprietary AOSP library which contains summaries for native code. These summaries are specifically designed for client taint and points-to analyses. Similar to FlowDroid, DroidSafe crafts a main entry point (harness) to simulate the lifecycles of application components.

Smartdroid [55] focuses similarly to GATOR on GUI components of Android apps. It utilizes static and dynamic analysis techniques to find method invocations of sensitive library methods. Their static model aims to abstract the call hierarchy of activities, i.e. starting from the main activity, Smartdroid tries to find all activities that can be invoked within the current activity.

6.2 Android Library Summaries

EdgeMiner [56] is an analysis tool that creates application-independent summaries of implicit control-flows (callbacks) for Android libraries. It analyzes the whole Android library to find all registration-callback pairs. Computed summaries by EdgeMiner encapsulate all callback methods that might be invoked by library methods. Existing client taint analyses like FlowDroid can leverage this information to improve callback handling.

A work by D. Perez and W. Le [57] proposes a similar approach. Their tool, Lithium, performs an analysis of the Android framework that summarizes callbacks of Android API methods. Compared to EdgeMiner, Lithium is more expressive because it also captures control flow information of callbacks, though the analysis is also more expensive. Edgeminer and Lithium create more precise summaries than our approach because we provide only one summary node that over-approximates all potential callbacks.

7 Conclusion and Future Work

In this thesis, we have presented and evaluated `AveDroid`, a tool that creates replacement libraries for Android applications. We have seen that modeling the Android framework is challenging and unsound in general. Therefore, existing approaches focus on modeling the parts of the Android library that are important for the respective client analysis. Our static model of the Android library is akin to `FlowDroid`'s model, thus we have similar limitations. Further, we have evaluated that placeholder libraries enable an efficient taint analysis with `FlowDroid` up to a certain application size. With increasing application size, the taint analysis needs exponentially more resources.

By comparing call graphs, we have found that call graphs derived from our model miss sound call edges present in call graphs derived from a dummy main method that was created with `FlowDroid`. Therefore, we suggest that the first step for future work should be to clarify the reasons for the unsoundness in our model. Moreover, our model has limitations, such as missing handling for ICC, reflection, and native methods. While all these limitations lead to unsoundness in call graph construction, we think that supporting ICC should be one of the next steps since static models of other approaches could potentially be adopted in order to cope with ICC.

Declaration of Academic Integrity

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Date: Signature:

Bibliography

[1] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 259–269.

[2] Ali, K. and Lhoták, O. "Averroes: Whole-program analysis without the whole program". In: *European Conference on Object-Oriented Programming*. Springer. 2013, pp. 378–400.

[3] IDC. *Smartphone OS market share, 2016 q2*. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on: 09/21/2016).

[4] AppBrain. *Number of available applications on Google Play*. URL: http://www.appbrain.com/stats/number-of-Android-apps (visited on: 10/02/2016).

[5] G DATA. *Mobile malware report*. URL: https://file.gdatasoftware.com/web/en/documents/whitepaper/G_DATA_Mobile_Malware_Report_H1_2016_EN.pdf (visited on: 10/02/2016).

[6] Johnson, R., Wang, Z., Gagnon, C., and Stavrou, A. "Analysis of Android applications' permissions". In: *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. IEEE. 2012, pp. 45–46.

[7] Gibler, C., Crussell, J., Erickson, J., and Chen, H. "AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale". In: *International Conference on Trust and Trustworthy Computing*. Springer. 2012, pp. 291–307.

[8] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. "IccTA: Detecting inter-component privacy leaks in Android apps". In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 280–291.

[9] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Le Traon, Y. "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis". In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 543–558.

[10] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. "Chex: statically vetting Android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 229–240.

[11] Payet, É. and Spoto, F. "Static analysis of Android programs". In: *Information and Software Technology* 54.11 (2012), pp. 1192–1201.

[12] Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. "Optimizing Java bytecode using the Soot framework: Is it feasible?" In: *International conference on compiler construction*. Springer. 2000, pp. 18–34.

[13] IBM. *T.J. Watson Libraries for Analysis WALA*. URL: http://wala.sourceforge.net (visited on: 10/02/2016).

[14] Dean, J., Grove, D., and Chambers, C. "Optimization of object-oriented programs using static class hierarchy analysis". In: *European Conference on Object-Oriented Programming*. Springer. 1995, pp. 77–101.

[15] Bacon, D. F., Wegman, M, and Zadeck, K. "Rapid type analysis for C++". In: *Rapport technique* (1996).

[16] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. *Practical virtual method call resolution for Java*. Vol. 35. 10. ACM, 2000.

[17] Lhoták, O. and Hendren, L. "Scaling Java points-to analysis using Spark". In: *International Conference on Compiler Construction*. Springer. 2003, pp. 153–169.

[18] ARZT, S. and BODDEN, E. "StubDroid: automatic inference of precise data-flow summaries for the Android framework". In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 725–735.

[19] GOOGLE INC. *Android security overview*. URL: https://source.android.com/security/ (visited on: 10/25/2016).

[20] GOOGLE INC. *Security-Enhanced Linux in Android*. URL: https://source.android.com/security/selinux/ (visited on: 10/25/2016).

[21] GOOGLE INC. *Requesting Permissions at Run Time*. URL: https://developer.android.com/training/permissions/requesting.html (visited on: 10/25/2016).

[22] GOOGLE INC. *Intents and Intent Filters*. URL: https://developer.android.com/guide/components/intents-filters.html (visited on: 10/30/2016).

[23] LI, L., BISSYANDE, T. F.D. A., PAPADAKIS, M., RASTHOFER, S., BARTEL, A., OCTEAU, D., KLEIN, J., and LE TRAON, Y. *Static Analysis of Android Apps: A Systematic Literature Review*. Tech. rep. SnT, 2016.

[24] GOOGLE INC. *Activities*. URL: https://developer.android.com/guide/components/activities.html (visited on: 10/25/2016).

[25] ARZT, S., RASTHOFER, S., and BODDEN, E. "Instrumenting Android and Java applications as easy as abc". In: *International Conference on Runtime Verification*. Springer. 2013, pp. 364–381.

[26] GOOGLE INC. *Android Open Source Project*. URL: https://source.android.com/ (visited on: 10/30/2016).

[27] ALI, K. "The Separate Compilation Assumption". PhD thesis. University of Waterloo, 2014.

[28] LI, L., BISSYANDÉ, T. F., OCTEAU, D., and KLEIN, J. "Droidra: Taming reflection to support whole-program analysis of Android apps". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 318–329.

[29] LIN, C.-M., LIN, J.-H., DOW, C.-R., and WEN, C.-M. "Benchmark dalvik and native code for Android system". In: *Innovations in Bio-inspired Computing and Applications (IBICA), 2011 Second International Conference on*. IEEE. 2011, pp. 320–323.

[30] SON, K.-C. and LEE, J.-Y. "The method of Android application speed up by using NDK". In: *2011 3rd International Conference on Awareness Science and Technology (iCAST)*. IEEE. 2011, pp. 382–385.

[31] LEE, S. and JEON, J. W. "Evaluating performance of Android platform using native C for embedded systems". In: *Control Automation and Systems (ICCAS), 2010 International Conference on*. IEEE. 2010, pp. 1160–1163.

[32] AFONSO, V., BIANCHI, A., FRATANTONIO, Y., DOUPÉ, A., POLINO, M., GEUS, P. de, KRUEGEL, C., and VIGNA, G. "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy". In: *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*. 2016.

[33] WANG, Y., ZHANG, H., and ROUNTEV, A. "On the unsoundness of static analysis for Android GUIs". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2016, pp. 18–23.

[34] LAM, P., BODDEN, E., LHOTÁK, O., and HENDREN, L. "The Soot framework for Java program analysis: a retrospective". In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. Vol. 15. 2011, p. 35.

[35] BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., and MEZINI, M. "Taming Reflection". In: *International Conference on Software Engineering*. Vol. 4. 2011, p. 25.

[36] GRACE, M. C., ZHOU, W., JIANG, X., and SADEGHI, A.-R. "Unsafe exposure analysis of mobile in-app advertisements". In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2012, pp. 101–112.

[37] ZHOU, X., DEMETRIOU, S., HE, D., NAVEED, M., PAN, X., WANG, X., GUNTER, C. A., and NAHRSTEDT, K. "Identity, location, disease and more: Inferring your secrets from Android public resources". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 1017–1028.

[38] Liu, X., Zhou, Z., Diao, W., Li, Z., and Zhang, K. “An Empirical Study on Android for Saving Non-shared Data on Public Storage”. In: *IFIP International Information Security Conference*. Springer. 2015, pp. 542–556.

[39] Lortz, S., Mantel, H., Starostin, A., Bähr, T., Schneider, D., and Weber, A. “Cassandra: Towards a certifying app store for Android”. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM. 2014, pp. 93–104.

[40] Huang, W., Dong, Y., Milanova, A., and Dolby, J. “Scalable and precise taint analysis for Android”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM. 2015, pp. 106–117.

[41] Blackshear, S., Gendreau, A., and Chang, B.-Y. E. “Droidel: A general approach to Android framework modeling”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2015, pp. 19–25.

[42] Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., and Rountev, A. “Static Window Transition Graphs for Android (T)”. In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015, pp. 658–668.

[43] Google Inc. *Tasks and a Back Stack*. URL: https://developer.android.com/guide/components/tasks-and-back-stack.html (visited on: 11/29/2016).

[44] Wu, H., Yang, S., and Rountev, A. “Static Detection of Energy Defect Patterns in Android Applications”. In: *International Conference on Compiler Construction*. 2016, pp. 185–195.

[45] Wang, Y. and Rountev, A. “Profiling the Responsiveness of Android Applications via Automated Resource Amplification”. In: *IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 2016, pp. 48–58.

[46] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. “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*. ACM. 2012, pp. 27–38.

[47] Freke, J. *Smali, an assembler/disassembler for Android’s dex format*. https://github.com/JesusFreke/smali. 2013.

[48] Dahm, M., Zyl, J van, and Haase, E. *The bytecode engineering library (BCEL)*. 2003.

[49] Bartel, A. “Security Analysis of Permission-Based Systems using Static Analysis: An Application to the Android Stack”. PhD thesis. University of Luxembourg, Luxembourg, 2014.

[50] Ben Livshits. *Stanford SecuriBench Micro*. URL: http://suif.stanford.edu/~livshits/work/securibench-micro/ (visited on: 12/05/2016).

[51] Van Der Veen, V., Bos, H., and Rossow, C. “Dynamic analysis of Android malware”. In: *Internet & Web Technology Master thesis, VU University Amsterdam* (2013).

[52] Google Inc. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey.html (visited on: 12/05/2016).

[53] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. “SCanDroid: Automated security certification of Android applications”. In: *University of Maryland, Tech. Rep. CS-TR-4991* (2009).

[54] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. “Information Flow Analysis of Android Applications in DroidSafe.” In: *NDSS*. Citeseer. 2015.

[55] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. “Smartdroid: an automatic system for revealing ui-based trigger conditions in Android applications”. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2012, pp. 93–104.

[56] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. “EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.” In: *NDSS*. 2015.

[57] Perez, D. D. and Le, W. *Summarizing Control Flow of Callbacks for Android API Methods*. http://web.cs.iastate.edu/~weile/docs/dominguez_techreport1603.pdf. Technical Report, Iowa State University. 2016.