Program Analysis for Anomaly Detection

A dissertation submitted towards the degree Doctor of Engineering

of the Faculty of Mathematics and Computer Science of Saarland University

Vitalii Avdiienko

Saarbrücken, 2020

Day of Defense	02.12.2020
Dean	Univ.-Prof. Dr. Thomas Schuster
Head of the Examination Board	Prof. Dr. Sven Apel
Members of the Examination Board	Prof. Dr. Andreas Zeller, Dr. Sven Bugiel, Dr. Rafael Dutra

Abstract

When interacting with mobile applications, users may not always get what they expect. For instance, when users download Android applications from a market, they do not know much about their actual behavior. A brief description, a set of screenshots and a list of permissions, which give a high level intuition of what applications might be doing, form user expectations. However applications do not always meet them. For example, a gaming application intentionally could send SMS messages to a premium number in a background without a user's knowledge. A less harmful scenario would be a wrong message confirming a successful action that was never executed.

Whatever the behavior of a mobile application might (app) be, in order to test and fully understand it, there needs to be a technique that can analyse the User Interface (UI) of the app and the code associated with it as the whole.

This thesis presents a static analysis framework called SAFAND[1] that given an ANDROID app performs the following analysis:

- gathers information on how the application uses sensitive data;
- identifies and analyses UI elements of the application;
- binds UI elements with their corresponding behavior.

The thesis illustrates how results obtained from the framework can be used to identify problems ranging from small usability issues to malicious behavior of real-world applications.

[1] SAFAND = Static Analysis For ANomaly Detection

Zusammenfassung

Bei der Interaktion mit mobilen Anwendungen erhalten Benutzer möglicherweise nicht immer das, was sie erwarten. Wenn Benutzer beispielsweise Android-Anwendungen von einem Marktplatz herunterladen, wissen sie nicht viel über das tatsächliche Verhalten dieser Anwendungen. Eine kurze Beschreibung, eine Reihe von Screenshots und eine Liste von Berechtigungen, die eine umfassende Vorstellung davon geben sollen, welche Anwendungen möglicherweise ausgeführt werden können, bilden die Erwartungen der Benutzer.

Die Anwendungen entsprechen diesen Erwartungen aber nicht immer. Zum Beispiel könnte ein Spiel ohne Wissen des Benutzers im Hintergrund absichtlich SMS-Nachrichten an eine Premium-Nummer senden. Ein weniger schädliches Szenario wäre eine falsche Meldung, welche eine erfolgreiche Aktion bestätigt, die jedoch niemals durchgeführt wurde.

Unabhängig vom Verhalten einer mobilen Anwendung (App) muss eine Technik vorhanden sein, die die Benutzeroberfläche (User Interface, UI) der App und des damit verbundenen Codes testet und als Ganzes versteht.

In dieser Arbeit wird ein statisches Analyseframework namens SAFAND[2] vorgestellt, bei dem eine ANDROID-App die folgende Analyse durchführt:

- sammelt Informationen darüber, wie die Anwendung sensible Daten verwendet;
- identifiziert und analysiert UI-Elemente der Anwendung;
- verbindet UI-Elemente mit ihrem entsprechenden Verhalten.

Die Arbeit zeigt, wie Probleme, von kleinen Usability-Problemen bis hin zu böswilligem Verhalten realer Anwendungen, mit den Ergebnissen des Frameworks identifiziert werden können.

[2] SAFAND = Static Analysis For ANomaly Detection

Acknowledgments

First of all, I would like to thank my advisor Prof. Andreas Zeller for his exceptional support in my PhD life. I cannot imagine my achievements without him.

I am grateful to Konstantin Kuznetsov and Dr. Alessandra Gorla for being my research consultants and co-authors of all publications. Alessandra, your proof-reading skills are amazing, especially in the last night before a deadline.

I am also very grateful to Dr. Alessio Gambi for his valuable suggestions and proof-reading the thesis.

I would like to say my big thank you to all former and current members of the Software Engineering Chair. It was a big pleasure to work with you!

Last but not least, I am grateful to my family for the support throughout my PhD life. A special thanks goes to my wife Kristina for believing in me, supporting me and pushing me in critical moments.

Contents

Chapter 1

Introduction

People like to use mobile applications, it is quite convenient to have an instant access to services, data and media out of the pocket. According to the recent statistics [16], the Google Play Android application market [23] has about 70% of the market share and contains two and half million applications [50]. It is thus not surprising that there are lots of bad quality apps among them.

1.1 Research Problem

When users install applications they already have expectations on how these applications should work. The expectations are usually based on their previous experience with similar apps. Sometimes applications do not meet user expectations because of their intransparent or harmful behavior: app developers can intentionally hide or mask functionality and aim to steal user's personal information or disguises the use of premium services [6]. Although the quality standards of the official Android market are generally high, malicious applications can be still found, and they are more prevalent in other markets [18][53][21]. Despite being the main focus of the research on Android, malware detection is still an open challenge [38]. In fact, as malware detection techniques improve, malware writers find new ways to hide malicious behavior. As a consequence, it becomes more and more difficult to understand whether an application has some hidden behavior.

Expectations of users are unmet even if a developer made a mistake. In this case, an unintentionally introduced bug can lead to behavioral and functional changes. Such changes can be visible, e.g., UI mistakes, or invisible, e.g., an incorrect condition in the code resulting in the completely different functionality.

Whatever the conditions and intentions could be there is a need of a framework that could analyze applications as a whole: from the UI up to its code discovering the corresponding functionality.

There are two ways to perform program analysis: statically and dynamically.

Static program analysis is sound, i.e., it reports all possible functionalities of an application, but it also introduces false positives. For example, it warns a user that the application under analysis leaks some sensitive information, but in reality these are infeasible paths. This happens because some conditions cannot be resolved statically, and consequently the analysis over-approximates the results. The false positive rate is not the only problem of static taint analysis. In order to achieve soundness, the analysis may be complex and expensive, and might consequently incur scalability problems. Static analysis has also well-known limitations. Namely, it struggles with code reflection, code obfuscation and encryption. Code reflection as well as code obfuscation are quite common techniques to circumvent signature-based malware detection techniques. Moreover, most techniques implemented for Android do not support native code analysis. Native code (C/C++ code) is widely used in a development of games in order to have better memory management and performance. It is thus necessary to support these features when analyzing Android applications.

Dynamic program analysis can only report behavior that has been observed during actual executions. The problem of dynamic approaches is that they mainly depend on the set of inputs that have been used to generate executions, and can thus lack relevant information. Moreover, mobile devices have limited resources such as CPU, RAM, storage and battery life, and it is challenging to implement dynamic analyses on such devices.

At first sight, dynamic program analysis is the best choice, but unfortunately in reality it becomes infeasible. The prevalence of modern applications requires a user account. Users must provide login credentials for allmost every application. Such requirement is practically unrealizable for automatic approaches thousands of applications have to be analyzed. Thus, static techniques are the only option to fully analyze behavior of applications.

As of December 2017 there is 3.5 million of application in Google Play Store [50]. However, there are a lot of bad-quality applications among them: from apps with bad design to apps with harmful behavior. In order to identify such bad-quality applications it is necesarry to understand their behavior.

1.2 Contributions of the Thesis

The thesis presents a set of static analysis techniques to investigate ANDROID applications called SAFAND (Static Analysis For ANomaly Detection). SAFAND implements techniques to extract program features which are useful to find anomalies both in behavior of applications and their UI elements. SAFAND is a synergy of program analysis stages of BACKSTAGE [13] and MUDFLOW [12] tools.

MUDFLOW describes a behavior of an application by using flows of sensitive data between its APIs. For example, Travel Guide app usually sends a device location to the internet in order to suggest best places to visit. MUDFLOW constructs behavioral patterns of applications based on their sensitive data flows and group them. At the end, it identifies abnormal applications as their behavior do not fit in the group's pattern. Such abstraction is very useful not only for automatic classification, but also for general understanding of what applications doing. In the practice MUDFLOW was able to reveal malicious behavior of applications based on extracted dataflows with high accuracy.

Modern applications, in fact, access a lot of Android APIs to improve user's experience and provide more functionality. This makes it more problematic to identify abnormal behavior as it becomes less distinguishable on the API and dataflows level. To address this issue, we developed a technique called BACKSTAGE, which analyses GUI of applications and connects it to the corresponding behavior. BACKSTAGE can detect and report triggered APIs and dataflows, for example, after clicking on the *Share* button. Such technique makes it possible to identify whether some action happens automatically without user's notice or his explicit agreement. Moreover, this technique gives the possibility to understand whether the app meets user's expectations in terms of its functionality. The proposed approach improves MUDFLOW and introduces new program features which are helpful for end-users and security researches to understand how applications behave. BACKSTAGE, thus, is very useful for detecting typical programming failures, such as assigning a wrong callback function to the button, as well as for identifying apps with suspicious behavior.

Together with the static analysis framework the thesis presents a study of the typical information flows of Android applications that are popular on the Google Play market.

Despite the fact that the presented techniques are targeted to ANDROID applications, they can be easily adopted to analyze apps from other platforms like Windows Mobile[58] or iOS[31].

SAFAND has a modular structure and technically is built on the top of SOOT [35] framework.

It consists of five main independent modules (see Figure 1.1):

Resource Analysis SAFAND takes an application as an input and dissolves it to analyze its static resources, such as screen layouts, icons and text declarations.

Mining Sensitive Data Flows from the application SAFAND leverages FLOWDROID [8] static taint analysis tool in order to extract flows of sensitive data in the application. SAFAND implements customizations to dissolve access to sensitive resources and resolves the target activity and data of intercomponent calls.

Mining Sensitive APIs from the application SAFAND leverages a call-graph of FLOWDROID and traverses it to find all reachable Sensitive APIs. SAFAND implements optimizations to minimize the possibility of discovering APIs in the dead code.

Mining GUI Elements from screens For each UI screen SAFAND extracts the set of UI elements associated with it.

Identifying a behavior of UI elements For each extracted UI element on the screen, SAFAND associates it with a set of APIs and Data Flows. Such association represents a mapping between the *expected behavior of the UI element* (e.g. what a user sees on the screen) and its *actual behavior* (e.g. what the UI element actually does).

SAFAND is the highly customizable framework. For example, UI analysis module is fully independent and can be run without any prerequisites, while the association of API and Data Flows with UI elements modules requires results of UI phase as an input. One can run UI analysis module once and then reuse results in the future as SAFAND provides an option to load previously saved results of UI analysis.

To support further research in app mining, as well as replication and extension of the results in this thesis, all our data is available for download. For details, see our project pages

https://www.st.cs.uni-saarland.de/appmining/mudflow
https://www.st.cs.uni-saarland.de/appmining/backstage/

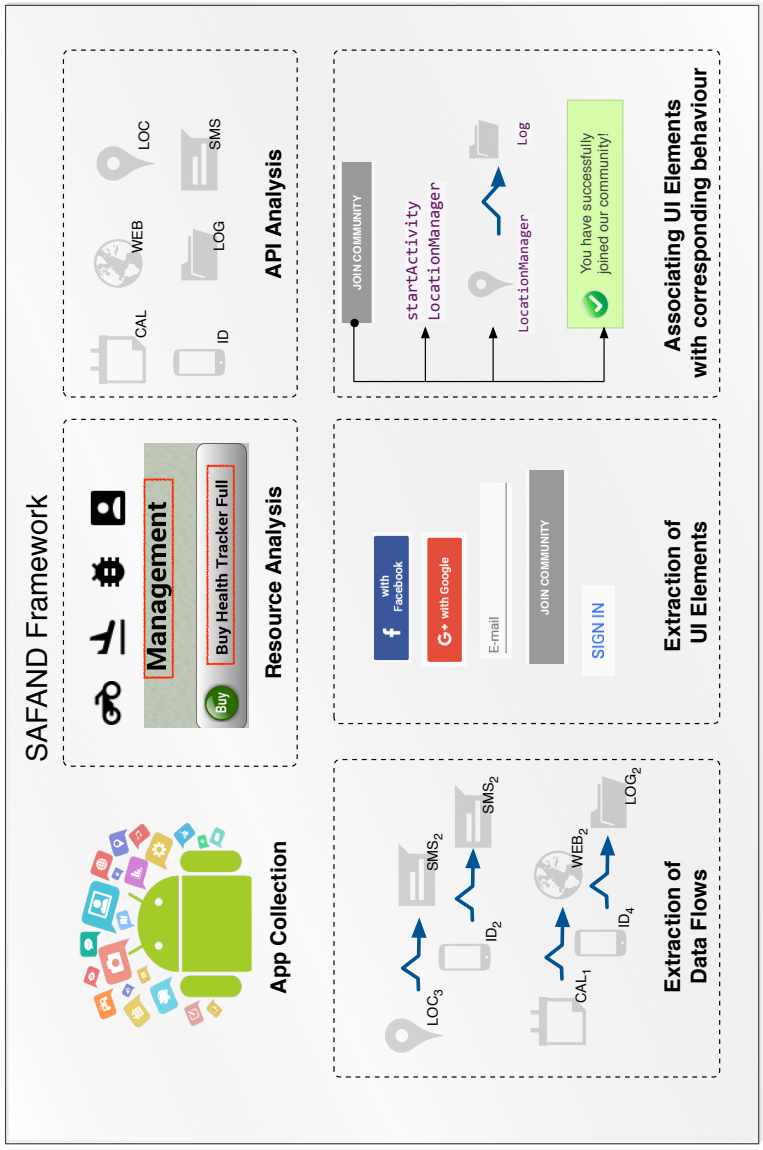

Figure 1.1: Overview of the SAFAND framework.

The source code of the MUDFLOW program analysis could be found in

https://github.com/uds-se/soot-infoflow-android
https://github.com/uds-se/soot-infoflow

The source code of the BACKSTAGE program analysis could be found in

https://github.com/uds-se/backstage

1.3 Research Questions

The goal of this thesis is to present static analysis techniques which can be useful to effectively identify applications with suspicious behavior. Therefore research questions are:

RQ1 : Can APIs on itself can be used to identify applications with suspicious and malicious behavior?

RQ2 : Can flows of sensitive information in an application on itself can be used to identify its suspicious and malicious behavior?

RQ3 : Can a combination of UI and its corresponding behavior (e.g., APIs or flows of sensitive data) can be used to identify applications whose behavior deviate from the expected one?

1.4 Structure of the Thesis

The thesis is organized as follows:

- In Chapter 2 we discuss the basics of ANDROID and its ecosystem. Further, we introduce data flow and taint analysis techniques as well as give a short overview of existing frameworks for analyzing ANDROID apps.
- In Chapter 3 we present a set of techniques to *mine* APIs and dataflows in a large number of Android applications. We will also show how extracted features can be used to *compare and classify* applications.
- In Chapter 4 we introduce an approach to *mine* user interfaces by extracting visible information from the screen of an application.

- In Chapter 5 we present a technique that improves previously discussed approaches. It searches for mismatches between the ANDROID APIs and Data Flows, which applications invoke, and the visible information on the screen by combining the approaches from Chapter 3 and Chapter 4.
- Chapter 6 provides conclusions and a discussion of the future work.

1.5 Publications

This thesis is built on the following papers (in chronological order):

1. **Vitalii Avdiienko**, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15), Vol. 1. IEEE Press, Piscataway, NJ, USA, pp. 426-436. [12]
2. **Vitalii Avdiienko**. 2015. Mining patterns of sensitive data usage. In Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15), Vol. 2. IEEE Press, Piscataway, NJ, USA, 891-894. [10]
3. **Vitalii Avdiienko**, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Abnormal Sensitive Data Usage in Android Apps. In Proceedings of the Jornadas Nacionales de Investigación en Ciberseguridad (JNIC 2016). Best Published Research Award. Granada, Spain.
4. **Vitalii Avdiienko**, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In Proceedings of Grande Region Security and Reliability Day 2015, Trier, Germany.
5. Konstantin Kuznetsov, **Vitalii Avdiienko**, Alessandra Gorla and Andreas Zeller. Checking App User Interfaces Against App Descriptions. In Proceedings of the International Workshop on App Market Analytics (WAMA 2016). [34]

6. **Vitalii Avdiienko**, Konstantin Kuznetsov, Paolo Calciati, Juan Carlos Caiza Roman, Alessandra Gorla and Andreas Zeller. CALAPPA: a Toolchain for Mining Android Applications. In Proceedings of the International Workshop on App Market Analytics (WAMA 2016). [11]

7. **Vitalii Avdiienko**, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla and Andreas Zeller. Detecting Behavior Anomalies in Graphical User Interfaces. Technical Report. [13].

8. **Vitalii Avdiienko**, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla and Andreas Zeller. Detecting Behavior Anomalies in Graphical User Interfaces. In Proceedings of the 39th International Conference on Software Engineering (ICSE 2017) - Poster Track.

9. **Vitalii Avdiienko**, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. App Mining. In Proceedings of SE 2017 (Tagung Software Engineering), Hannover, Germany.

Chapter 2

Background

2.1 Android

Android is a mobile operating system developed by Google[1]. Android is an open-source Linux-based software stack designed for a wide range of mobile devices. Nowadays more than 1.4 billion devices are running under Android[2]. Nevertheless, the platform architecture is universal across all of them. Vendors can introduce their specific changes in User Interface (UI) or in a collection of applications, but changes to the kernel itself are not permitted. In order to build a comprehensive program analysis technique it is important to know building blocks of the platform under test and working principles of apps.

2.1.1 Platform Architecture

Android is built on the top of the Linux Kernel. Such design takes an advantage of key security features of Linux kernels and allows vendors to work with the well-know software. The diagram on Figure 2.1 shows the major components of Android platform.

The Linux Kernel provides the possibility to access sensors, memory, a display and wireless channels of communication. It simplifies and unifies access to the most-desired hardware functionality. Power Management is also encapsulated within Linux Kernel so that developers and software vendors do not have

[1] https://www.android.com/

[2] http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide

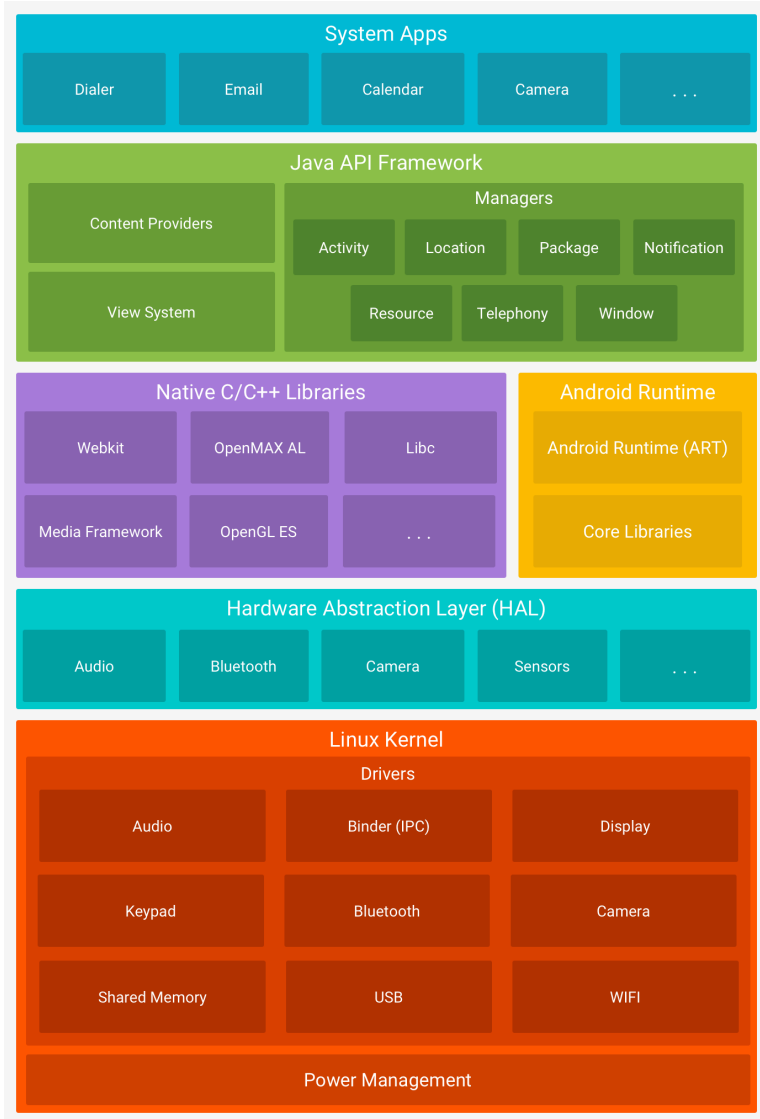

Figure 2.1: Components of Android Platform [44]

to care about it and better spend their valuable time on implementing software features.

Hardware Application Layer (HAL) provides interfaces to expose hardware capabilities to the high-level Java API framework. It defines the standard interfaces for hardware vendors and allows Android to be agnostic about lower-level driver implementations.

Prior to Android 5.0 a mobile application was executed in its own instance of the Dalvik Virtual Machine (VM) and it was fully isolated from other apps on the same device. Each application was executed in a full isolation by using default Linux security mechanisms: a Linux Id and a Group Id uniquely identified each application. The Dalvik VM, however, has the register-based architecture in contrast to Java VMs, which are stack-based. Such architectural solution caused a need of have an operating system with low battery consumption. In addition, the Dalvik VM has the CPU-optimized bytecode interpreter and uses the runtime memory very efficiently. It executes Dalvik EXecutable (DEX) files resulted from the compilation of the Java code. Each application can have multiple DEX files, as each DEX file is limited to 64,000 methods[3]. DEX files are part of an APK file, which in addition contains also resource files and metadata. The APK file is a final deliverable from developers.

Starting from Android 5.0 (Lollipop) Google introduced new Android Runtime(ART)[4]. ART uses Ahead-of-time compilation of the code which improves application performance. ART provides Ahead-of-time (AOT) compilation and Just-in-time (JIT) compilation, while Dalvik supports only AOT. Moreover, ART has the optimized garbage collection, better debugging and diagnostic support. Apps that are developed for ART-devices can also run on Dalvik-devices but not always vice-versa.

Many core Android system components and services, such as access to sensors via HAL interface[5], are built from native code that requires C/C++ libraries. The Android framework provides Java API to expose the functionality of some of these native libraries. Developers can also write apps using C/C++ and can have direct access to Android native libraries with help of Android NDK[6]. Android NDK is mostly used in game development, as it provides access to OpenGL library[7].

[3] https://developer.android.com/studio/build/multidex.html
[4] https://source.android.com/devices/tech/dalvik/
[5] https://source.android.com/devices/sensors/hal-interface.html
[6] https://developer.android.com/ndk/index.html
[7] https://developer.android.com/guide/topics/graphics/opengl.html

The most of Android apps are written in Java, as using Java API is the easiest and the most convenient way to develop them. The entire set of Android OS features like accessing the current location or contacts in the address book is available trough API written in Java language. Android SDK is shipped alongside with Android Studio[8] that helps developers to create and debug apps with little effort.

Android has a set of system apps. These apps are predefined in devices and cannot be uninstalled. They provide key capabilities that developers can access from their own apps. For example, one does not need to implement a functionality og making photos. One can just open a predefined Camera app that will do it and return the photo to the app. The same holds for sending emails, opening web-pages, making phone calls etc.

2.1.2 App Components

Android applications communicate with the Android platform through its Application Programming Interfaces (API), which can be used to access different platform functionalities. Within the API only a small set of methods is used for accessing sensitive information such as user's *address book*, *SMS messages* and *GPS coordinates*. To protect users from unauthorized usage of their sensitive information, Android requires *permissions* to access them. Prior to Android 6.0 these permissions were enlisted in the *Android Manifest XML* file. The Android platform reads them when installing an application and asks the user for a confirmation (Figure 2.2). The user must agree to grant all necessary permissions, otherwise the application will not be installed. If the application accesses sensitive information without asking permission, the Android platform will deny the access and throw an appropriate exception.

Starting from Android 6.0 (Marshmallow) users grant permissions to apps while the apps are running and not during their installation. It gives the user more control over the app's behavior, as it is almost impossible to understand why the app requires some permission before using it. The user can choose to allow the navigation app to access his location but not to the list od contacts in her address book (Figure 2.3).

Android applications have four types of components[9]:

- an *activity* represents a single screen with a user interface which user can interact with in order to do something. An application typically consists

[8] https://developer.android.com/studio/index.html
[9] https://developer.android.com/guide/components/fundamentals.html

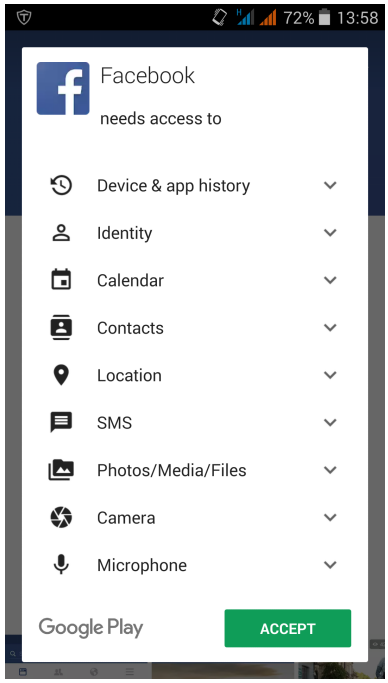

Figure 2.2: Requesting permissions while installing the Facebook app.

of multiple activities and each activity is responsible for a specific action, e.g. sign-up, an add payment method etc.

- a *service* is a component that runs in the background to perform long-running operations or work for remote processes. Services do not have user interfaces.
- a *content provider* manages a shared set of application data. It also enables sharing data between apps, e.g. the address book, the photo gallery. In order to access a specific resource developers need to provide a URI. For example, access to contacts in the address book is done by providing *ContactsContract.Contacts.CONTENT_URI*.

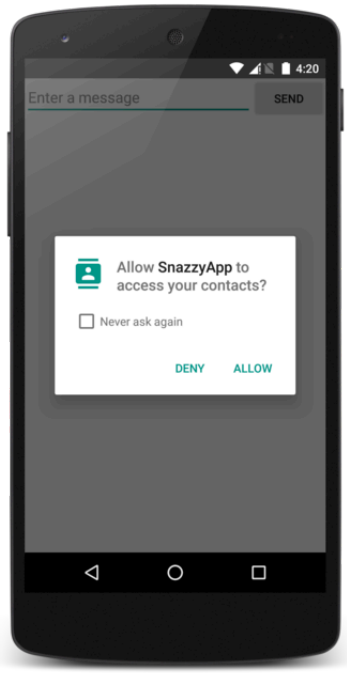

Figure 2.3: Requesting permissions at a runtime [5]

- a *broadcast receiver* is a component that responds to system-wide broadcast announcements. For example, it is a good approach to listen to system events such as the low battery state or whether a device is connected to power supply to adapt the behavior of the application in such scenarios.

An Android application typically has several activities, which can communicate by means of *Intents*. *Intents* can start activities within the same or from another application. *Intents* can be *explicit*, i.e. they can trigger a specific predefined component in the same application, and *implicit*, i.e. they can start any activity which supports a specific action. *Intents* are thus the main means to support Inter Process Communication within the Android platform. Intents are

widely used to start another screen or app and send messages to a background service.

2.1.3 Activity Lifecycle

Activities are essential building blocks of applications. Each application consists of multiple activities. Each activity usually represents a piece of functionality and has a connection with another activity. Consider the case of buying a pencil in a web-shop. First, you search for a pencil, then select it, enter a delivery address, select a payment method and, finally, order it. Each of these steps is typically placed in a separate activity, but they are interconnected, e.g. a user can go multiple steps back or proceed to the next step of a process. Such complex logic of activity management is the heart of the Android platform.

Activities of an application are stored in the activity stack. The most recent used activity is always located on the top of the stack. All other activities are inactive until they become the top element of the stack. Figure 2.4 illustrates the complete activity lifecycle. An activity has event-based nature and has the following states:

- If the activity is on the top of the activity stack and in the foreground of the screen, it is *active*.
- If the activity lost focus but is still visible, it is *paused*.
- If the activity is no longer visible, it is *stopped*.

Android has a predefined sequence of events in each scenario and the execution sequence is quite important. Developers should always keep in mind that *onStart* event follows *onCreate* event and precedes *onResume*. This provides a better control over the processing order of the certain parts of the code will be processed.

On one hand, such design gives more flexibility and power to developers. On the other hand, program analysis tools should take care of such complex lifecycle and create a precise model which takes into consideration the ordering of events. We will discuss how state-of-the-art static analysis tools deal with it in Section 3.3.

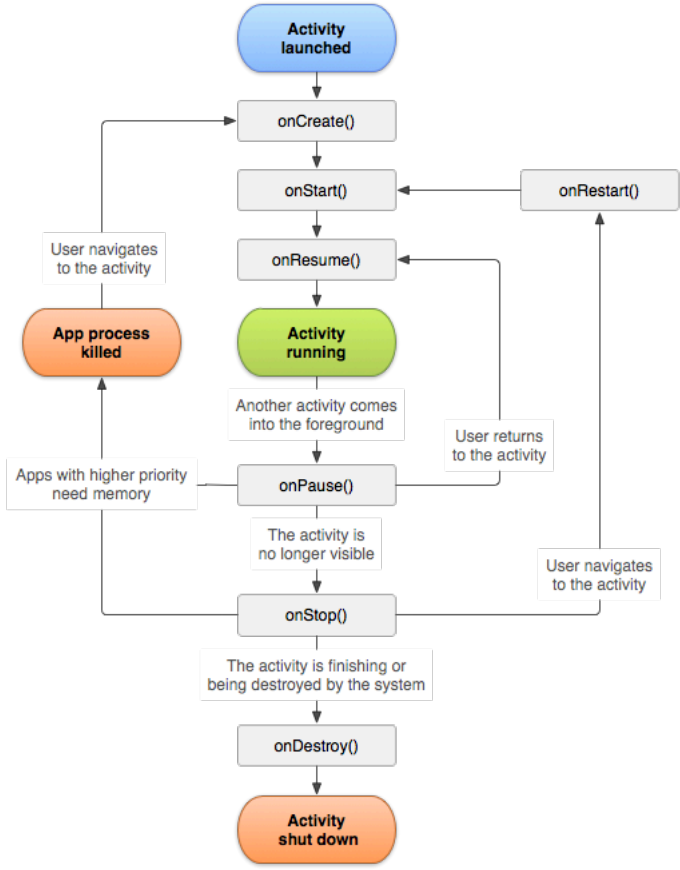

Figure 2.4: Activity Lifecycle [1]

2.2 Android User Interface

2.2.1 Defining and Combining Layouts

In the Android framework, an activity is a single screen containing several UI elements, such as buttons and text fields, organized in a hierarchy. Each app can and typically does contain multiple activities. The layout of the activity is usually declared in an XML file named `layout.xml`. Different files and names can be

used though, as developers can bind an activity to the layout XML file thanks to the `android.app.Activity:setContentView(layoutFileId)` method. Refer to Listing 2.1 for an example of the layout file.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout android:layout_width="fill_parent"
3        android:layout_height="fill_parent">
4        <fragment android:id="@+id/fragment"
5              class="uinomaly.fragmentclass".../>
6        <Button android:id="@+id/buttonOK"
7              android:text="@string/buttonOK"
8              android:onClick="xmlDefinedOnClick"
9              style="@style/okButtonStyle"/>
10       <ImageButton android:id="@+id/imageButtonPrint" ...
11             android:src="@drawable/print_button"
12             android:contentDescription="@string/printText" />
13 </LinearLayout>
```

Listing 2.1: A Sample Activity layout declared in a XML file.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3        <style name="okButtonStyle" parent="Theme.Material">
4              <item name="colorPrimary">#673AB7</item>
5              <item name="text">Save</item>
6        </style>
7  </resources>
```

Listing 2.2: A Definition of the Text using Styles.

Reusable layouts and fragments take an important place in the modern app development as there is a strong need in the flexible and adoptable design for hundreds of different devices with multiple versions of the Android platform.

A layout can be entirely or partially reused in different activities in multiple ways:

`<include>` and `<merge>` XML tags. Developers can include other XML files by means of the `<include>` tag. To do this they simply have to specify the file `Id` such as `<include layout="@layout/reusableLayout"/>`. Developers can also use the `<merge>` tag to achieve the same purpose with the advantage of eliminating redundant hierarchical elements.

Inflate layouts programmatically. `LayoutInflater` instantiates the corresponding layout file into a `View` object. The `View` object can later be added to the layout of the activity. This way makes it possible to assign

more than one layout to the activity, which can be only done from the Java code. Let's consider an example in Listing 2.3. It describes the way how to create a screen with two different layouts: one on the left side and one on the right side.

```
1        LinearLayout layoutMain = new LinearLayout(myActivity);
2        LayoutInflater inflate = (LayoutInflater) getSystemService(Context.
3        LAYOUT_INFLATER_SERVICE);
4        LinearLayout layoutLeft = (LinearLayout) inflate.inflate(R.layout.
5        leftlayout, null);
6        LinearLayout layoutRight = (LinearLayout) inflate.inflate(R.layout.
7        leftlayout, null);
8        layoutMain.addView(layoutLeft, layoutParams); layoutMain.addView(
   layoutRight, layoutParams);
9
```

Listing 2.3: Example of inflating layouts from the code

Fragments. Fragments are modular sections of an activity. Unlike the previously discussed layouts, a fragment has its own lifecycle and receives its own input events. The comprehensive information concerning lifecycle methods of fragments can be found in the official Android documentation[20]. According to it, there are two different ways to include a fragment into an activity statically through the layout file and programmatically through the code:

1. The simplest way is to declare a fragment in the layout file of the activity directly (see lines 5–11 in Listing 2.1 for an example).
2. It is also possible to create dynamically the fragment in the activity code by means of the `FragmentManager` class. This can be done by using `LayoutInflater:inflate` method which has been discussed above. The only difference is that this behavior should be implemented in `onCreateView` lifecycle method of fragment.

   A great advantage of using fragments in the activity is the ability to add, remove, replace and perform other actions with them in response to user interaction. Each set of changes that you commit to the activity is called a transaction and you can perform it using APIs in `FragmentTransaction`. The example in Listing 2.4 shows how to replace `my_button` element with the content of the `myFragmentClass` at runtime.

Table 2.1: Signatures of Fragment Lifecycle Methods

android.app.AlertDialog$Builder: void ⟨*init*⟩(android.content.Context)
android.app.AlertDialog$Builder: android.app.AlertDialog$Builder setTitle(int)
android.app.AlertDialog$Builder: android.app.AlertDialog$Builder setMessage (java.lang.CharSequence)
android.app.AlertDialog$Builder: android.app.AlertDialog$Builder setNeutralButton (java.lang.CharSequence,android.content.DialogInterface$OnClickListener)
android.app.AlertDialog$Builder: android.app.AlertDialog$Builder setPositiveButton (java.lang.CharSequence,android.content.DialogInterface$OnClickListener)
android.app.AlertDialog$Builder: android.app.AlertDialog$Builder setNegativeButton (java.lang.CharSequence,android.content.DialogInterface$OnClickListener)
android.app.AlertDialog$Builder: android.app.AlertDialog create()
android.app.AlertDialog: void show()

```
1         FragmentManager fragmentManager = getFragmentManager();
2         FragmentTransaction fragmentTransaction = fragmentManager.
    beginTransaction();
3         Fragment fragment = new MyFragmentClass();
4         fragmentTransaction.add(R.id.myButton, fragment);
5         fragmentTransaction.commit();
6
```

Listing 2.4: Manipulating fragments at runtime

2.2.2 Defining Alert Dialogs

Alert dialogs act as a confirmation pop-up windows that typically ask user to confirm some action (refer to Figure 4.1 for an example). An alert dialog is a special element that it is fully separated from the activity on which it appears. Considering the example in Figure 4.1, the text of the dialog does not have any relation to the tweets in the background and just asks for permission to use the device location for future purposes. Only the dialog message and button labels matter here. Each button in the dialog is responsible for a different kind of actions. There are positive, negative and neutral buttons. A callback of each button performs a completely different action. The full list of methods that are responsible for building dialogs can be found in Table 2.1. It is important to mention that the procedure of creating the alert dialog should be finished with invocations of `create()` and `show()` methods. Otherwise the dialog will not be displayed on the page.

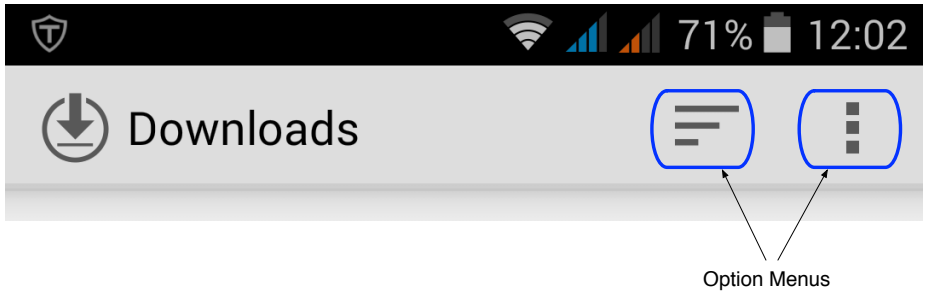

Figure 2.5: Example of Option Menu

2.2.3 Defining Menus

Additional challenges in the analysis of GUI comes from dealing with complex elements such as menus. The ANDROID framework provides several of them:

Option Menus. Option menus are placed in the top right corner of a screen. They usually give access to functionalities that are relevant for the application regardless of the context (e.g. the *Settings* button). Option menu items can be easily created by implementing the `onCreateOptionsMenu` method of the activity (refer to Figure 2.5 and Listing 2.5 for an example).

```
1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) {
3  MenuInflater inflater = getMenuInflater();
4  inflater.inflate(R.menu.optionmenu, menu);
5  return true;
6  }
7
```

Listing 2.5: Defining a layout of the option menu

Contextual Menus. Contextual menus appear when a user presses a UI element with a long-click. They can display further actions for a specific element especially inside a `ListView`. They are created by implementing the `onCreateContextMenu` method, and they can be bound to UI elements by means of `registerForContextMenu(elementId)` (refer to Figure 2.6 and Listing 2.6 for an example).

```
1         @Override
2         public void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenu.ContextMenuInfo menuInfo) {
3                 super.onCreateContextMenu(menu, v, menuInfo);
4                 MenuInflater inflater = getMenuInflater();
5                 inflater.inflate(R.menu.contextmenu, menu);
6         }
7
```

Listing 2.6: Defining a layout of the contextual menu

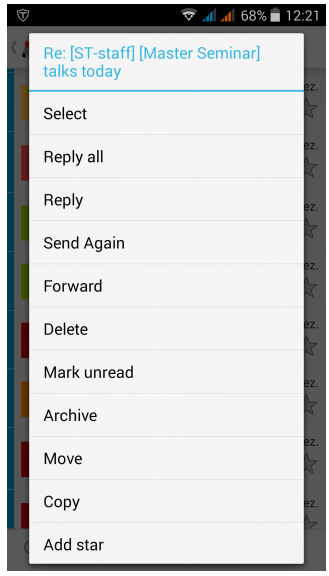

Figure 2.6: Example of Contextual Menu

Navigation-Drop-Down Menus. Navigation-drop-down menus are used for a quick and easy navigation through the whole application and can be identified through a small triangle in the lower right corner of the displayed text. The menu items are specified through the implementation of an adapter with the corresponding array of strings. To create such menus, the developer can invoke the `setListNavigationCallbacks(adapter, navListener)` method on an `ActionBar` instance (refer to Figure 2.7 and Listing 2.7 for an example). This menu was deprecated in the Android

API level 21. But there are many apps written for old ANDROID versions and they are still widely used.

```
1          ActionBar ab = getActionBar();
2          ab.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
3          SpinnerAdapter adapter = ArrayAdapter.createFromResource(this, R.
   array.myItems, android.R.layout.simple_list_item_1);
4          ActionBar.OnNavigationListener navListener = new ActionBar.
   OnNavigationListener(){
5                  @Override
6                  public boolean onNavigationItemSelected(int i, long l) {
7                          ...
8                          return false;
9                  }
10         };
11         ab.setListNavigationCallbacks(adapter, navListener);
12
```

Listing 2.7: Defining a layout of the navigation-drop-down menu

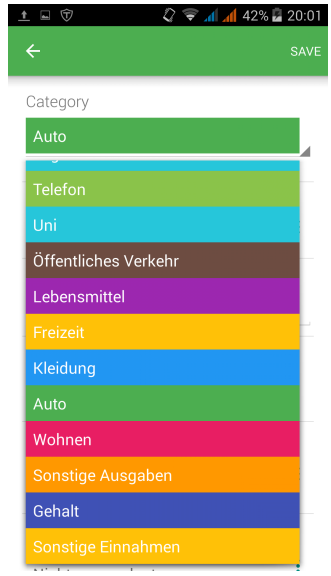

Figure 2.7: Example of Navigation-Drop-Down Menu

Drawer Layouts. Drawers are panels that can be opened with a swipe from the outer vertical side of the screen to the middle. A drawer can be created with a `DrawerLayout` tag in the XML layout file (refer to Figure 2.8 and Listing 2.8 for an example).

```
1        drawerLayout = (DrawerLayout) findViewById(R.id.drawerlayout);
2        drawer = (ListView) findViewById(R.id.drawer);
3        // set up the drawer's list view with items and click listener
4        drawer.setAdapter(
5        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1,
   android.R.id.text1, getResources().getStringArray(R.array.drawerItems)));
6        drawer.setOnItemClickListener(this);
7
```

Listing 2.8: Defining a layout of the drawers

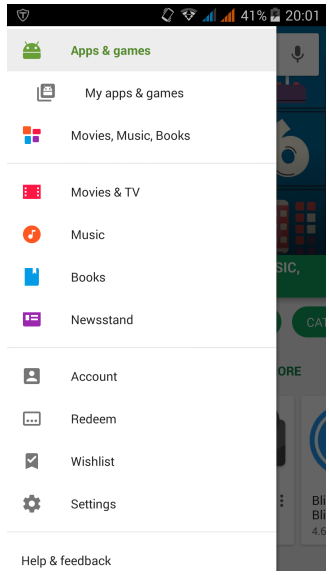

Figure 2.8: Example of Drawer Layout

Tab Views. Tab views are created dynamically via `actionBar.newTab()`, and can later be added to an action bar. Each tab view is represented by a fragment (refer to Figure 2.9 and Listing 2.9 for an example).

```
1  // setup action bar for tabs
2  ActionBar actionBar = getActionBar();
3  actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
4  actionBar.setDisplayShowTitleEnabled(false);
5
6  ActionBar.Tab tab = actionBar.newTab()
7  .setText(R.string.Tab1Title)
8  .setTabListener(new TabListener<Fragment1>(this, "Tab1", Fragment1.class
   ));
9  actionBar.addTab(tab);
10
```

Listing 2.9: Defining a layout of the tab views

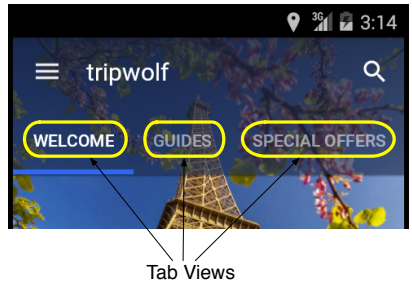

Figure 2.9: Example of Tab Views

2.2.4 Assigning Text to UI Elements

There are several ways in Android to define the text of UI elements:

Label assignment in layout files. Developers usually define the label of UI elements in the XML layout file by using the `android:text` attribute. Refer to line 13 in Listing 2.1 for an example. The text can be defined either by using the reference to the app's resources with the `"@string/"` prefix or directly by providing the string that will be displayed. Even if the second option is deprecated, as it introduces localization problems, it is still used. There are more attributes that can be used to set a label for a UI element (refer to Table 4.3).

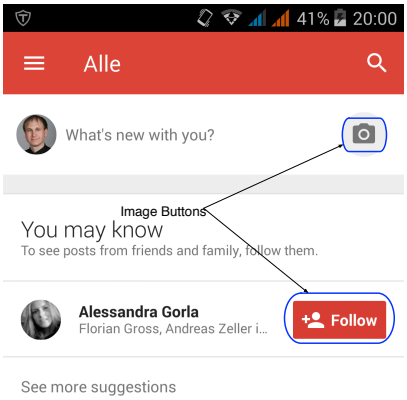

Figure 2.10: Example of Usage Actionable Icons

Label assignment in Java code. Layout templates can be reused across different activities. However, the text of the UI elements in such layouts usually differs depending on the context (i.e. activity). Therefore, developers usually assign textual labels to such UI elements in the code depending on the activity. `View:setText(resourceId)` and `View:setText(text)` allow to redefine labels for UI elements. Refer to Listing 4.16, lines 4 and 6 for an example.

Label assignment in style files. Developers can assign labels to UI elements using the `styles.xml` file. This option is typically used when the text of UI labels changes depending on the style. Developers can specify labels of UI elements by creating an `<item>` with the attribute `name="android:text"`. Refer to Listing 2.1, line 15 as an example.

2.2.5 Defining Icons

Icons are prevalent in GUIs as they can represent the semantic of UI elements in an intuitive way. A camera icon, for instance, can be easily interpreted by a user as a button to take pictures. Icons are extensively used in mobile GUIs also because they are more space efficient than text (refer to Figure 2.10 for an example).

Table 2.2: A set of attributes responsible for binding icons to UI elements

android:background	android:drawableRight	android:drawableTop
android:src	android:drawableLeft	android:drawableBottom
android:drawableEnd	android:drawableStart	

A sample use of icons for UI elements is reported in Listing 2.1, where the `print_button` icon is bound to the `ImageButton` element. The full list of attributes responsible for binding icons to UI elements is presented in Table 2.2.

2.3 Sensitive APIs in Android

As discussed in the introduction, APIs can be used as a proxy for behavior of an application. But which APIs are important for characterization the app behavior? Android has thousands of methods that provide a possibility to deal with its different features [46]. For program analysis it is important to concentrate on the predefined set of APIs to reduce the search space. Thus, in this thesis we concentrate on APIs that access *sensitive* information such as the user's address book or the microphone, or perform sensitive tasks, e.g. altering system settings, sending messages, etc. For this, though, we need to identify which APIs are sensitive and which are not. This is less easy than it might seem, because several Android APIs are not or hardly documented. Furthermore, lists of APIs crafted by researchers get outdated with every new Android version.

2.3.1 PSCOUT

The first and the most straightforward approach to identify sensitive APIs was proposed by Au et al [9]. In their work authors presented a set of APIs that are governed by the Android permission setting. Prior to Android 6.0, when a user wants to install an application she has to make herself familiar with the list of permissions that the app needs and accept them. Starting from Android 6.0, Google introduces the concept of dynamic permissions that allows users to make their decision at runtime. Now permissions are not listed during the installation process but access to each protected API has to be granted whenever it is invoked. Moreover, users can grant only a subset of permissions while in older versions of Android a user has to accept or deny all of them. Denying an action prior to Android 5 leads to rejection of the app installation.

Nevertheless, the concept stays the same. Permissions have to be explicitly declared in an *AndroidManifest.xml* file and, if the application invokes an API protected by an unmentioned permission, Android will deny this action and throw an exception. These permissions are proxies for the sensitive behavior that the app will do. Typically, one permission is just a group of different APIs as there are multiple ways to perform the same task. Authors presented a tool called PSCOUT that performs analysis of Android platform and provides the mapping between permissions and the corresponding list of APIs. For example, *android.telephony.TelephonyManager: java.lang.String getDeviceId()* API maps to *READ_PHONE_STATE* permission which is responsible for revealing the unique identifier of an Android device. The full mapping is available at [45].

2.3.2 SUSI

As Rasthofer et al. [46] showed that not all sensitive data is protected by permissions. Authors proved that current approaches based on permission checks alone are inadequate because, contrary to the popular belief, permission checks are not a good indicator for the method relevance. There is still a chance to obtain the user-sensitive information without requesting an appropriate permission. Moreover, taint-analysis techniques, as will be discussed in Section 3.3, require categorization of sensitive APIs to sources (where information comes from) and sinks (where information goes to). They proposed SUSI, a novel machine-learning guided approach for identifying sources and sinks directly from the code of any Android API. They ran SUSI on Android 4.2 and found that there are a lot more sources and sinks than was previously known in the scientific literature. Therefore we leverage the SUSI framework by Rasthofer et al., which automatically classifies all methods in the whole Android API as sources, sinks or neither of them using a small hand-annotated fraction of the Android API to train a classifier. Besides providing a list of APIs that access sensitive data, SUSI also provides a categorization of these APIs listed in Table 2.3. For instance, the method *android.telephony.gsm.SmsManager: void sendTextMessage(...)* is a sink and falls into the *SMS_MMS* category and *android.location.Address: double getLongitude()* is a source and it belongs to the *LOCATION* category. The comprehensive list of the categorized sources and sinks can be found at the official web-site of the SUSI project [52].

Sources and sinks form the main concept in a data-flow analysis and therefore should be well defined. Rasthhofer et al. introduce the first and the most comprehensive definitions of these terms.

Source : *Sources are calls into resource methods returning a non-constant values into the application code.* For example, *android.location.Address: double getLongitude()* is a source because it returns non-constant value depending on the device location.

Sink : *Sinks are calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource.* For example, *android.telephony.gsm.SmsManager: void sendTextMessage(...)* is a sink because it accepts a telephone number as a first parameter and a text of the message as a third parameter.

Table 2.3: SUSI API categories of sensitive sources and sinks

Sources	Sinks	Shared
• HARDWARE_INFO	• PHONE_CONNECTION	• AUDIO
• UNIQUE_IDENTIFIER	• VOIP	• SMS_MMS
• LOCATION_INFORMATION	• PHONE_STATE	• CONTACT_INFORMATION
• NETWORK_INFORMATION	• EMAIL	• CALENDAR_INFORMATION
• ACCOUNT_INFORMATION	• BLUETOOTH	• SYSTEM_SETTINGS
• EMAIL_INFORMATION	• ACCOUNT_SETTINGS	• IMAGE
• FILE_INFORMATION	• SYNCHRONIZATION_DATA	• BROWSER_INFORMATION
• BLUETOOTH_INFORMATION	• NETWORK	• NFC
• VOIP_INFORMATION	• EMAIL_SETTINGS	
• DATABASE_INFORMATION	• FILE	
• PHONE_INFORMATION	• LOG	
• CONTENT_RESOLVER (*)	• INTENT (*)	
• NO_SENSITIVE_SOURCE (*)	• NO_SENSITIVE_SINK (*)	

(*) New category, see Section 3.3

2.4 Information flow control

Information flow control (IFC) is a technique to assert the security of a given program with respect to a given security policy. The classical security policy noninterference requires that the public output of a program should not be influenced by the secret input [25]. In the context of Android, it is interesting to track information flows between a sensitive *source* and a sensitive *sink*. In essence, given a *source* of information (e.g., a SQLite database containing the list of contacts) and a sink (e.g., a HTTP connection to a third party server), program analysis can tell whether this information flows to the specified sink.

Information Flow Control needs to track two types of flows [32]:

- *explicit flows*, which arise due to computations dependent on the values of their parameters;
- *implicit flows*, which arise from predicates that control the execution of certain code blocks.

Next we will discuss these types of flows in detail as well as the way how to track them.

2.4.1 Taint Analysis

Taint Analysis is a form of Information flow analysis. In the taint analysis all sources of information are marked as *"sensitive"* from the very beginning. This *"sensitive"* label is also called *taint*. Whenever a program variable in a code has a data assignment from or is influenced by the tainted variable, this variable also becomes tainted. Such taints are propagated from sensitive sources to sensitive sinks.

According to the method, the analysis can be propagated more than one taint. This may be helpful if not only the presence of a particular data flow should be reported, but also which data is exactly leaked. For example, let us consider the code in Listing 2.10. The *imei* variable on line 3 receives *UNIQUE_IDENTIFIER* taint and the *location* variable on line 6 receives *LOCATION* taint. Later on line 7 these variables are concatenated to the *leakedData* variable that is then sent via SMS. The *leakedData* variable obtains two taints after the concatenation that makes it possible to report that *LOCATION* and *UNIQUE_IDENTIFIER* data are leaked via SMS and not just that the taint is present.

```
1  public void foo(){
2        TelephonyManager telephonyManager = (TelephonyManager)getSystemService(
   Context.TELEPHONY_SERVICE);
3        String imei = telephonyManager.getDeviceId();
4        SmsManager smsManager =  SmsManager.getDefault();
5        LocationManager locationManager = (LocationManager) mContext.
   getSystemService(LOCATION_SERVICE);
6        String location = locationManager.getLastKnownLocation(
   LocationManager.NETWORK_PROVIDER);
7        String leakedData = imei + location;
8        smsManager.sendTextMessage("+49176000000", null, leakedData, null,
   null);
9  }
```

Listing 2.10: Example of merging two taints

Static taint analysis, on the one hand, examines a program without running it and reports all possible flows. On the other hand, it introduces false positives due to conditions that depend on runtime values. This means that it warns a user that an application under analysis leaks some sensitive information, but in reality the found taint paths are infeasible. This happens because some conditions cannot be resolved statically, and consequently the analysis over-approximates the results. The high false positive rate is not the only problem of static taint analysis. In order to achieve soundness, i.e., find all possible flows, the analysis may become complex and resource-expensive. Consequently it might incur scalability problems. Static analysis has also well-known limitations. Namely, it struggles with code reflection, code obfuscation and encryption. Code reflection as well as code obfuscation are quite common techniques to circumvent signature-based malware detection techniques. Moreover, most techniques implemented for Android do not support the analysis of native code. Native code (C/C++ code) is widely used in development of games in order to have better memory management and performance. It is thus necessary to support these features when analyzing Android applications.

Dynamic taint analysis tools like TaintDroid [19], only report data flows that have been observed during actual executions. The problem of the dynamic taint analysis tools is that they mainly depend on the set of inputs that have been used to generate executions. Therefore they can thus lack relevant information. Moreover, mobile devices have limited resources such as CPU, RAM, storage and battery life, and it is challenging to implement dynamic taint analyses on devices.

2.4.2 Explicit Flows

Explicit Flows are also called *Data Flows*. Data flow analysis propagates a source value through direct assignments of variables in the inter-procedural way, e.g., it propagates values through the parameters of a method and its return value. Let's consider the code in Listing 2.11. The code is triggered when a user clicks on a button in an application. *TelephonyManager* class provides access to the information about the telephony services on the device. In particular, an IMEI (International Mobile Equipment Identity) on line 3 is retrieved. According to the SUSI mapping (Section 2.3.2) this information is sensitive as it uniquely identifies the device and, thus, all actions in the app can be mapped to the particular device and be potentially used by unauthorized parties.

```
1  public void onClick(View view){
2      TelephonyManager telephonyManager = (TelephonyManager)getSystemService(
   Context.TELEPHONY_SERVICE);
3      String imei = telephonyManager.getDeviceId();
4      SmsManager smsManager =  SmsManager.getDefault();
5      String leakedData = getMessageToSend(imei);
6      smsManager.sendTextMessage("+49176000000", null, leakedData, null,
   null);
7  }
```

Listing 2.11: Example of leaking a Device Id through the SMS channel

Now the *imei* variable contains the sensitive IMEI number, i.e., it is tainted. Then it is passed to the method *getMessageToSend* as a parameter. The data flow propagator knows that the input parameter of this function is tainted and automatically assigns a taint to the *inputString* variable in the function *getMessageToSend* (refer to Listing 2.12). In turn, the *getMessageToSend* function just returns the substring of the *inputString* variable on a line 3. Therefore, its return value also becomes tainted.

```
1  private String getMessageToSend(String inputString) {
2      return inputString.substring(1,5);
3  }
```

Listing 2.12: Explicit Data Flow Example

Next, we go back to the *onClick* method of Listing 2.11 (see line 5). Now we know that the return value of the *getMessageToSend* method is tainted and the dataflow propagator automatically assigns a taint to the *leakedData* variable. Finally, the *leakedData* variable is passed to the *sendTextMessage* method as a parameter. The *sendTextMessage* method is responsible for sending SMS

messages and, thus, it performs a sensitive action. SUSI labels the *getDeviceId* source and the *sendTextMessage* as the *UNIQUE_IDENTIFIER* and the *SMS_MMS* categories, respectively. Thus, there is a flow of sensitive data in this example (refer to Table 2.4).

Table 2.4: Flow of Sensitive Data in Listing 2.11

TelephonyManager: java.lang.String getDeviceId()	⇝ SmsManager: void sendTextMessage(...)
UNIQUE_IDENTIFIER	⇝ SMS_MMS

2.4.3 Implicit Flows

However, Explicit Flows are easier to detect than Implicit Flows. Implicit Flows assume tracking of control-flow dependencies. It means that a taint should be assigned to a given variable according to the certain heuristic as no direct data assignments happen. Let us consider the previous example in Listing 2.11. But now the *getMessageToSend* method is more complicated than before, see Listing 2.13.

```
1  private String getMessageToSend(String inputString) {
2      StringBuilder message = new StringBuilder();
3      for(char character: inputString.toCharArray() ){
4          switch(character){
5              case '1':
6                  message.append('1');
7                  break;
8              case '2':
9                  message.append('2');
10                 break;
11             case '3':
12                 message.append('3');
13                 break;
14             case '4':
15                 message.append('4');
16                 break;
17             case '5':
18                 message.append('5');
19                 break;
20             case '6':
21                 message.append('6');
22                 break;
23             case '7':
24                 message.append('7');
25                 break;
26             case '8':
```

```
27                              message.append('8');
28                              break;
29                      case '9':
30                              message.append('9');
31                              break;
32                      case '0':
33                              message.append('0');
34                              break;
35              }
36      }
37      return message.toString();
38  }
```

Listing 2.13: Implicit Data Flow Example

According to taint analysis, the *inputString* variable in the method *getMessageToSend* is tainted. However, the return variable *message* does not accept any piece of data from it, i.e. there is no dataflow dependency between these two variables. Explicit Flow analysis will not assign a taint to the return variable *message* and, thus, will miss the flow. There is a leak of sensitive data, though. What the *getMessageToSend* does is simply splitting an IMEI into single characters, iterating over them and then performing some action. In this case an attacker has prior knowledge about the *inputString* variable and processes it smartly. She knows that the input variable consists of digits in the range of 0-9. Thus, she just implements a check for each value of the digit and appends the same value to the *message* variable. At the end, the *message* variable will have the same value as the *inputString*.

Taint analysis, which supports Implicit Flows, usually detects such control dependencies and reports that the *message* variable depends on the *inputString* variable. But it is likely to be a false positive alarm. Consider an example in Listing 2.14.

```
1   private String getMessageToSend(String inputString) {
2       String outputValue = "1";
3       if(inputString.equals("0000000000")){
4           outputValue = "1";
5       }
6       return outputValue;
7   }
```

Listing 2.14: False Positive of the Implicit Flow analysis

As we see on line 3, the *outputValue* depends on the value of the *inputString* variable. Therefore taint analysis will assign a taint to the *outputValue* variable. There is no leak of sensitive data here, though. The value of the *outputValue* will be always *"1"* independent of the value of the *inputString* variable.

Due to the runtime costs most of static taint analysis tools support only explicit flows.

2.5 Program Analysis

Program analysis is a process of automatically analyzing the behavior of a program with respect to selected criteria. People examine a piece of code to understand whether it satisfies a certain criterion, such as performance or correctness. Nearly everybody who has written a code performed program analysis.

Examining a small piece of code or a self-written application is not a hard task. But when a system grows and has many modules with the complex logic, manual investigation becomes difficult and almost infeasible. When it comes to analyzing thousands of programs, one should think about the way to automatize it. For this reason people develop frameworks, such as SOOT [35] and WALA [56], to perform such analysis fast and efficiently. The most of needed functionality is already implemented there, one just needs perform an analysis on the top of them.

2.5.1 SOOT analysis framework

SOOT [35] is the most well-known framework for analysis of Android and Java applications. It was initially designed as the Java-optimization framework, but nowadays people from all over the world use it for analyzing and instrumenting of Java and Android programs.

SOOT supports Java as well as Android binaries as an input. Then it transforms them into an Intermediate Representation (IR) called JIMPLE [54]. JIMPLE is the SOOT 's primary typed 3-address intermediate representation which is specifically designed for the bytecode transformation and analysis. SOOT supports lots of different analysis techniques and all of them are implemented using JIMPLE. Some of them are as follows:

- Call-graph construction
- Points-to analysis
- Def/use chains
- Template-driven Intra-procedural data-flow analysis
- Template-driven Inter-procedural data-flow analysis, in combination with heros [26]

- Taint analysis in combination with FLOWDROID [8].

The Figure 2.11 illustrates different packs of SOOT. First, it applies *jb* (JIMPLE body) pack, which is responsible for transforming bodies of all methods, to the JIMPLE IR. This phase is fixed and usually no custom analysis is built here.

After *jb* comes *cg*, which builds the call-graph based on entry points. *cg* applies to all methods in the app as well as to three next phases that start with the letter *w*. *wjtp* is the right place to perform analysis if the call-graph is needed (refer to Listing 2.15 for an example).

```
1  public static void main(String[] args){
2      PackManager.v().getPack("wjtp").add(new Transform("wjtp.
   exampleTransform", new SceneTransformer(){
3          @Override
4          protected void internalTransform(String phaseName,
5              Map<String, String> options) {
6              CallGraph callGraph = Scene.v().getCallgraph();
7          }
8      }));
9  }
```

Listing 2.15: Example of analysis on the top of *wjtp*

Next three phases are again applied to each method and its body. *jtp* is the usual pack to write the intra-procedural analysis which is based on the method body (refer to Listing 2.16 for an example).

```
1  public static void main(String[] args){
2      PackManager.v().getPack("jtp").add(new Transform("jtp.exampleTransform",
   new BodyTransformer(){
3          @Override
4          protected void internalTransform(final Body body, String phaseName,
   @SuppressWarnings("rawtypes") Map options) {
5              final PatchingChain<Unit> units = body.getUnits();
6              for(final Iterator<Unit> iter = units.snapshotIterator(); iter.
   hasNext();) {
7                  final Unit u = iter.next();
8              }
9          }
10     }));
11 }
```

Listing 2.16: Example of analysis on the top of *jtp*

Every transformer added to the whole Jimple pack must be a *SceneTransformer*, otherwise it must be a *BodyTransformer*. Scene Transformer is fired once a Call Graph is present. By using it developers can traverse the Call Graph and perform the desired analysis. Body Transformer, in turn, is fired for each method in each class and is primarily used to alter the code.

Before considering the last two phases, it should be mentioned that SOOT supports not only loading and analyzing the code, but also writing it back. As we already discussed, SOOT was initially designed as an optimization framework and, thus, it has the capability to write the optimized code back to the binaries. The last phases are responsible exactly for writing the code back. Finally, SOOT runs *bb* and *tag* packs. The former converts JIMPLE bodies into Baf bodies, i.e., a stack based intermediate representation, from which SOOT creates bytecode. The latter aggregates certain tags, such as a line number, to gain uniqueness of the code.

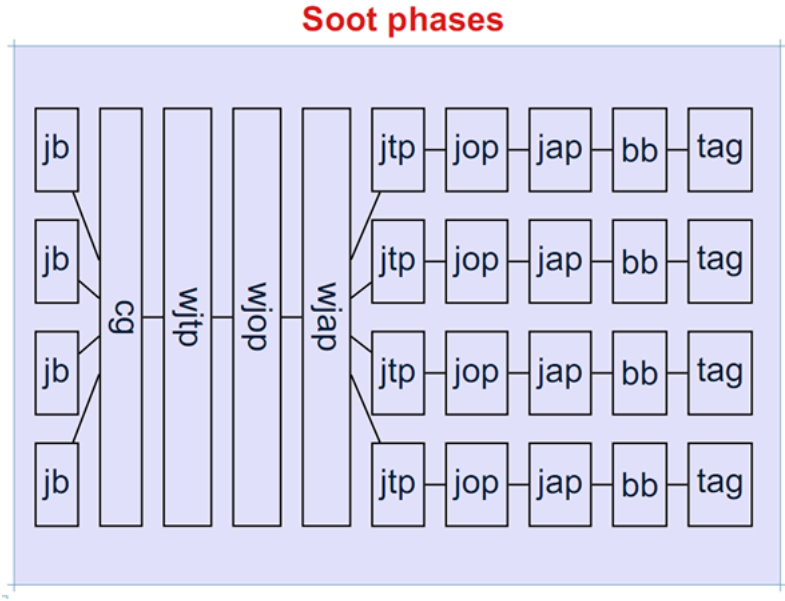

Figure 2.11: SOOT phases [49]

Chapter 3

Mining Sensitive APIs and Dataflows

3.1 Related Work

As mobile devices are a particularly rich store of sensitive data, it is not surprising that that a lot of research on mobile security resulted in taint analysis techniques for detecting information leaks. Among the research works that leverage *dynamic* taint analysis, TAINTDROID is de-facto the state-of-the-art tool for Android applications [19]. Thanks to the efficient instrumentation of the Android execution environment, TAINTDROID can report information leaks without any false positives in apps even when they involve native code.

Dynamic taint analysis has the obvious limitation of reporting only on what has been observed during a limited set of executions. On the contrary, *static* taint tracking tools report any information leak that *may* occur at runtime. The FLOWDROID tool, which will be described in Section 3.3, employs the highly precise static control and data flow analysis of Android apps to report both explicit and implicit information flows [8]. Other static taint tracking tools work in a similar fashion, but they miss several possible information flows since they implement less precise data-flow analysis [62, 22].

Other techniques focus on detecting information flows involving inter-applications communication. Thus, they can detect when multiple applications act together to leak sensitive information [33, 36].

The intention of the approach described in this Chapter is orthogonal to all these techniques. In fact, while they can detect whether there is any information flow in Android applications, they cannot tell whether such behavior is likely to be malicious or not. At the same time, our approach has no ability to detect information flows on its own, and therefore extends FLOWDROID to collect information regarding the behavior of apps.

3.2 Mining Sensitive APIs

Applications communicate with ANDROID system by using Application Programming Interfaces (APIs). Sensitive APIs are the APIs that access or operate with sensitive information. We already discussed ways to find sensitive APIs in Section 2.3. SAFAND uses the set of SUSI APIs (Section 2.3.2) as it is more comprehensive and comes with categorization of every API.

The algorithm of mining sensitive APIs is illustrated in Algorithm 1 and works as follows:

1. as ANDROID ecosystem is event-based, each callback and event is an entry point. In order to take all of them into account SAFAND builds a `dummyMainMethod` by using the functionality of FLOWDROID [8]. FLOWDROID provides a possibility to build a highly precise dummy main method, which includes emulations of all kinds of callbacks and events;

2. it builds the Control Flow Graph by using SOOT and a CHA [17] algorithm. CHA algorithm is a default algorithm of SOOT framework.. In order to build a Call Graph an entry point is required. The entry point is used to identify the starting point of the analysis. SAFAND passes the `dummyMainMethod` from the previous step;

3. it calculates the set of reachable methods from the current entry point and the call graph by iterating over its edges;

4. it identifies the set of used sensitive APIs by comparing them with the set of reachable methods;

5. it runs post-processors which will be described in Section 3.2.1 and Section 3.2.2.

Algorithm 1 Strategy of mining sensitive APIs.

Require: Application APP, Sensitive APIs $SAPIS$
1: **procedure** MINESENSITIVEAPIS(APP, $SAPIS$)
2: $DMM \leftarrow$ BUILDDUMMYMAINMETHOD(APP)
3: $FOUND\_APIS \leftarrow$ NEW LIST
4: $CG \leftarrow$ BUILDCALLGRAPH(APP)
5: $RMETHODS \leftarrow$ CALCULATEREACHABLEMETHODS(CG, DMM)
6: **for** $RMETHOD$ in $RMETHODS$ **do**
7: **if** $RMETHOD$ in $SAPIS$ **then**
8: ADDTOLIST($RMETHOD$, $FOUND\_APIS$)
9: **end if**
10: **end for**
11: RUNPOSTPROCESSORS($FOUND\_APIS$, $RMETHODS$)
12: **end procedure**

3.2.1 Mining Sensitive Resources from an Application

During our investigation, we found that Android apps also access sensitive resources through *content providers*—external components that resolve appropriate resource identifiers. A *CalendarContract* provider, for instance, can access calendar data. All these flows start from the *android.content.ContentResolver* API, which gets the desired resource identifier as an argument.

Listing 3.1 shows example of a typical access to the user Address Book. It is realised by using `ContentResolver:query` method. The type of data access strongly depends on its first argument, e.g. `android.net.Uri`. `ContactsContract.Contacts.CONTENT_URI` tells the system that the Address Book should be accessed, while `Telephony$Sms$Inbox:android.net.Uri CONTENT_URI`, for example, is responsible for accessing sms messages in the inbox.

```
1  public String readContact(){
2        ContentResolver cr = getContentResolver();
3        Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI,null, null,
   null, null);
4        if (cur.getCount() > 0) {
5              while (cur.moveToNext()) {
6                    return cur.getString(cur.getColumnIndex(ContactsContract.
   Contacts.DISPLAY_NAME));
7              }
8        }
9        return null;
10 }
```

Listing 3.1: Example of the access to Sensitive Resources

SUSI assigns the *ContentResolver* API to NO_CATEGORY because the same API can be used to access all sorts of resources, sensitive or non-sensitive. In 2012, however, Au et al. [9] published a list of sensitive resource schemes used in Android. We therefore conducted an additional step of static analysis: using SOOT [35], we have implemented a postprocessor of results, extracted *android.net.URI* usages from found APIs and assigned them to the appropriate SUSI source categories. Any resource usage which is not in the list would be classified into the CONTENT_RESOLVER sensitive source category. The precise algorithm of resolving URIs is stated in Algorithm 2.

The actual implementation of URI resolving algorithm can be found on the official GitHub page of the project [41].

Limitations

Custom URIs Access to sensitive resources is not limited to the set of predefined URIs that come with Android. People can come up with their own values and their are out of scope of this work.

Complex string operations with URIs Developers can also create URIs by using `Uri:parse` method as it is shown below:

Uri uri = Uri.parse("content://downloads/public_downloads")

But sometimes they can concatenate the passed string before as it is shown in Listing 3.2.

Algorithm 2 Strategy of resolving URI values.

Require: Caller method CM, URI register REG, Current unit CU, Call graph CG

```
1: procedure resolveUri(CM, REG, CU)
2:     D ←GetImmediateDominator(CU,CM)
3:     while D is not NULL do
4:         if D is AssignStmt then
5:             L ← getLeftOp(D)
6:             if L == REG then
7:                 R ← getRightOp(D)
8:                 if R is VAR then
9:                     REG = R
10:                else if R is URI then
11:                    return R
12:                else if R is InvokeExpr then
13:                    if R is URI_PARSE then
14:                        return getParameterValue(R,0)
15:                    end if
16:                    return resolveReturnValueInInvocation(U)
17:                end if
18:            end if
19:        else if D is the input parameter then
20:            PNUM ← getParamNumber(D)
21:            return findValueInCallers(PNUM,CM)
22:        end if
23:        D ←GetImmediateDominator(D,CM)
24:    end while
25: end procedure
26: procedure resolveReturnValueInInvocation(U)
27:    CE ← getMethodsOutOf(CG,U)
28:    for Edge in CE do
29:        RET ← findReturnStatement
30:        CM ← getMethodOf(U)
31:        ARG ← getReturnReg(RET)
32:        URI ← resolveUri(CM,ARG,RET)
33:        if URI is not NULL then
34:            return URI
35:        end if
36:    end for
```

```
37: end procedure
38: procedure FINDVALUEINCALLERS(PNUM,CM)
39:     CE ← GETMETHODSTO(CG,CM)
40:     for Edge in CE do
41:         ARG ← GETARGUMENTREG(PNUM,Edge)
42:         CU ← GETSOURCEUNIT(Edge)
43:         CM ← GETMETHODOF(CU)
44:         URI ← RESOLVEURI(CM,ARG,CU)
45:         if URI is not NULL then
46:             return URI
47:         end if
48:     end for
49: end procedure
```

```
1         String stringUri = String.format(“%s://%s/%s”,“content”,“downloads
  ”,“public_downloads”);
2         Uri uri = Uri.parse(stringUri)
3
```

Listing 3.2: Complex string operations with URI

In this case our prototype can extract all string items, but resolving the mask is out of scope of this work. The worst scenario would be here is when individual string components came from dynamic operations such as networking or accessing a file. Static analysis cannot resolve such cases by its nature.

3.2.2 Mining Intercomponent Communications inside an Application

We found that a huge number of sensitive APIs are responsible for communication between multiple apps of app components (“Intent” in Android parlance). Intents are the single communication channels between components and apps. Therefore they are used for storing data that should be passed to the target activity. In Line 9 of Listing 3.3 a developer passes the message *“Value for the second screen”* to the `SecondActivity` class, which in turn sends it out.

```
1  public class FirstActivity extends Activity{
2      public void startMaps(View view){
3          String latitude = ‘‘49.254696’’;
4          String longtitude = ‘‘7.040594’’;
5          String uri = String.format(Locale.ENGLISH, ‘‘geo:%f,%f’’, latitude,
   longitude);
6          Intent intent = new Intent(android.content.Intent.ACTION_VIEW,
7          Uri.parse(uri));
8          startActivity(intent);
9      }
10     public void readAndGo(View view){
11         Intent startNew = new Intent(this, SecondActivity.class);
12         startNew.putExtra(”myKey”, ”Value for the second screen”);
13         startActivity(startNew);
14     }
15 }
16
17 public class SecondActivity extends Activity{
18     @Override
19     public void onCreate(Bundle savedInstanceState){
20         ...
21         Intent intent = getIntent();
22         inputStr = intent.getStringExtra(”myKey”);
23         sendMeSomewhere(inputStr);
24         ...
25     }
26 }
```

Listing 3.3: Example of using of explicit and implicit intents

Listing 3.3 shows an example of typical usages of intents. There are two different kinds of intents: explicit and implicit.

Explicit Intents Explicit intents declare the target receiver *explicitly* in the code. In Line 12 of Listing 3.3 an explicit intent is defined by providing the class `SecondClass` as the target component. Explcit intents can be sent to the classes of the same application to avoid security flaws.

Implicit Intents Impicit intents are all intents that go through the Android system. Whenever an app sends such intent, it goes to the Android system, which, in turn, performs routing and resolving a set of the target apps. If there is more than one app which can receive such intents, the Android system displays a dialog and asks a user to select the desired app. Line 6 of Listing 3.3 shows the way how to open some app that can show a map with the desired position. The developer does not care about the target

Table 3.1: Signatures of methods that start new activity

android.content.Context:	void startActivity(...)
android.content.Context:	void startActivityForResult(...)
android.content.Context:	void startActivityFromChild(...)
android.content.Context:	void startActivityFromFragment(...)
android.content.Context:	void startActivityIfNeeded(...)

Table 3.2: Signatures of methods that define important properties of intents

android.content.Intent:	android.content.Intent setData(android.net.Uri)
android.content.Intent:	android.content.Intent setAction(java.lang.String)
android.content.Intent:	android.content.Intent setClassName(android.content.Context, java.lang.String)
android.content.Intent:	android.content.Intent putExtra(java.lang.String,java.lang.String[])
android.content.Intent:	android.content.Intent putExtras(android.os.Bundle)
android.content.Intent:	android.content.Intent setComponent (android.content.ComponentName)

app, he just wants to show the position on the map. The Android system is fully responsible for showing it.

Our analysis tracks both types of intents and extracts all needed properties as *action*, *URI*, *data*, *extras* and *target class* to distinguish them. In MUDFLOW, however, we do not use this information and assign all intents to `INTENT` category.

Let us go step by step through the precise algorithm of the Intent resolution (Algorithm 3). First of all, given sinks of dataflows, it checks for occurrences of special methods that are responsible for launching new activity (Table 3.1). When the algorithm reaches these methods, it identifies the value of `android.content.Intent` parameter and saves it. Next, it iterates backward through the method body and tries to find invocations of the methods that are responsible for assigning the class name or URI, for example. The full list of such methods can be found in Table 3.2.

During the analysis, it is checked that assignments are performed to a correct instance of the `android.content.Intent` class and bound to the gathered values. `setAction`, `setClassName` and `putExtra` methods have `java.lang.String`

Table 3.3: Signatures of Intent constructors

android.content.Intent:	void ⟨init()⟩
android.content.Intent:	void ⟨init(java.lang.String)⟩
android.content.Intent:	void ⟨init(java.lang.String,android.net.Uri)⟩
android.content.Intent:	void ⟨init(android.content.Context,java.lang.Class)⟩
android.content.Intent:	void ⟨init(java.lang.String,android.net.Uri, android.content.Context,java.lang.Class)⟩

Algorithm 3 Strategy of gathering data on the intent.

Require: Dataflows D, List of Signatures from Table 3.1 SA, List of signatures from Table 3.3 IC

```
1: procedure searchForNewActivity(D)
2:     for S in getSink(D) do
3:         if S in SA then
4:             IREG ←extractIntentParameter(S)
5:             I ←gatherIntentInformation(S,IREG)
6:             enrichSinkWithIntentInformation(S,I)
7:         end if
8:     end for
9: end procedure
10: procedure gatherIntentInformation(S,IREG)
11:     I ← new instance of IntentInformation
12:     M ←GetMethodOf(S)
13:     U ←GetUnitOfMethod(S,M)
14:     D ←GetImmediateDominator(U,M)
15:     while D is not NULL do
16:         if !ContainsInvokeExpression(D) then; continue;
17:         end if
18:         if ReachedConstructorOfIntent(D,I) then break;
19:         end if
20:         if processExtras(D,I) then
21:             D ←GetImmediateDominator(D,M); continue;
22:         end if
23:         if processAction(D,I) then
24:             D ←GetImmediateDominator(D,M); continue;
25:         end if
26:         if processClass(D,I) then
27:             D ←GetImmediateDominator(D,M); continue;
28:         end if
29:         if processData(D,I) then
30:             D ←GetImmediateDominator(D,M); continue;
31:         end if
32:         if processComponent(D,I) then
33:             D ←GetImmediateDominator(D,M); continue;
34:         end if
35:         D ←GetImmediateDominator(D,M);
```

```
36:     end while
37:     return I
38: end procedure
39: procedure ReachedConstructorOfIntent(D,I)
40:     DM ← getMethodOf(D)
41:     if DM in IC then
42:         switch DM do
43:             case INIT_DEFAULT
44:                 return true
45:             case INIT_WITH_ACTION
46:                 A ←ExtractAction(D)
47:                 EnrichIntentWithAction(A,I)
48:                 return true
49:             case INIT_WITH_ACTION_URI
50:                 A ←ExtractAction(D)
51:                 EnrichIntentWithAction(A,I)
52:                 URI ←ExtractUri(D)
53:                 EnrichIntentWithUri(URI,I)
54:                 return true
55:             case INIT_CONTEXT_CLASS
56:                 CON ←ExtractContext(D)
57:                 EnrichIntentWithContext(CON,I)
58:                 CL ←ExtractClass(D)
59:                 EnrichIntentWithClass(CL,I)
60:                 return true
61:             case INIT_WITH_ACTION_URI_CONTEXT_CLASS
62:                 A ←ExtractAction(D)
63:                 EnrichIntentWithAction(A,I)
64:                 URI ←ExtractUri(D)
65:                 EnrichIntentWithUri(URI,I)
66:                 CON ←ExtractContext(D)
67:                 EnrichIntentWithContext(CON,I)
68:                 CL ←ExtractClass(D)
69:                 EnrichIntentWithClass(CL,I)
70:                 return true
71:     end if
72:     return false
73: end procedure
```

```
74: procedure ProcessExtras(D,I)
75:     DM ← getMethodOf(D)
76:     if DM is PUT_EXTRA then
77:         E ← ExtractExtra(D)
78:         EnrichIntentWithExtra(E,I)
79:         return true
80:     else if DM is PUT_EXTRAS then
81:         ELIST ← ExtractExtrasBundle(D)
82:         for E in ELIST do
83:             EnrichIntentWithExtra(E,I)
84:         end for
85:         return true
86:     end if
87:     return false
88: end procedure
```

```
89: procedure ProcessAction(D,I)
90:     DM ← getMethodOf(D)
91:     if DM is SET_ACTION then
92:         AC ← ExtractAction(D)
93:         EnrichIntentWithAction(AC,I)
94:         return true
95:     end if
96:     return false
97: end procedure
```

```
98: procedure ProcessClass(D,I)
99:     DM ← getMethodOf(D)
100:    if DM is SET_CLASS_NAME then
101:        C ← ExtractClass(D)
102:        EnrichIntentWithClass(C,I)
103:        return true
104:    end if
105:    return false
106: end procedure
```

```
107: procedure ProcessData(D,I)
108:     DM ← getMethodOf(D)
109:     if DM is SET_CLASS_NAME then
110:         URI ← ExtractUri(D)
111:         EnrichIntentWithUri(URI,I)
112:         return true
113:     end if
114:     return false
115: end procedure
116: procedure ProcessComponent(D,I)
117:     DM ← getMethodOf(D)
118:     if DM is SET_CLASS_NAME then
119:         DT ← ExtractComponent(D)
120:         EnrichIntentWithComponent(DT,I)
121:         return true
122:     end if
123:     return false
124: end procedure
```

parameters or key-value pairs of them as in `putExtra`. Therefore we perform the String propagation which will be discussed in detail in Chapter 4. `setData`, in turn, has `android.net.Uri` argument and therefore we perform the URI propagation as discussed in Algorithm 2.

In this section we will describe the algorithm of propagating `android.os.Bundle` value as it is used in `putExtras` method. In essence `android.os.Bundle` is a list of key-value pairs. The key-value pair can be anything, but we support only `java.lang.String` for performance reasons. The problem of resolving extras is reduced to propagating strings corresponding to the key and the value in the pair. However, the value in the pair does not give a lot of information and is very specific to the app and the activity. Therefore we collect only keys.

Another way to determine the target activity is to use a component instead of a class (e.g. `android.intent.Intent:setComponent`). In this case, a developer should pass the instance of `android.content.ComponentName` class, which gives the possibility to identify the target class by providing its package and its name. The analysis algorithm for the instantiations of `ComponentName`, resolves the name of the package and the class and merges them together. Again, the problem is reduced to the string propagation.

The actual implementation of Intent resolving algorithm can be found on the official GitHub page of the project [40].

Limitations

Variety of data types in extras `android.os.Bundle` is a key-value pair collection that supports data types derived from `java.lang.Object`, as well as all primitive types. In the current work for simplicity we concentrate only on the case when the value if `java.lang.String`. The program analysis presented in the thesis is used for detecting anomalies among thousands of apps. Thus, apps should have a set of comparable features.

Custom URIs As we discussed in Section 3.2.1, developers can create their own custom URIs, but we only support the predefined set of default URIs defined by ANDROID.

3.3 Mining Sensitive Data Flows

SAFAND internally uses FLOWDROID [8] static taint tracking tool to extract sensitive data flows from Android apps. We chose a FLOWDROID because it is a state-of-the-art tool for Android applications (Section 2.4.1). Unlike standard Java programs Android apps do not have a single entry point (e.g. the static main method), which poses an important question of how to handle this scenario. Android has the event-based nature (Section 2.1.3). Therefore to achieve high-precision analysis it is required to take into account the order of their execution (e.g., onPause can not be executed before onCreate or onResume).

FLOWDROID provides this functionality out-of-the-box and accurately models the lifecycle of Android applications and their interactions with the Android OS. Callbacks in Android apps can be registered both statically and dynamically (refer to Section 4.8 for more details). Static callbacks are defined in the XML layout files that describe the layout of the corresponding activity (e.g., screen). Reusable components are fundamental features of the Object Oriented Design(OOD) and, thus, developers should have a possibility to redefine a callback for a particular purpose. For example, one could develop a custom alert dialog with *OK* and *Cancel* buttons, reuse it within the whole app and set the custom *onClick* action in each particular case. This can only be done by dynamically redefining callbacks. FLOWDROID first analyses the app for registered components and callbacks and then creates a *dummy main method* that

simulates these interactions with the operating system. Such workflow causes the static analysis to analyse behavior during the runtime correctly.

FLOWDROID provides the highly precise taint analysis that supports the following principles:

flow-sensitive: a flow-sensitive analysis is aware of the order of program statements.

context-sensitive: according to this principle, when analyzing a target of the function call, one keeps track of the calling context. This makes it possible to reduce the noise and always return to the target caller instead of assuming all possible call-sites of the method.

object-sensitive: This is a context-sensitive approach that distinguishes invocations of methods made on different objects. It is a very important problem in object-oriented languages as child classes usually redefine the behaviour of a superclass. Thus, object-sensitivity makes the analysis more precise.

An interested reader could refer to the following source [37] to better understand these terms. High precision is a very important requirement for machine learning approaches like the MUDFLOW to reduce the false-positive (missed malware) rate and the noise of data.

FLOWDROID uses the instantiation of IFDS framework by Reps and Horwitz [47] in order to reduce the data-flow problem to the graph reachability problem, where nodes represent combinations of possible facts about the program. If one fact is derived from another one, then these facts are connected by the appropriate edge in the graph. If a certain fact at a sink is reachable from the source node than the analysis reports the dataflow between them.

Listing 3.4 shows an example of leaking a unique device id by sending it via SMS. In this example the variable *imei* on line 3 forms a root of the graph. It is connected to the node that models "*someString* is tainted" due to the normal forward propagation. When accessing the *resolve* method, the algorithm creates an invocation edge to the callee causing the input parameter *str*(line 10) to be tainted. The method return value becomes tainted causing the variable *message*(line 6) to be also tainted. As the *message* variable is used later for sending an SMS it automatically becomes a sink. *getDeiviceId* method is transitively reachable from this sink and, thus, the analysis reports a flow of sensitive data. Context-sensitive analysis makes it possible to distinguish between different method calls to *resolve* methods with different parameter values.

Context-insensitive analysis would act conservative considering all return sites of the *resolve* method tainted even with benign parameter values.

FLOWDROID is not a contribution of this thesis, we therefore omit a lot of details and kindly point an interested reader to the original FLOWDROID paper [8].

```
1  public void onClick(View view){
2        TelephonyManager telephonyManager = (TelephonyManager)getSystemService(
   Context.TELEPHONY_SERVICE);
3        String imei = telephonyManager.getDeviceId();
4        String someString = imei;
5        SmsManager smsManager =  SmsManager.getDefault();
6        String message = resolve(someString);
7        smsManager.sendTextMessage("+49176000000", null, message, null, null);
8  }
9
10 private String resolve(String str){
11       return String.format(''Device Id :%s'', str);
12 }
```

Listing 3.4: Example of data leak in Android

In order to perform a taint analysis one should define methods that are responsible for obtaining sensitive data and which—for leaking them out. More precisely, FLOWDROID requires a categorized list of sources and sinks as an input for the analysis. In Section 2.3 we discussed the possible ways to obtain them. As it is staved in Section 2.3.2, SUSI approach by Rasthofer et al.[46] automatically classifies methods to sources and sinks. Moreover, it assigns them meaningful labels listed in Table 2.3.

In addition to the originally published SUSI categories, we created three new categories to further break down the behavior of Android apps, which are marked with (*) in Table 2.3.

3.3.1 Dealing with Advertisement Frameworks

Free Android apps generate revenue through advertisements, which are delivered by specific *advertising frameworks.* These frameworks access sensitive data such as account data to deliver personalized advertisements. However, they are separate from the actual app code and do not describe functionality of an application. As advertising frameworks are frequently used, their behavior thus becomes "normal" and makes malicious behavior harder to detect. Furthermore, malicious software may use an advertisement framework to justify and mask its suspicious behavior (Figure 3.1).

Assuming that advertisement frameworks are to be trusted, SAFAND ignores all sensitive flows taking place within advertisement frameworks, allowing to focus on the actual app code. Table 3.4 shows a list of frequently used frameworks where flows are excluded in SAFAND. Currently SAFAND is unable to detect such advertisement frameworks in presence of code obfuscation. This problem is in the scope of our future work and will be discussed in Chapter 6. We kindly point an interested reader to the recent work in the field of library detection under code obfuscation [14].

Table 3.4: Ad frameworks where flows are excluded

com.admob.android	com.adsdk.sdk
com.adsmogo	com.aduwant.ads
com.applift.playads	com.google.ads
com.inneractive.api.ads	com.mopub.mobileads
com.revmob.ads	com.smartadserver.android
com.swelen.ads	de.selfadservice

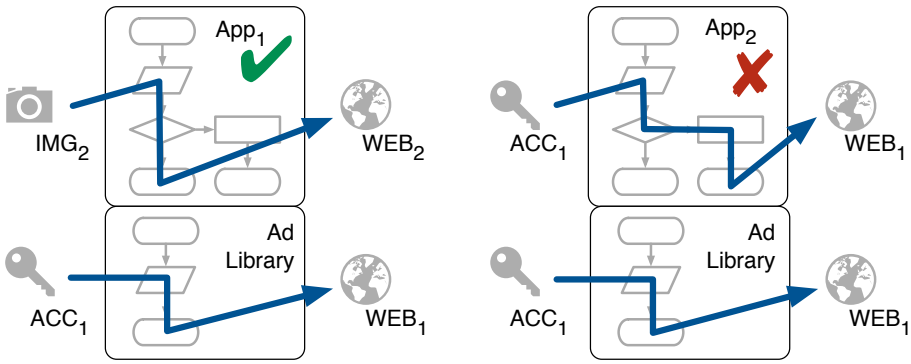

Figure 3.1: Noise induced by ad libraries

3.3.2 Emulating Non-sensitive Sources and Sinks

Almost all applications in Google Play Store access sensitive sources. However, the accessed data does not necessarily end up in a sensitive sink. For instance, wallpaper apps access user's images as sensitive sources, but the user's display is not a sensitive sink. To leverage such access patterns, every used source that does not flow into a sensitive sink is modelleded as a flow every source from

that source "to" the special category NO_SENSITIVE_SINK. Similarly, we modeled a flow that does not start from a sensitive source and ends up in a sensitive sink from the special category NO_SENSITIVE_SOURCE. The precise algorithm is explained in Algorithm 4. To avoid ambiguous cases when there are multiple API invocations of the same method, we always compare instances of SOOT `Stmt` which represents unique lines of code. Thus, if the app has three invocations of `TelephonyManager:getDeviceId` and one instance has a flow to the sensitive sink, only two other instances will have a flow to NO_SENSITIVE_SINK and not this one.

Algorithm 4 Strategy of processing non-sensitive sources and sinks.

Require: Dataflows D, List of Sources from the app SO, List of sink from the app SI

```
1:  procedure EnrichDataflowsWithNonSensitiveSoSi(D,SO,SI)
2:      for S in SO do
3:          if S not in getSources(D) then
4:              addNewFlow(S,NO_SENSITIVE_SINK)
5:          end if
6:      end for
7:      for S in SI do
8:          if S not in getSinks(D) then
9:              addNewFlow(NO_SENSITIVE_SOURCE,S)
10:         end if
11:     end for
12: end procedure
```

3.3.3 Dataflow Representation

Applied on a single *app*, SAFAND uses FLOWDROID to extract all data flows from *all sensitive data sources* to *all sensitive data sinks*(Figure 3.2). The result is a set of pairs that characterizes the sensitive flows in the application—and thus the application itself:

$$Flows(app) = \{source_1 \rightsquigarrow sink_1, source_2 \rightsquigarrow sink_2, \ldots\}$$

where each $source_i$ and $sink_i$ are sensitive Android API methods. (Again, "sensitive" means that a method falls into one of the SUSI categories listed in Table 2.3). As examples of such flows, consider Table 3.7 and Table 3.8 discussed in Section 3.4.5.

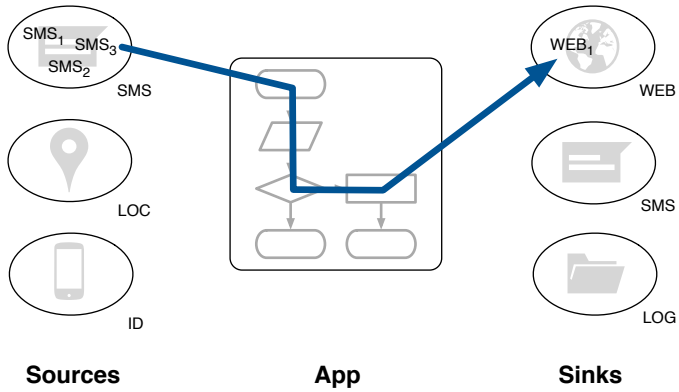

Figure 3.2: Dataflows produced by FLOWDROID

By default, each source and sink contain a full method name and a signature. For the sake of coarser granularity, this information can be shortened, allowing for multiple sources and sinks to be aggregated. SAFAND supports the following three granularity levels, from finest to most coarse:

Method. This is the full method signature—for instance, *LocationManager.requestLocationUpdates(...)*. Table 3.7 shows the flows in Android Twitter app at the *method* level.

Class. Considering only the class name (*LocationManager*) allows to express flows between classes rather than methods. This treats all methods of the class uniformly.

Category. Considering only the SUSI category of the API (LOCATION_INFORMATION) allows to express flows between categories. This is the coarsest way of expressing flows, yet one that could be made accessible to end users. Table 3.5 shows the flows in the Android Twitter app at the category level. Here, it is indeed easy to spot how the sensitive data is used.

The granularity level of features is a very important aspect of machine learning techniques. If one wants to compare multiple apps among each other it is a quite important to have as many common features as possible. In the context of API usage it means that the *method* granularity level can introduce a huge number of features and it would be hard to find commonalities there. It is known that there are many ways in programming to achieve the same goal.

Rasthofer et al in their SUSI work showed that there are multiple APIs in Android that are actually performing the same task [46]. Thus, if two apps use different APIs to perform the same task, the *method* granularity level will not group them together. Therefore there is no *golden hammer* solution on selecting the appropriate level of the granularity and it mainly depends on the goal of the task.

Table 3.5: Flows in Android Twitter App, by SUSI category

ACCOUNT_INFORMATION	⇝	SYNCHRONIZATION_DATA
ACCOUNT_INFORMATION	⇝	ACCOUNT_SETTINGS
ACCOUNT_INFORMATION	⇝	INTENT
ACCOUNT_INFORMATION	⇝	LOG
NETWORK_INFORMATION	⇝	INTENT
NETWORK_INFORMATION	⇝	LOG
DATABASE_INFORMATION	⇝	LOG

3.4 Evaluation

To evaluate the data flow module we use experiments from the MUDFLOW paper [12]. As mentioned before, MUDFLOW internally uses SAFAND dataflows module, which, in turn, leverages FLOWDROID [8].

MUDFLOW is based on the idea that having the instant access to a sufficiently large set of *benign* apps it is possible to automatically detect *suspicious* apps with no prior knowledge about them. The assumption is that all benign apps are similar to each other in terms of usage of sensitive data and, thus, suspicious apps should be dissimilar to them. Checking for dissimilarity is not the same as checking for similarity. Each app is written in its unique style and uses quite unique code fragments. The code by itself can not be used for establishing dissimilarity. We introduces the term *usage of sensitive data* as a feature to compare apps. Of course, a messaging app is not similar to a flashlight app in terms of a usage of sensitive data, thus we also introduces the notion of *context* that better describes a group of apps.

3.4.1 Apps Mined

The distribution of apps in MUDFLOW is shown in Figure 3.3.

Benign apps We mined 2950 apps from the Google Play Store. More precisely—we took top 100 apps based on the number of downloads from 30 Google Play categories as of March 2014. We treat these apps as *benign* as they come from the trusted market. Indeed, the hypothesis that all apps from the Google Play store is benign is not fully correct. There is still a chance to have some malware samples there but the probability is a quite low.

Malicious apps For the *malicious* set of apps we used two different sources: MalGenome [63] and VirusShare [55] datasets that contain 1260 and 24,317 apps respectively

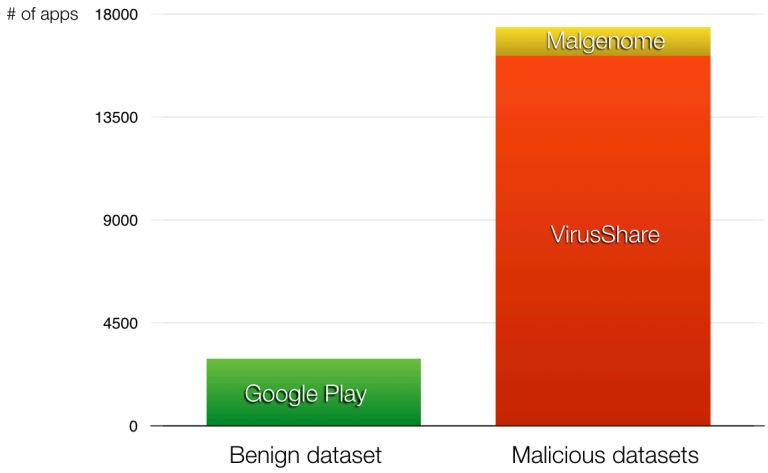

Figure 3.3: Dataset of apps used in experiments

3.4.2 Analysis Settings

Running a precise static taint analysis on real-world applications is challenging. In favor of faster analysis or the ability to analyze a larger application, which would otherwise not fit in memory, we used the following FLOWDROID settings [8]:

- *No flow across intents.* Android apps use special components, which are called *intents,* to implement messaging between components, in particular to start activities or provide services in the background. We do not track flows across intents; when sensitive data is sent to an intent, the flow is marked with the INTENT category as a sink;
- *Explicit flows only.* Our static taint analysis settings consider neither conditionals controlling specific flows, nor the flows leading to these conditionals. This is in contrast to *information flow analysis,* which also takes such implicit flow into account;
- *Flow-insensitive alias search*, which may generate false positives and greatly reduces runtime for large applications;
- *Maximum access path length of 3,* again possibly reducing precision with respect to the default setting of 5;
- *No-layout mode,* ignoring Android GUI components, such as input fields, as data flow sources;
- *No static fields,* ignoring the tracking of static fields.

All these choices sacrifice some amount of precision for speed and memory. As a result, the list of flows determined by MUDFLOW can have false positives, e.g., am example, flows that are infeasible during executions as well as false negatives, e.g., am example, missing flows that actually might be possible. But still, FLOWDROID is much more precise than the basic object- or context-insensitive data flow analysis. As ever when applying precise static analysis on real-world programs with finite time and resources, striking the good balance between false positives and false negatives is an important challenge. Let us remind at this point, that our goal is *to detect anomalies* and not to prove the presence or absence of flows. Thus, we can tolerate imprecision as long as the overall results are fine.

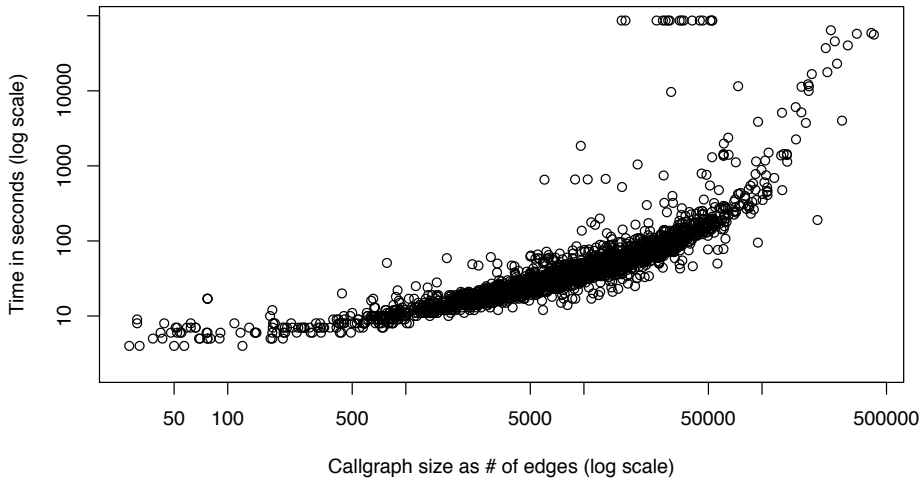

Figure 3.4: Analysis time of benign applications with respect to their callgraph sizes. All time measurements obtained on an Intel 64-core machine with 730 GB RAM.

Still, let us state what "finite time and resources" mean in our setting, and why compromises are badly needed. The main machine we used to run MUDFLOW was a compute server with 730 GB of RAM and 64 Intel Xeon CPU cores, far exceeding the standard memory sizes of today's personal computers. Even with all the compromises listed above, the server sometimes used all its memory, running on all cores for more than 24 hours to analyze *one single* Android app, as shown in Figure 3.4. Overall, we had this machine run for two months without interruption to extract data flows from Android applications.

Out of the 2,950 "benign" apps, 84 (3%) were not analyzable: 16 apps exceeded the RAM limit of 730 GB or the 24-hour timeout, and 68 apps caused a SOOT exception while transforming DEX bytecode to JIMPLE representation. Of the "malicious" apps, 10,239 (40%) were not analyzable because of corrupted or incomplete APKs; most frequently, the required Android manifest was missing. We also removed all such non-analyzable apps from our dataset. This resulted in final datasets of 2,866 "benign" apps and 15,338 "malicious" apps (Figure 3.3).

Table 3.6: Data flows in benign (left) and malicious (right) applications, by SUSI categories

Source	LOG	INTENT	NETWORK	FILE	SYSTEM_SETTINGS	ACCOUNT_SETTINGS	SMS_MMS	AUDIO	NFC	SYNCHRONIZATION_DATA	CALENDAR_INFORMATION	LOCATION_INFORMATION	BLUETOOTH	NO_SENSITIVE_SINK	Total	
DATABASE_INFORMATION	6643	6174	127	98	108	43	0	61	0	0	0	3	0	18046	**31303**	9,2%
CALENDAR_INFORMATION	3529	7199	244	305	189	17	0	12	15	0	74	3	0	25208	**36795**	10,9%
NETWORK_INFORMATION	4774	1529	122	70	53	37	85	24	0	0	0	0	0	27809	**34503**	10,2%
LOCATION_INFORMATION	3758	1242	37	22	12	0	0	15	2	0	0	28	0	14187	**19303**	5,7%
CONTENT_RESOLVER	1769	823	74	24	22	2	5	12	4	0	2	1	0	3572	**6310**	1,9%
UNIQUE_IDENTIFIER	796	514	17	1	2	3	1	5	0	0	0	0	0	1819	**3158**	0,9%
ACCOUNT_INFORMATION	477	264	18	2	6	266	0	2	0	80	0	0	0	1032	**2147**	0,6%
CONTACT_INFORMATION	390	92	0	0	0	5	0	0	0	0	0	0	0	21	**508**	0,2%
FILE_INFORMATION	250	136	4	14	5	0	0	2	0	0	0	0	0	921	**1332**	0,4%
NFC	53	129	7	6	0	0	0	0	73	0	0	0	0	165	**433**	0,1%
BLUETOOTH_INFORMATION	153	23	0	0	1	0	0	2	0	0	0	0	0	609	**788**	0,2%
SMS_MMS	42	0	0	0	0	0	81	0	0	0	0	0	0	156	**279**	0,1%
SYNCHRONIZATION_DATA	19	0	0	0	0	0	0	0	0	10	0	0	0	137	**166**	0,0%
IMAGE	15	0	0	0	0	0	0	0	0	0	0	0	0	58	**73**	0,0%
BROWSER_INFORMATION	5	3	0	0	0	0	0	0	0	0	0	0	0	1	**9**	0,0%
SYSTEM_SETTINGS	5	0	0	0	0	0	0	0	0	0	0	0	0	26	**31**	0,0%
NO_SENSITIVE_SOURCE	147936	45028	4092	2626	355	39	61	1235	1	49	22	26	2	0	**201472**	59,5%
Total	**170614**	**63156**	**4742**	**3168**	**753**	**412**	**233**	**1370**	**95**	**139**	**98**	**61**	**2**	**93767**	**338610**	
	50,4%	18,7%	1,4%	0,9%	0,2%	0,1%	0,1%	0,4%	0,0%	0,0%	0,0%	0,0%	0,0%	27,7%		

Source	LOG	INTENT	NETWORK	FILE	SYSTEM_SETTINGS	ACCOUNT_SETTINGS	SMS_MMS	AUDIO	NFC	SYNCHRONIZATION_DATA	CALENDAR_INFORMATION	LOCATION_INFORMATION	BLUETOOTH	NO_SENSITIVE_SINK	Total	
DATABASE_INFORMATION	21006	25363	402	332	32	4	108	212	0	0	0	0	0	71906	**119365**	8,0%
CALENDAR_INFORMATION	5649	3599	94	74	61	0	4	66	0	0	0	0	0	62683	**72230**	4,8%
NETWORK_INFORMATION	29930	3293	5174	373	73	3	11802	116	0	0	0	0	0	261739	**312503**	20,9%
LOCATION_INFORMATION	22548	1831	1242	6	0	0	68	14	0	0	0	6	0	80152	**105867**	7,1%
CONTENT_RESOLVER	9897	3312	859	79	29	0	36	89	0	0	0	0	0	10176	**24477**	1,6%
UNIQUE_IDENTIFIER	10981	1179	6608	240	1	0	579	17	0	0	0	0	0	49279	**68884**	4,6%
ACCOUNT_INFORMATION	86	114	5	0	2	87	0	2	0	6	0	0	0	1246	**1548**	0,1%
CONTACT_INFORMATION	412	276	0	1	11	0	581	13	0	0	0	0	0	29	**1323**	0,1%
FILE_INFORMATION	706	255	0	0	12	0	0	32	0	0	0	0	0	1791	**2796**	0,2%
NFC	36	7	2	0	0	0	4	0	11	0	0	0	0	95	**155**	0,0%
BLUETOOTH_INFORMATION	79	222	10	6	0	0	0	21	0	0	0	0	1	1968	**2307**	0,2%
SMS_MMS	33	18	0	0	0	0	117	0	0	0	0	0	0	185	**353**	0,0%
SYNCHRONIZATION_DATA	3	1	0	0	0	0	0	0	0	0	0	0	0	30	**34**	0,0%
IMAGE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	0,0%
BROWSER_INFORMATION	15	12	0	0	0	0	0	0	0	0	0	0	0	173	**200**	0,0%
SYSTEM_SETTINGS	0	3	0	0	0	0	0	0	0	0	0	0	0	38	**41**	0,0%
NO_SENSITIVE_SOURCE	505584	222320	25897	23293	125	160	331	4362	0	137	0	313	1	0	**782523**	52,4%
Total	**606965**	**261805**	**40293**	**24404**	**346**	**254**	**13630**	**4944**	**11**	**143**	**0**	**319**	**2**	**541490**	**1494606**	
	40,6%	17,5%	2,7%	1,6%	0,0%	0,0%	0,9%	0,3%	0,0%	0,0%	0,0%	0,0%	0,0%	36,2%		

3.4.3 Data Flow in Benign Apps

Table 3.6 summarizes the data flows detected in our set of "benign" apps. Most interesting, 68.3% of all accesses to sensitive data do not end in a sensitive sink. Across the sensitive sinks, we detected 43,371 different data flows, i.e., 43,371 distinct pairs of code locations accessing a sensitive source API and a sensitive sink API linked by data flow between them. The most important source is DATABASE_INFORMATION followed by CALENDAR_INFORMATION, NETWORK_INFORMATION and LOCATION_INFORMATION. This reflects what most Android apps do: interacting with external services using information maintained in their own databases.

As it comes to the least frequently used sources, we find patterns that reflect programming practices in Android. The source EMAIL shows no flows at all, which might be surprising, considering the number of the apps that handle phone calls or e-mails. This is because most of e-mail access takes place via IMAP and POP protocols and thus belongs to the NETWORK_INFORMATION source category. The sources SYSTEM_SETTINGS and BROWSER_INFORMATION rarely end in sensitive sinks.

The most important sinks are LOG and INTENT, which make up more than 94% of all sinks in sensitive flows. As discussed in Section 3.4.2, the INTENT category means that the data was used by another activity in the app, a flow we currently cannot analyze. LOG, however, is a true sink but it is less harmful as starting from Android 4.1 log files can only be accessed by diagnostic and administrative tools.

The data set coming with this thesis contains detailed information on all flows, showing the exact flows between APIs for all benign applications.

In "benign" apps, 94% of all sensitive data flows are to logging and Inter-Process-Communications, i.e. intent.

3.4.4 Data Flow in Malicious Apps

Table 3.6 summarizes the data flows detected in our set of "malicious" apps, showing similarities but also important differences to the "benign" apps from Table 3.6. The most important source here is NETWORK_INFORMATION, which is almost twice as prevalent as in "benign" apps. To our suprise, CALENDAR_INFORMATION is accessed as a sensitive source far less frequently that ACCOUNT_INFORMATION.

In sinks, we also see important differences. Most striking is the SMS_MMS sink, that is over 77 times more prevalent than in our benign apps. This reflects the common stealthily attack sending SMS messages to premium numbers and allowing the owner of these numbers to earn money from the victim. As the flows to SMS_MMS indicate, the malicious apps also include sensitive data such as UNIQUE_IDENTIFIER and CONTACT_INFORMATION in their messages, as well as NETWORK_INFORMATION such as network MAC addresses or SIM card information.

Given that 25% of our malicious apps use SMS as a sink, whereas this is the case for only 1% of the "benign" apps, a simple check for the ability to send SMS messages would easily weed out 25% of malicious apps, with a precision of 99%. Note, however, that several of such simple checks may bring conflicting classifications. Also, while our "benign" set is representative in that it encompasses the most popular apps, our "malicious" set is in no way representative for malware actually prevalent in the wild or the types of attacks actually conducted. In that sense, Table 3.6 serves as descriptive statistics of the dataset we use for the evaluation of SAFAND.

Our set of "malicious" apps differs from the "benign" apps in terms of sources, sinks, and flows.

3.4.5 Sensitive Data Flows

MUDFLOW introduces a *flow of sensitive* data as a notion of the *usage of sensitive data*. A pattern of sensitive data usage has a following form:

$$Data\ Flow\ Pattern = \{Sensitive\ Source \rightsquigarrow Sensitive\ Sink\}$$

As discussed in Section 2.3.2 *sources* are Android APIs that are responsible for obtaining data from a device. In turn *sinks* are Android APIs that are responsible for leaking data from the device. Depending on the type of analysis, patterns of sensitive data usage can have different representations. MUDFLOW extracts all dataflows within a particular Android app by using SAFAND. However, due to inability of static analysis to know values of variables, these dataflows can only be represented as pairs of *signatures* of methods, which are in fact pairs of Android API methods:

$$\{TelephonyManager.getSubscriberId() \rightsquigarrow URL.openConnection()\}$$

Table 3.7: Flows in Android Twitter App

AccountManager.get()	⇝ ContentResolver.setSyncAutomatically()
AccountManager.get()	⇝ AccountManager.addOnAccountsUL()
AccountManager.get()	⇝ Activity.setResult()
AccountManager.get()	⇝ Log.w()
AccountManager.getAccountsByType()	⇝ ContentResolver.setSyncAutomatically()
AccountManager.getAccountsByType()	⇝ Activity.setResult()
AccountManager.getAccountsByType()	⇝ Log.w()
Uri.getQueryParameter()	⇝ Activity.startActivity()
Uri.getQueryParameter()	⇝ Activity.setResult()
Uri.getQueryParameter()	⇝ Activity.startActivityForResult()
Uri.getQueryParameter()	⇝ Log.d()
Uri.getQueryParameter()	⇝ Log.v()
Uri.getQueryParameter()	⇝ Log.w()
SQLiteDatabase.query()	⇝ Log.d()
SQLiteOpenHelper.getReadableDatabase()	⇝ Log.d()
SQLiteOpenHelper.getWritableDatabase()	⇝ Log.d()

Table 3.8: Flows in com.keji.danti604 malware

TelephonyManager.getSubscriberId()	⇝ URL.openConnection()
TelephonyManager.getDeviceId()	⇝ URL.openConnection()

As an example of such flows, consider the well known Android *Twitter* app. Table 3.7 shows its extracted data flows. We can see that, while the Twitter app accesses sensitive account information, it uses this information only to manage synchronization across multiple devices. Network information was accessed (as a part of the main functionality of the app), saved in logs, and passed on to other components.

In contrast, consider the *com.keji.danti604* malware from the VirusShare database [55]. Table 3.8 shows two flows in that application; they leak the subscriber and device ID to a Web server. Both these flows are very uncommon for benign applications. Furthermore, *danti604* does not contain any of the flows that would normally come with apps that use the *TelephonyManager* for legitimate reasons. Thus, *danti604* is an anomaly—not only because it may be similar to known malware, but in particular because its data flows are *dissimilar* to flows found in benignware such as Twitter.

Section 3.4.3 and Section 3.4.4 showed that benign and malicious apps use sensitive data differently. Therefore thus, MUDFLOW implemented multiple classifiers trained on the data flow of benign apps to automatically flag apps with suspicious features. We kindly point to the original MUDFLOW paper [12] for obtaining more information concerning the used machine learning techniques. In Section 3.4.6 we briefly discuss the idea and obtained results.

3.4.6 Detecting Malicious and Abnormal Apps

As shown above, the flow within malicious apps may differ significantly from the flow within benign apps. MUDFLOW leverages such differences to *automatically classify* novel apps whether they are malicious or not. While most malware detection is *retrospective* in nature—checking apps against patterns found in known malware— MUDFLOW is able to compare a new app *against benignware only* and check whether it contains abnormal flows with respect to this set. This allows MUDFLOW to detect malware as abnormal even if the specific attack is the first of its kind.

The MUDFLOW malware classification takes a set of benign apps, e.g., say, all apps from an app store, and then performs in the following three steps:

Per-category outlier detection For each SUSI category such as UNIQUE_IDENTIFIER (shortened to "ID"), MUDFLOW selects apps that use APIs of that category as source and uses their flows as features. Then it takes a new unknown app and determines its outlier score with respect to the "normal" apps. The higher the score is, the less "normal" the app behaves inside a particular SUSI category. See Figure 3.5.

Aggregating probabilities across API usage Given an app, for each SUSI category, we use the approach from Figure 3.5 to determine the distance of the app with respect to the benign training set. The resulting vector of scores ("maliciogram") tells how abnormal the app is in each category. See Figure 3.6.

Classifying apps across multiple categories For each "benign" app in the Google Play store, we determine its vector of probabilities of being an outlier in each SUSI category (Figure 3.6). A one-class classifier trained from these vectors can label an unknown app as "likely benign" if it is normal across all categories, or "likely malicious" instead. See Figure 3.7.

To obtain more details on how the classification works, we kindly refer the interested reader to the MUDFLOW paper [12].

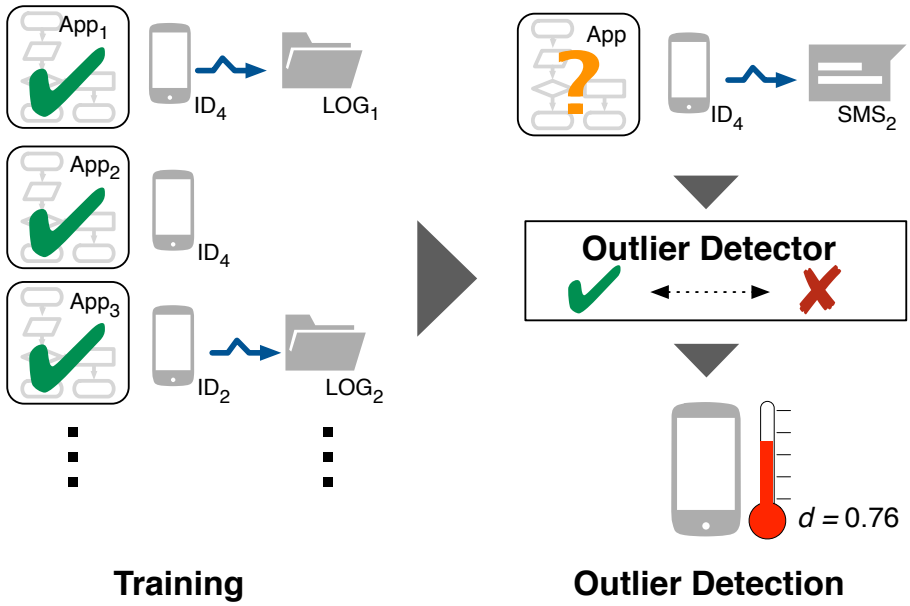

Figure 3.5: Per-category outlier detection.

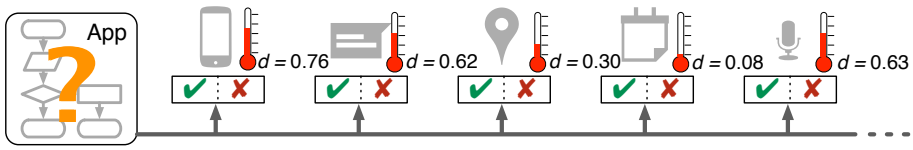

Figure 3.6: Aggregating probabilities across API usage.

Results

In our first experiment, we evaluated the full MUDFLOW classifier on the dataset from Section 3.4.1 as described in Section 3.4.6.

The average results are as follows:

True positives (malware recognized as such): 86.4%

True negatives (benignware recognized as such): 81.3%

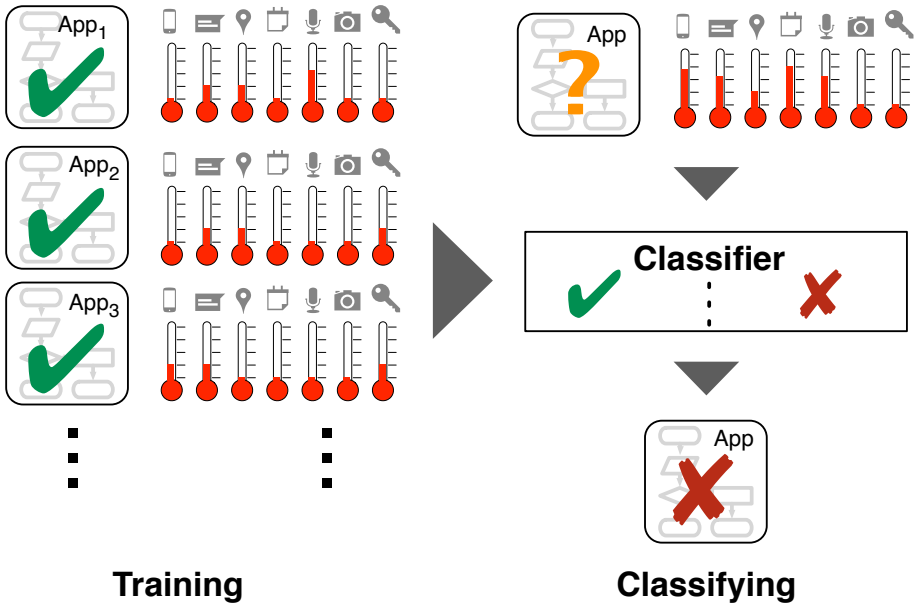

Figure 3.7: Classifying apps across multiple categories.

Accuracy (apps correctly classified): 83.8%

MUDFLOW recognizes 86.4% of malware as such, with the false positive rate of 18.7%.

As we are most interested in apps that access and send sensitive data, we ran the second evaluation on the subset of 10,552 "malicious" apps that have at least one flow from a sensitive source to a sensitive sink (i.e., malware leaking sensitive data). For this "sensitive" subset, we get the following results:

True positives (malware recognized as such): 90.1%

True negatives (benignware recognized as such): 81.3%

Accuracy (apps correctly classified): 86.0%

Table 3.9: Effectiveness of MUDFLOW using different features.

Features	Malicious set	True positives	True negatives	Accuracy
Source methods	all	81.7%	82.5%	82.1%
Sink methods	all	71.0%	83.9%	77.2%
Flow between classes	all	82.7%	79.7%	81.2%
	sensitive	87.7%	79.9%	83.7%
Flow between methods	all	86.4%	81.3%	83.8%
	sensitive	90.1%	81.3%	86.0%

MUDFLOW recognizes 90.1% of malware leaking sensitive data as such, with the false positive rate of 18.7%.

Again, all these numbers come from one-class classification; that is, no existing malware is used for training.

In **RQ1** and **RQ2** we wanted to understand if APIs and flows of sensitive data on itself can be used to identify applications with suspicious and malicious behavior. To answer to these questions we repeated the evaluation using different features. Notably, we checked the classification results using *source methods alone* as features, as this would not require complex static analysis. We can therefore confirm that both sensitive APIs and sensitive Dataflows can be used effectively to identify applications with suspicious behavior. As summarized in Table 3.9, *data flow between methods* shows the best accuracy perfromance across all features.

Chapter 4

Mining UI elements

4.1 Related Work

ASDROID [29] by Huang et al. is the first work that explicitly analyzes Android applications for mismatches between user interfaces and program behavior. Its general setting is similar to BACKSTAGE, in the sense that it maps UI elements to invoked functions and checks labels as well as invoked APIs. However, it only checks for a small set of fixed scenarios, such as sending text messages or making phone calls in the background, both in terms of analyzed labels as well as in the set of invoked APIs. This is because ASDROID focuses on stealthy (malicious) behavior only.

BACKSTAGE can be seen as the generalization of ASDROID. By mining thousands of UI elements, BACKSTAGE can detect *arbitrary mismatches* between user interfaces and associated behavior. Such mismatches include stealthy behavior as detected by ASDROID. In mature apps with well-tested user interfaces, most anomalies found by BACKSTAGE would show stealthy behavior, as this would not be found during GUI testing. However, such mismatches also include misleading button labels, wrong API associations and more. All these would be discovered by BACKSTAGE only.

GATOR [60] by Yang et al. was the first work to provide precise mappings between ANDROID UI elements and their callbacks via pure static analysis. Earlier works had focused on dynamic analysis and exploration, either in a black box [3] or a grey box [61] style. The advantage of static analysis is that it can explore and identify UI elements that would be hard to reach dynamically—because ac-

cessing them would require, for example, a password, an in-app purchase or the defeat of a boss monster.

The analysis in BACKSTAGE follows the GATOR approach in creating such mappings. However, we also address the specific need to extract the visible *text* and *context* from the UI elements, as well as to identify *dynamic changes* of text, context and callbacks. They do not consider menus, fragments and dialogs on the layout level. Furthermore, they do not extract listeners from XML layout files and APIs of UI elements. Our analysis can thus be interpreted as the specialization of GATOR towards text extraction.

UIPicker [43], SUPOR [27] and BIDTEXT [28] also analyze UI elements of Android apps. However they do so to automatically identify sensitive user inputs and sensitive data disclosure. Thus, their final aim is quite different from ours.

One field that is missing in the above list is *Human-Computer Interaction* (HCI). Interestingly, to the best of our knowledge and to the knowledge of HCI experts in the field, *we are not aware of any work in HCI that would rely on large-scale mining and analysis of UI elements.* Thus BACKSTAGE opens the door for general *automatic anomaly detection* in user interfaces considering features such as their visual appearance, their natural language semantics, their layout, their interaction or their behaviors. We see plenty of future potential in this direction.

4.2 Overall Design

The next module of SAFAND is the module to analyze and extract UI elements from applications. As easy as it may initially seem, implementing a sound technique to analyze the Android UI is not trivial, given the complexity of the Android GUI [57].

Each UI mining module implements the inter-procedural analysis. For example, if SAFAND searches for the `View:setText(resourceId)` API it does not assume a very simple case when the value of the `resourceId` variable is directly passed to the `View:setText` method. Our initial experiments showed that developers tend to use more complex scenarios when the `resourceId` is stored as a variable inside the method and even as the argument of the method. We will discuss the inter-procedural analysis of each UI mining phase in detail in the corresponding section.

The UI mining phase starts with exploring lifecycle methods of activities (Table 4.1) and fragments (Table 4.2). These methods act as top-level methods in Android. For each of these methods, it will be searched for the set of reachable

Table 4.1: Signatures of Activity Lifecycle Methods

```
void onCreate(android.os.Bundle)
void onStart()
java.lang.CharSequence onCreateDescription()
void onPostCreate(android.os.Bundle)
boolean onCreateOptionsMenu(android.view.Menu)
void onCreateContextMenu(android.view.ContextMenu,
        android.view.View,android.view.ContextMenu$ContextMenuInfo)
```

methods, which is then analyzed in the context of the corresponding top-level parent. If the analysis identifies the button in reachable methods, it will be assigned to the corresponding activity of the top-level parent. For each of the reachable methods, SAFAND searches for layouts, tab views and fragments and menus in order to construct the UI hierarchy as well as dialogs, dynamically defined strings and listeners in order to enrich UI elements with later required information.

Algorithm 5 UI mining strategy

Require: Callgraph CG
1: **procedure** UIMining(CG)
2: $LC \leftarrow$getActivityAndFragmentLifecycleMethods
3: **for** L in LC **do**
4: analyzeReachableMethods(L,CG)
5: **end for**
6: **end procedure**
7: **procedure** analyzeReachableMethods(L,CG)
8: $RM \leftarrow$getReachableMethods(L,CG)
9: **for** M in RM **do**
10: searchForDialogs(M)
11: searchForLayouts(M)
12: searchForDynamicStrings(M)
13: searchForListeners(M)
14: searchForFragments(M)
15: searchForTabViews(M)
16: searchForMenus(M)
17: **end for**
18: **end procedure**

Table 4.2: Signatures of Fragment Lifecycle Methods

```
android.view.View onCreateView(android.view.LayoutInflater,
        android.view.ViewGroup,android.os.Bundle)
void onViewCreated(android.view.View,android.os.Bundle)
void onAttach(android.os.Bundle)
void onCreate(android.os.Bundle)
void onStart()
java.lang.CharSequence onCreateDescription()
void onPostCreate(android.os.Bundle)
boolean onCreateOptionsMenu(android.view.Menu)
void onCreateContextMenu(android.view.ContextMenu,
        android.view.View,android.view.ContextMenu$ContextMenuInfo)
```

In Section 2.2 we provided background information on the Android GUI, i.e. how to declare elements and how to arrange them in the layout. We now proceed with an explanation how we extract the information that anomaly detection mechanisms need.

Some UI mining techniques as analysis of reusable layouts, menus, tab views and fragments are contribution of another thesis. We kindly refer the interested reader to them for more details[48].

4.3 Resource Analysis

Google in its official documentation suggests to externalize application resources such as images and strings to simplify their maintenance without changing the code[7].

Kuznetsov et al. showed that statically defined resources are also useful for anomaly detection[34]. In this section we will demonstrate how SAFAND extracts them from applications.

4.3.1 Extraction of Statically-Defined Text

All declarations of text values and their references are done within XML files. Therefore processing of these XML files is nothing more than parsing of them. All strings are placed in `res\values` folder. Those strings are default and developers can redefine them for each language. If they do not do it, the default values are used.

SAFAND uses the same concept. It searches for strings in the default folder which is `res\values` and then overwrites the values from the specified language. The default language is a native language of the application and is not neces-

Table 4.3: A set of attributes responsible for binding text to UI elements

android:text	android:title	android:textOn
android:hint	android:contentDescription	android:textOff
android:label		

sary English. Currently SAFAND supports only extraction of English resources by searching for overrides of default strings in `res\values-en` folder. By performing the small extension of the SAFAND framework one can make the target folder as a program parameter and extract the text for the desired language.

Text declaration in layout files. Technically, the definition of the text is done by setting the `android:text` attribute in the layout file. As an example, consider line 7 in Listing 2.1. The text can be defined either by using the reference to the app's resources with the `"@string/"` prefix or directly by providing the string that will be displayed. Even if the second option is deprecated, since it introduces localization problems, SAFAND supports it. There are more attributes that can be used to set a label for a UI element (refer to Table 4.3), and SAFAND supports all of them. If the definition is done by referencing the value of resources, SAFAND will search it in the default resources location and then override values from the English resources as discussed before.

Text declaration in style files. Less obvious but also a quite convenient way is to define the text in style files. These files should have `.xml` ending and be placed inside the `res\values` folder. This option is typically used when the text of UI labels changes depending on the style. Developers can specify text of UI elements by creating an `item` with the attribute `name="android:text"`. As an example, consider Listing 2.2, line 5. SAFAND identifies the reference to the styles and then searches for the definition of the referenced style in XML files under the default location. Then it overrides the reference if it is present in the location of English resources.

4.3.2 Analysis of Icons

Icons serve the same psychological purpose as paragraph breaks: they visually break up the content, making it less intimidating.[30]. Icons are placed under `res\drawable` folder. But developers can redefine icons for each screen resolution and then place them, for example, under `res\drawable-hdpi` folder. Icons from this folder are used for devices with resolution $72px \times 72px$, which is the high-resolution screen.

However, SAFAND does not process icons by itself right now, it reads only their annotation. Therefore the framework takes referenced icons from the `res\drawable` folder and provides them alongside with their annotation as an output.

Annotations of icons are, in fact, their alternative text, which can be read out loud to the user by a speech-based accessibility service. The reference to the icons is specified via `android:drawable` attribute (refer to line 11 in Listing 2.1). The annotation of the icon, in turn, is specified in the `android:contentDescription` attribute of the UI element (refer to line 12 in Listing 2.1). SAFAND then performs the search for the actual text by using the technique described in Section 4.3.1. This information is what SAFAND extracts and uses for UI elements which have icons instead of text labels.

SAFAND relies on developers during the analysis of icons. It assumes that they always the initialize alternate text of the icon with the respective value. Otherwise, it can not identify the description of the icon. One possible task for the future work is to analyze content of icons with image-processing techniques and annotate them based on extracted labels.

4.4 Analyzing Android Activities and their Layout

In Section 2.2.1 it has been discussed how to define a layout of an application. However, people rarely use such simple cases when the value of `layoutFileId` is directly provided as an integer constant. Modern software has quite complex architecture and consists of dozens of modules and classes. Moreover, Object-Oriented-Principles (OOP) guide developers to write high quality code that can be sometimes quite complicated for static analysis.

Consider the following scenario which we found in one of the analyzed applications to evaluate how complex the design could be. Normally, when the developer implements a new activity, she has to set `android.app.Activity` as

its superclass and then assign its `XML layout` by using `setContentView` method. However, if the app has a lot of screens, developers will create their own base class that extends the `android.app.Activity` and then inherit it by all activities in the app. Such base classes often contain a lot of common functionality and the common. In particular, each and every activity has to call `setContentView` method to bind an XML layout. Guidelines would help developers to extract this functionality to the base class and put it to its `onCreate` method (refer to Listing 4.1 as an example). But then the developer should somehow pass the `layoutFileId` which is unique for each child activity. As `onCreate` does not accept any additional parameters, guidelines would say that the developer needs to define the *abstract* `getLayoudFileId` method and force each child activity to implement it, return the appropriate value of the layout file (refer to Listing 4.2 as a final code of the child class).

```
1  public abstract class AbstractActivity extends Activity{
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5      }
6      protected void setLayout(){
7          this.setContentView(getLayoudFileId());
8      }
9      protected abstract int getLayoudFileId();
10 }
```

Listing 4.1: An abstract Activity class

```
1  public class ChildClass extends AbstractActivity {
2      @Override
3      protected void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5          this.setLayout();
6      }
7      @Override
8      protected int getLayoudFileId() {
9          return R.layout.activity_child_class;
10     }
11 ...
12 }
```

Listing 4.2: Implementation of the abstract Activity

On the one hand, this code, follows OOP guidelines. On the other hand, it is quite complicated for static analysis. If we search for occurrences of `android.app.Activity:setContentView(layoutFileId)` in the code, we will reach line 7 of Listing 4.1. Then the intra-procedural analysis will try to resolve the return value of the `getLayoutFileId()` method. As his is the *abstract* method and every child activity redefines it, analysis will have the mapping of the `AbstractActivity` to all layout files of its children. Such mapping does not give any helpful information as the `AbstractActivity` does not represent any screen and moreover has multiple layout files.

To catch such cases we have designed the UI Mining phase of SAFAND in a way that we analyze each activity in complete isolation and track the return context. In particular, our analysis will start from the `onCreate` method of the `ChildClass` as it has a GUI representation on the screen. Then it will procceed to the `AbstractActivity:onCreate` because there is a call edge. From there it will reach `AbstractActivity:getLayoutFileId` and then choose the correct implementation of the method by memorizing that it started from the `ChildActivity` class. By using the described heuristic, the analysis will end up at line 9 of the Listing 4.2.

Besides using XML files, developers can create the entire activity layout dynamically in the app code. This strategy is rarely used, since it is error-prone and makes the maintenance of the GUI harder. Given their little prevalence, entire dynamically generated screen layouts are out of the scope of SAFAND for now.

4.5 Analyzing Alert Dialogs

As we discussed in Section 2.2.2, the callback of each button in the alert dialog performs completely a different action. Therefore we store the exact mapping of a call back to a button in the dialog (refer to Listing 4.18 for an example). Moreover we bind the label to the corresponding button in order to have the mapping between label, action and set of APIs later on (see Section 4.10).

The procedure of finding alert dialogs works as described in Algorithm 6. The algorithm searches for the method call which is responsible for displaying dialogs on the app's screen. From these code points the algorithm searches for dialog messages, titles, buttons and their corresponding callbacks, using backward analysis.

Algorithm 6 Mining of Alert Dialogs

Require: Callgraph CG

```
1: procedure searchForDialogs(M)
2:     UNITS ← getUnitsOfMethod(M)
3:     for U in UNITS do
4:         if U contains InvokeExpr then
5:             CM ← getInvokedMethod(U)
6:             if CM == SHOW_METHOD then
7:                 REG ← getInstanceOfDialog(U)
8:                 AD ←propagateAndCollectData(U, M, REG)
9:             end if
10:        end if
11:    end for
12: end procedure
13: procedure resolveButtonAndAdd(D,AD, BTNTYPE)
14:    STRREG ← getLabelReg(D)
15:    STR ← resolveString(STRREG)
16:    CBREG ← getCallbackReg(D)
17:    CBSIG ← resolveSignatureOfCB(CBREG,D)
18:    addButton(AD,CBSIG,SRT,BTNTYPE)
19: end procedure
```

```
20: procedure PROPAGATEANDCOLLECTDATA(U, M, REG)
21:     AD = NULL
22:     D ← GETIMMEDIATEDOMINATOR(U,M)
23:     while D is not NULL do
24:         if D contains InvokeExpr AND ISTHESAMEINSTANCE(REG) then
25:             CM ← GETINVOKEDMETHOD(D)
26:             if CM == CREATE_METHOD then
27:                 AD ← new AlertDialog; continue
28:             end if
29:             if CM == INIT_METHOD then return AD
30:             end if
31:             if AD is not NULL then
32:                 if CM == SET_TITLE then
33:                     STRREG ← GETTITLEREG(D)
34:                     STR ← RESOLVESTRING(STRREG)
35:                     ADDTITLE(AD,SRT)
36:                 else if CM == SET_MESSAGE then
37:                     STRREG ← GETMESSAGEREG(D)
38:                     STR ← RESOLVESTRING(STRREG)
39:                     ADDMESSAGE(AD,SRT)
40:                 else if CM == SET_NEGATIVE_BUTTON then
41:                     RESOLVEBUTTONANDADD(D,AD,NEGATIVE)
42:                 else if CM == SET_POSITIVE_BUTTON then
43:                     RESOLVEBUTTONANDADD(D,AD,POSITIVE)
44:                 else if CM == SET_NEUTRAL_BUTTON then
45:                     RESOLVEBUTTONANDADD(D,AD,NEUTRAL)
46:                 end if
47:             end if
48:         end if
49:         D ← GETIMMEDIATEDOMINATOR(D,M)
50:     end while
51:     return AD
52: end procedure
```

Table 4.4: Signatures of StringBuilder methods

java.lang.StringBuilder:	java.lang.String toString()
java.lang.StringBuilder:	java.lang.StringBuilder append(java.lang.String)

4.6 String Analysis

Extraction of the text of UI elements is essential when it comes to the definition of the visible behavior on the screen. In Section 4.3 we discussed how SAFAND handles statically defined text. Many UI mining modules which identify the text of UI elements deal with the problem of not explicitly defined strings in the method parameters. There is a variable or even another method invocation instead. The most interesting and the most complex case is string concatenation. String concatenation is done in Java by using the `StringBuilder` class. Thus, we continue with an explanation of the technique for resolving string concatenations.

SAFAND deals with string concatenation by analyzing `StringBuilder` instances. Moreover, it performs backward analysis from the method parameter if the relevant string is not specified there directly, but rather stored in some variables in other parts of the code (refer to Algorithm 7).

`StringBuilder` is an essential class in Java for working with textual information. Even if developers argue that they never use `StringBuilder` and prefer the simple concatenation with `+` sign, they are wrong. Java Virtual Machine (JVM) translates all simple concatenations like `''John `` + ''Doe``` to the StringBuilder at compile time while optimizing the code. The mentioned example is translated to the code in Listing 4.3.

```
1  String line = ``John '' + ``Doe'';
2  /* translates to */
3  StringBuilder stringBuilder = new StringBuilder();
4  stringBuilder.append(``John '');
5  stringBuilder.append(``Doe'');
6  String line = stringBuilder.toString();
```

Listing 4.3: Convertion of string concatenation to the StringBuilder by JVM

Our string analysis performs the inter-procedural backward propagation of a parameter of `View:setText()`. If `StringBuilder:toString()` assigns its value to the tracked variable, it collects all values of the `StringBuilder:append()` of the same instance of `StringBuilder` and joins them in a way how they would be displayed. Each and every variable in this module is analyzed using the inter-procedural backward method (refer to Algorithm 8).

Algorithm 7 Strategy of resolving the value of strings

```
Require: Caller method CM, Argument ARG, Current Unit CU
 1: procedure resolveString(CM,ARG,CU)
 2:     D ← getImmediateDominator(CU,CM)
 3:     while D is not NULL do
 4:         if D is AssignStmt then
 5:             R ← getRightOp(D)
 6:             L ← getLeftOp(D)
 7:             if L == ARG then
 8:                 if D contains InvokeExpr then
 9:                     M ← getMethodOf(R)
10:                     if M == STRING_BUILDER_TOSTRING then
11:                         B ← getRegOfSBInstance(R)
12:                         return findValueInStringBuilder(B,D,CM)
13:                     else
14:                         return resolveReturnValueInInvocation(D)
15:                     end if
16:                 else
17:                     if R is StringConstant then
18:                         return R
19:                     end if
20:                     if R is CastExpr then
21:                         L ← getCastedOp(R)
22:                     end if
23:                     if R is Field then
24:                         return findValueOfField(R)
25:                     end if
26:                 end if
27:             end if
28:         else if D is InvokeExpr then
29:             return resolveReturnValueInInvocation(D)
30:         else if D is the input parameter then
31:             PNUM ← getParamNumber(D)
32:             return findValueInCallers(PNUM,CM)
33:         end if
34:         D ← getImmediateDominator(D,CM)
35:     end while
36: end procedure
```

Algorithm 8 Strategy of resolving the value of the String Builder

```
1: procedure findValueInStringBuilder(B,CU,CM)
2:     S ← new String
3:     D ← getImmediateDominator(CU,CM)
4:     while D is not NULL do
5:         if D is AssignStmt then
6:             R ← getRightOp(D)
7:             L ← getLeftOp(D)
8:             if L == ARG then
9:                 if D containsInvokeExpr then
10:                    M ← getMethodOf(R)
11:                    if M == STRING_BUILDER_APPEND then
12:                        V ← getValueOfParameter(0)
13:                        RES ← resolveString(CM,V,D)
14:                        join(S,RES,"#")
15:                    end if
16:                else
17:                    if R is NewExpr then
18:                        return S
19:                    end if
20:                end if
21:            end if
22:        end if
23:        D ← getImmediateDominator(D,CM)
24:    end while
25:    return S
26: end procedure
```

Limitations

Over-approximation The content of some strings sometimes depends on specific conditions. For example, if it depends on the content of the file or its length, the problem becomes unsolvable for static analysis. Our analysis can not resolve such conditions due to their complexity and assumes that strings will contain assigned values from all conditions (e.g. over-approximation).

Dynamic values When user makes a purchase, she enters his credit card number to the text field, and then this information is being sent to the remote server as a part of `JSON` object. Our analysis can not tell which exactly value is being sent due to its dynamic nature. SAFAND only reports hard-coded strings and strings from resource files.

4.7 Extracting Context of Text Labels

Well-designed applications usually have semantically meaningful text labels to improve the application usability. For example, the button label "Send SMS" makes it clear that the user would send an SMS message by clicking that button. However, most of the times text labels are semantically generic and can be correctly interpreted only given their context. For instance, this is the case for labels such as "OK" or "Yes", which are highly prevalent. The expected behavior for clicking an "OK" button is to confirm an operation that has been mentioned earlier or is described somewhere else in the GUI (refer to Figure 4.1 for an example). As a consequence, together with the text label for a UI element, BACKSTAGE collects all the surrounding text, which we interpret as relevant *context* to understand the semantic of the label itself. More precisely, for each UI element we collect all text that the activity containing the element displays.

As shown in Figure 4.1, "OK" button has the following context: "Twitter would like to use your current location to customize your experience". But "Don't allow" text should not be definitely a part of the context of the "OK" button. Thus, we assume that only *inactive* text could be a part of the context. In other words, *inactive* text is the text that does not relate to any UI element, which has at least one assigned callback (refer to Section 4.8) and can perform any action. Static labels are instances of *inactive* context, while buttons and text fields usually are not as they may invoke `onKeyPressed` or `onClick` callbacks. But if a text field, for example, does not have any assigned callbacks,

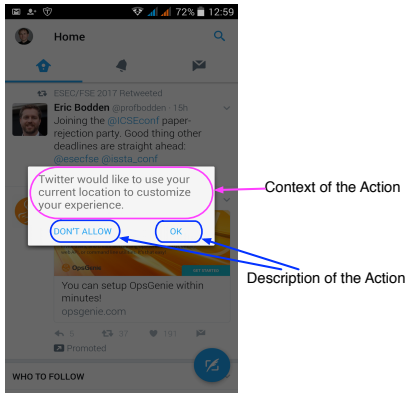

Figure 4.1: Actions and their Context

it wll participate in the context of UI elements from the same screen (refer to Algorithm 9).

Alert Dialogs, as shown in Figure 4.1 do not use the context of the screen where they appear. This type of UI elements is quite independent from outer environment. The title of the dialog, the message and three buttons, positive (usually "OK"), negative (usually "NO") and neutral (usually "Cancel") are completely independent pieces of the context and perform actions that are not related to the screen.

Limitations

Currently, SAFAND associates the inactive context of the whole screen with respective UI element. It assumes that one particular screen talks about one action. However, screens can talk about more actions in real life. One of the possible heuristics to reduce the over-approximation is to associate the UI element only with the elements that are on the same parent layout. But it does not work well for all applications as layouts can be designed in quite different ways. A more sophisticated technique for gathering the context is a possible future work.

Algorithm 9 Strategy of resolving the context of labels

Require: id of UI elements $ELIDS$, Screen hierarchy SH, Layouts $LIDS$
Ensure: Map of the elements to context $ELLC$

```
1: procedure RESOLVECONTEXTOFELEMENTS(ELIDS,LIDS,SH)
2:     ELLC ← map of the elements to context
3:     LC ← GATHERCONTEXTOFLAYOUTS(LIDS,SH)
4:     for ELID im ELIDS do
5:         ELC ← EXTRACTCONTEXTOFTHEUIELEMENT(ELID,LC,SH)
6:         PUTTOMAP(ELID,ELC,ELLC)
7:     end for
8: end procedure
9: procedure GATHERCONTEXTOFLAYOUTS(LIDS,SH)
10:    LC ← map of the layout to context
11:    for LID in LIDS do
12:        for CH in GETCHILDS(LID) do
13:            if CH has no callbacks then
14:                L ← GETLABEL(CH)
15:                ADDTOMAPPING(LID,L,LC)
16:            end if
17:        end for
18:    end for
19:    return LC
20: end procedure
21: procedure EXTRACTCONTEXTOFTHEUIELEMENT(ELID, SH)
22:    LID ← GETLAYOUTOFELEMENT(ELID, SH)
23:    CON ← GETVALUEFROMMAP(LID,LC)
24:    return CON
25: end procedure
```

Table 4.5: A set of standard callbacks in Android

afterTextChanged	onTextChanged	onKey
onEditorAction	onClick	onDrag
onHover	onLongClick	onTouch
onKeyboadDismiss	onItemClick	onItemLongClick
onItemSelected	onNothingSelected	onScroll
...	(42 more)	...

4.8 Mining Callbacks of UI Elements

SAFAND characterizes each UI element, which represented by the text label and its surrounding text, with the behavior that it would trigger at runtime. As a proxy to represent this behavior, it uses the set of the Android API invocations that are reachable and therefore can be executed. The algorithm needs first to identify the *callbacks*, since these are the entry points of the analysis. A callback is a special function that is bound to a particular event on a UI element which triggers its execution. The most well-known example of the callback is the *onClick* function, which gets executed when the user clicks on some UI element on a screen.

There are dozens of predefined UI callbacks available in Android, but developers can also implement their own custom callbacks and bind them to any UI element. However, SAFAND currently deals only with the predefined set of Android callbacks, which we report in Table 4.5.

Developers can declare a callback for a UI element either statically in a *layout file* or dynamically in the *app code*. Thus, we proceed with the discussion of each particular case and its challenges.

Defining callbacks in a layout file

The most straightforward way to define callbacks is to directly declare them in the layout XML file together with the corresponding UI element. In order to bind a callback to the UI element, developers should provide a name of the function to the `android:onClick` attribute of the corresponding UI element (refer to Listing 2.1, line 8 for an example). However, only *onClick* callbacks can be defined this way. All other types of callbacks that are listed in Table 4.5 have to be binded from the code and will be discussed in Section 4.8.

In Listing 2.1 the developer provided a callback with the name `xmlDefinedOnClick`. The next step is to define the function with the *same name* in the code of the same activity. The callback function in this particular case is listed in Listing 4.4.

```
1  public class myActivity extends Activity{
2        public void onCreate(android.os.Bundle savedInstanceState){
3        ...
4        }
5        ...
6        public void xmlDefinedOnClick(android.view.View button){
7        // an implementation of the functionality of the button click
8        }
9  }
```

Listing 4.4: An implementation of the XML defined callback

All XML defined callbacks of all UI elements should be registered in the code of the corresponding activities as they relate to the same class. Such design poses one important problem: a developer should take care of naming of callbacks and assign the unique name to each callback as multiple methods with the same name inside the same class are forbidden in Java.

Such design simplifies static analysis of UI elements. SAFAND parses the layout of the corresponding activity, collects `onClick` callback for each UI element and automatically resolves its class name. `xmlDefinedOnClick` method from Listing 2.1 has the following signature:

`myActivity:xmlDefinedOnClick(android.view.View)`.

The last step is to check the existence of the callback. The developer can define the name of the callback in the XML layout file but forget to write its implementation in the code. If such method is not declared in the Java code, it will not be mapped to the corresponding UI element.

Defining callbacks in code

The modern UI of mobile apps is flexible and responsive. Therefore the actual behavior of UI elements strongly depends on a huge number of preconditions. Thus there is a need to define the behavior at runtime: for example, assign `onClick` callback to the button if some radio-button is selected.

Java language provides multiple ways to bind a particular callback to the UI element from the code. We will illustrate each of them using an assignment of `onClick` callback to the button. `onClick` callback is assigned by using `Button:setOnClickListener()` method. All considered cases are from the real life and have been gathered from the real code.

Anonymous handler Anonymous expressions are quite popular nowadays. They give to the developer the possibility to avoid writing separate functions and classes while implementing some behavior. Refer to Listing 4.5 for an example.

```
1   public class myActivity{
2       public void onCreate(android.os.Bundle savedInstanceState){
3           Button myButton = (Button)findViewById(R.id.btnSave);
4           myButton.setOnClickListener(new View.OnClickListener() {
5               @Override
6               public void onClick(View view) {
7               //implementation of the onClick functionality
8               }
9           });
10      }
11  }
12
```

Listing 4.5: Binding an anonymous onClick listener to the button

Activity class that implements OnClickListener interface The next design option is to make a class that extends `android.app.Activity` and is actually responsible for the lifecycle of the screen as well as for the Button events. In other words, the class extends `android.app.Activity` and also implements `View.OnClickListener` interface. Refer to Listing 4.6 for an example. Now the developer just needs to pass `this` object as the listener and the corresponding `onClick` method of the same activity class at line 8 will be triggered.

```
1   public class myActivity extends Activity implements View.
    OnClickListener{
2       public void onCreate(android.os.Bundle savedInstanceState){
3           Button myButton = (Button)findViewById(R.id.btnSave);
4           myButton.setOnClickListener(this);
5       }
6
7       @Override
8       public void onClick(View view){
9       //implementation of the onClick functionality
10      }
11  }
12
```

Listing 4.6: Binding the onClick listener to the button which is also an Activity

OnClickListener is a private field of the Activity Some developers define the onClickListener as a field of the Activity class. Refer to the Listing 4.7 for an example. `myClick` here is a private field of the class and the developer assigns it to the event which will be fired when a user will click on that button.

```
1       public class myActivity extends Activity{
2           private View.OnClickListener myClick = new View.
    OnClickListener(){
3               public void onClick(View view) {
4               //implementation of the onClick functionality
5               }
6           };
7
8           public void onCreate(android.os.Bundle savedInstanceState){
9               Button myButton = (Button)findViewById(R.id.btnSave);
10              myButton.setOnClickListener(myClick);
11          }
12      }
13
```

Listing 4.7: Binding the onClick listener to the button which is also a private field of the same Activity

Sub-class that implements onClickListener interface The last considered case is when the sub-class of the Activity class acts as a button. Refer to Listing 4.8 for an example.

```
1       public class myActivity extends Activity{
2           public void onCreate(android.os.Bundle savedInstanceState){
3               Button myButton = (Button)findViewById(R.id.btnSave);
4               myButton.setOnClickListener(new MyInnerListener());
5           }
6           private class MyInnerListener implements View.
    OnClickListener{
7               @Override
8               public void onClick(View view){
9               //implementation of the onClick functionality
10              }
11          }
12      }
13
```

Listing 4.8: Binding the onClick listener to the button which is also a sub-class of the same Activity

Unlike callbacks that are defined statically (Section 4.8), runtime-defined callbacks cannot have different names. All of them have `onClick(android.view.View)` subsignature. And the challenge here is to correctly resolve the name of the class where this callback is defined. Moreover, developers like to extract some meaningful blocks of code to separate functions to make the code more readable and supportable (refer to Listing 4.9 for an example). Fortunately, SAFAND supports inter-procedural propagation of the button object as well as the listener object.

```
1  public class myActivity extends Activity{
2      public void onCreate(android.os.Bundle savedInstanceState){
3      Button myButton = (Button)findViewById(R.id.btnSave);
4          setListener(myButton, new MyOnClickListener());
5      }
6      private void setListener(android.widget.Button button, View.onClickListener
   listener){
7          myButton.setOnClickListener(listener);
8      }
9  }
10
11 public class MyOnClickListener implements View.OnClickListener{
12     @Override
13     public void onClick(View view){
14     //implementation of the onClick functionality
15     }
16 }
```

Listing 4.9: Binding the onClick listener to the button from the separate method

Developers are very creative in writing the code, that is why it is very important to track such cases. SAFAND tracks all of them and can correctly map UI elements to the corresponding callbacks. The precise algorithm is described in Algorithm 10.

UI elements on reusable layouts can also have different callbacks depending on the context. To track the context of the callback, we keep track of the class name that declares it. We will discuss the precise algorithm in Section 4.9.

Algorithm 10 Strategy of mining callbacks

Require: Callgraph CG, Current Method M, Set of callbacks CBS (Table 4.5)

```
1: procedure searchForListeners(M)
2:     UNITS ← findAssignmentsOf(CBS,M)
3:     for U in UNITS do
4:         UIReg ← getElementRegister(U)
5:         CBREG ← getCallbackRegister(U)
6:         ELID ← resolveElementId(UIReg,U)
7:         CBSIG ← resolveSignatureOfCallback(CBREG,U)
8:         addCallbackToElement(ELID,CBSIG)
9:     end for
10: end procedure
```

Limitations

Due to complexity of the code it is sometimes impossible to identify which exactly UI element a callback relates to. In such cases we just forget about such callbacks and better miss some of them rather than introduce the over-approximation.

4.9 Dealing with Redefining Text and Callbacks in Reusable Layouts

As we already mentioned in Section 2.2.1, modern mobile applications heavily use reusable layouts. Let us assume that the developer wants to create a custom piece of layout and put there two buttons and a title (Listing 4.10). The title as well as the button labels have predefined text. The behavior of this layout is still unknown and heavily depends on the context, e.g. on the screen where it will be injected. There are two screens to which the layout will be injected (Listing 4.11 and Listing 4.12). Each screen will be responsible for redefining the default text of the buttons and the title label as well as redefining the list of assigned callbacks (Listing 4.13 and Listing 4.14). Thus, there is a need to correctly map the text of each button to the associated callback, e.g., the label of the button on the screen #1 should be mapped to the callback that will be fired when a user will click on this button at this particular screen. Analysis must not mix labels and behaviors from different screens.

```
<FrameLayout>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="38dp"
        android:text="This is a default text of a fragment"
        android:id="@+id/frag_default_text" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="DEFAULT_OK"
        android:id="@+id/default_ok"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="DEFAULT CANCEL"
        android:id="@+id/default_cancel" />
</FrameLayout>
```

Listing 4.10: An XML layout file of the fragment.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout>
    <fragment
        android:id="@+id/fragment_with_buttons_inside"
        class="com.example.avdiienko.fragmentapp.FragmentWithButtons"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</RelativeLayout>
```

Listing 4.11: Injecting the fragment ot the activity layout of the main screen

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout>
    <fragment
        android:id="@+id/fragment_with_buttons_inside_2"
        class="com.example.avdiienko.fragmentapp.FragmentWithButtons"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</RelativeLayout>
```

Listing 4.12: Injecting the fragment ot the activity layout of the second screen

```
public class MainActivity extends FragmentActivity implements View.
OnClickListener, FragmentWithButtons.OnFragmentInteractionListener {
    private Button okButton;
    private Button cancelButton;
```

```
4
5           @Override
6           protected void onCreate(Bundle savedInstanceState) {
7                   super.onCreate(savedInstanceState);
8                   setContentView(R.layout.activity_main);
9                   TextView myFrag = (TextView) findViewById(R.id.frag_default_text);
10                  myFrag.setText("Main activity Text for TextView");
11                  okButton = (Button) findViewById(R.id.default_ok);
12                  okButton.setText("Main Acitvity OK");
13                  okButton.setOnClickListener(this);
14                  cancelButton = (Button) findViewById(R.id.default_cancel);
15                  cancelButton.setText("Main Acitvity Cancel");
16                  cancelButton.setOnClickListener(this);
17          }
18
19          @Override
20          public void onClick(View v) {
21          //implementation of the onClick functionality
22          }
23  }
```

Listing 4.13: Redefining text and callbacks of the fragment on the main screen

```
1
2   public class SecondActivity extends FragmentActivity implements View.
    OnClickListener, FragmentWithButtons.OnFragmentInteractionListener {
3
4           @Override
5           protected void onCreate(Bundle savedInstanceState) {
6                   super.onCreate(savedInstanceState);
7                   setContentView(R.layout.activity_second);
8                   TextView myFrag = (TextView)findViewById(R.id.frag_default_text);
9                   myFrag.setText("Second activity Text for TextView");
10                  Button okButton = (Button)findViewById(R.id.default_ok);
11                  okButton.setText("Second Acitvity OK");
12                  okButton.setOnClickListener(this);
13                  Button cancelButton = (Button)findViewById(R.id.default_cancel);
14                  cancelButton.setText("Second Acitvity Cancel");
15                  cancelButton.setOnClickListener(this);
16          }
17          @Override
18          public void onClick(View v) {
19                  //implementation of the onClick functionality
20          }
21  }
```

Listing 4.14: Redefining text and callbacks of the fragment on the second screen

The developer could inject the layout to the third activity but redefine only the text of the title and leave everything else as is. Or the developer could also define the default callback for the layout. Such cases are possible and apart from differentiating the text and callbacks on different screens, the analysis must store also their default values. SAFAND, due to analyzing each activity and fragment in isolation, is able to create such mapping. The mapping of the `OK` button is shown in Listing 4.15. `default_value` key says that the value has been assigned in the XML layout file, whereas other values point to the respective classes.

```
1  <AppsUIElement>
2        <id>2131492947</id>
3        <text>
4              <entry>
5              <string>default_value</string>
6              <string>DEFAULT_OK</string>
7              </entry>
8              <entry>
9              <string>com.example.avdiienko.fragmentapp.SecondActivity</string>
10             <string>Second Acitvity OK</string>
11             </entry>
12             <entry>
13             <string>com.example.avdiienko.fragmentapp.MainActivity</string>
14             <string>Main Acitvity OK</string>
15             </entry>
16       </text>
17       <kindOfUiElement>Button</kindOfUiElement>
18       <listeners>
19             <st.cs.uni.saarland.de.entities.Listener>
20             <declaringClass>com.example.avdiienko.fragmentapp.SecondActivity</
   declaringClass>
21             <listenerMethod>void onClick(android.view.View)</listenerMethod>
22             <listenerClass>com.example.avdiienko.fragmentapp.SecondActivity</
   listenerClass>
23             <xmlDefined>false</xmlDefined>
24             </st.cs.uni.saarland.de.entities.Listener>
25             <st.cs.uni.saarland.de.entities.Listener>
26             <declaringClass>com.example.avdiienko.fragmentapp.MainActivity</
   declaringClass>
27             <listenerMethod>void onClick(android.view.View)</listenerMethod>
28             <listenerClass>com.example.avdiienko.fragmentapp.MainActivity</
   listenerClass>
29             <xmlDefined>false</xmlDefined>
30             </st.cs.uni.saarland.de.entities.Listener>
31       </listeners>
32 </AppsUIElement>
```

Listing 4.15: Mapping of text and callbacks produced by SAFAND

Matching callbacks and text labels with default values

Indeed, Listing 4.15 shows the internal representation of results in a form that is inconvenient to the user. Moreover, it is not clear how we want to use `default_value` in our mapping. Therefore we perform the post-processing phase that resolves the final mapping between text items and callbacks. The algorithm is described in Algorithm 11.

Algorithm 11 Strategy of matching callbacks and text of UI elements.

Require: Text Mapping T, Callback Mapping C
Ensure: Mapping of Matchings M

```
1:  procedure RESOLVE(T, C)
2:      for key in GETUNIQUEKEYS(C, T) do
3:          if key in T and C then
4:              ADDTOMAPPING(T.get(key), C.get(key))
5:              REMOVEITEM(T, key)
6:              REMOVEITEM(C, key)
7:          else if key in C and CONTAINS(T, "default_value") then
8:              ADDTOMAPPING(T.get("default_value"), C.get(key), M)
9:          else if key in T and CONTAINS(C, "default_value"), M then
10:             ADDTOMAPPING(T.get(key), C.get("default_value"), M)
11:         end if
12:     end for
13:     return M
14: end procedure
```

In the first step the analysis iterates over all keys in the mapping of text and callbacks to the values. If it finds the matching of keys in both collections, it will add them to the final matching and remove them from the initial ones.

If the key is present only in one collection, the analysis identifies this collection and then checks whether there is an item with `default_value` key in the second one. If the operation is successful then the analysis will add the mapping of the item from the first collection and the default item from the second one.

Such algorithm makes it possible to cover cases when developers redefine only text labels or callbacks in the injected screen. The `default_value` key from the text collection of Listing 4.15 will not have any matching in the callbacks collection as later does not have the `default_value` key.

4.10 Context-Sensitive Analysis of Callbacks

As discussed in Section 4.8, callbacks can be assigned to UI elements either statically in a layout file or dynamically in the code. Beside the simple case when a single callback is bound to a single button, there are cases when one callback is bound to multiple buttons. As an example, consider the code in Listing 4.16 where the same `myClick` callback is assigned to both `okButton` and `cancelButton`. This example shows that to precisely assign API invocations to the right button the analysis should be context-sensitive, i.e. it should be able to correctly bind the `okButton` to the branch at line 16 and the `cancelButton` to the one at line 19 respectively.

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3      Button okButton = (Button) findViewById(R.id.ok_button);
4      okButton.setText(R.string.okButton);
5      Button cancelButton = (Button) findViewById(R.id.cancel_button);
6      cancelButton.setText(R.string.cancelButton);
7      okButton.setOnClickListener(myClick);
8      cancelButton.setOnClickListener(myClick);
9  }
10
11 View.OnClickListener myClick = new View.OnClickListener() {
12     public void onClick(View v) {
13         switch (v.getId()) {
14         case R.id.ok_button:
15         //action if button is the okButton
16         break;
17         case R.id.cancel_button:
18         //action if button is the cancelButton
19         break;
20         }
21     }
22 };
```

Listing 4.16: An example of assigning the same callback to multiple buttons

SAFAND checks each function that is reachable from a particular callback for such comparisons and takes the correct branch based on the `Id of UI element` to which the callback is mapped.

The same mechanism applies to the analysis of menu items. For now, SAFAND supports the context-sensitive analysis of Option and Contextual menus, tracks `MenuItem:getItemId()` comparisons and takes the correct branch based on the menu item to which the callbacks is mapped (refer to Listing 4.17 for an example). Other types of menus are out of scope of the current analysis.

```
1  @Override
2  public boolean onCreateOptionsMenu(Menu menu) {
3          getMenuInflater().inflate(R.menu.main, menu);
4          return true;
5  }
6  @Override
7  public boolean onOptionsItemSelected(MenuItem item) {
8          switch (item.getItemId()) {
9          case R.id.search_menu:
10                 //action if search menu item is clicked
11                 return true;
12         case R.id.share_menu:
13                 //action if share menu item is clicked
14                 return true;
15         case R.id.print_menu:
16                 //action if print menu item is clicked
17                 return true;
18         default:
19                 return super.onOptionsItemSelected(item);
20         }
21 }
```

Listing 4.17: An example of assigning the same callback to multiple menu items

The last type of UI elements that requires a special treatment case in the analysis is Alert Dialogs (refer to Figure 4.1 for an example). They have completely different behavior in contrast to buttons and menus. An alert dialog consists of three buttons: `positive`, `negative` and `neutral` (Listing 4.18). Each of them has a special unique id inside the alert dialog: `-1`, `-2` and `-3`, respectively. SAFAND analyses the `which` parameter of `DialogInterface.OnClickListener()` in order to differentiate behavior of these buttons. Such analysis is a must-have for alert dialogs as mixing up the behavior of `positive` and `negative` buttons will lead to the overapproximation and will introduce noise.

```
1  private void showDialog(){
2      DialogInterface.OnClickListener dialogClickListener = new DialogInterface.
   OnClickListener() {
3          @Override
4          public void onClick(DialogInterface dialog, int which) {
5              switch (which){
6                  case DialogInterface.BUTTON_POSITIVE:
7                  //perform the action
8                  break;
9
10                 case DialogInterface.BUTTON_NEGATIVE:
11                 //cancel the action
12                 break;
13
14                 case DialogInterface.BUTTON_NEUTRAL:
15                 //do nothing
16                 break;
17             }
18         }
19     };
20
21     AlertDialog.Builder builder = new AlertDialog.Builder(this);
22     builder.setMessage("Do you want to perform the action X?");
23     builder.setPositiveButton("Yes", dialogClickListener);
24     builder.setNegativeButton("No", dialogClickListener);
25     builder.setNeutralButton("Skip", dialogClickListener);
26     builder.create().show();
27 }
```

Listing 4.18: An example of assigning the same callback to multiple buttons in Alert Dialogs

Our SAFAND tool implements static context-sensitive inter-procedural analysis that correctly handles the most typical cases (see Algorithm 12). Unfortunately, as we discussed above, developers are so creative in writing the code and it is impossible to handle all cases when it comes to the analysis of more than 10000 apps.

Algorithm 12 Strategy of context-sensitive analysis of callbacks.

Require: UI element type $UITYPE$, UI element id UID, Callback CB, Callgraph CG
Ensure: Calls $CALLS$

```
1: procedure analyzeCallback(UITYPE, UID CB)
2:     if UITYPE == "Dialog_Click" then
3:         CALLS ← getCallsOfDialog(UID, CB)
4:         return CALLS
5:     end if
6:     PARAM_TYPES ← getParamTypes(CB)
7:     if "android.view.View" in PARAM_TYPES then
8:         PNUM ← getIndexOfParam("android.view.View")
9:         ID_SIG ← "android.view.View: int getId()"
10:        CALLS ← getCallsOfViews(UID, CB, PNUM, ID_SIG)
11:        return CALLS
12:    end if
13:    if "android.view.View" in PARAM_TYPES then
14:        PNUM ← getIndexOfParam("android.view.MenuItem")
15:        ID_SIG ← "android.view.MenuItem: int getItemId()"
16:        CALLS ← getCallsOfViews(UID, CB, PNUM, ID_SIG)
17:        return CALLS
18:    end if
19:    CALLS ← edgesOutOfMethod(CG, CB)
20:    return CALLS
21: end procedure
22: procedure getCallsOfDialog(UID, CB)
23:    CALLS ← new List()
24:    IDREG ← getParamReg(1)
25:    UNITS ← getUnits(CB)
26:    for U in UNITS do
27:        UNIT_CALLS ← analyzeFromUnit(U,IDREG,UID)
28:        addToList(CALLS, UNIT_CALLS)
29:    end for
30:    return CALLS
31: end procedure
```

```
32: procedure getCallsOfViews(UID, CB, PNUM ID_SIG)
33:     CALLS ← new List()
34:     IDREG ← NULL
35:     REG_INSTANCE ← getParamReg(PNUM)
36:     UNITS ← getUnits(CB)
37:     for U in UNITS do
38:         if U contains ID_SIG then
39:             LOC_INST_REG ← getInstanceVar(U)
40:             if REG_INSTANCE == LOC_INST_REG then
41:                 IDREG ← getLeftOp(U)
42:             end if
43:         end if
44:         if IDREG != NULL then
45:             UNIT_CALLS ← analyzeFromUnit(U,IDREG,UID)
46:             addToList(CALLS, UNIT_CALLS)
47:         end if
48:     end for
49:     return CALLS
50: end procedure
51: procedure analyzeFromUnit(BRANCH_UNIT,IDREG,UID)
52:     if U is SwitchStmt then
53:         SWITCH_REG ← getSwitchValue(U)
54:         if SWITCH_REG == IDREG then
55:             CASE_UNIT ← findCaseBasedOnId(UID)
56:             CASE_CALLS ← analyzeFromUnit(CASE_UNIT,IDREG,UID)
57:             addToList(CALLS, CASE_CALLS)
58:             return CALLS
59:         end if
60:     end if
61:     if U is IfStmt then
62:         IF_REGS ← getIfRegs(U)
63:         if IDREG in IF_REGS then
64:             BRANCH_UNIT ← findBranchBasedOnId(UID)
65:             BR_CALLS ← analyzeFromUnit(BRANCH_UNIT,IDREG,UID)
66:             addToList(CALLS, BR_CALLS)
67:             return CALLS
68:         end if
```

```
69:     end if
70:     UNIT_CALLS ← EDGESOUTOFUNIT(CG, U)
71:     ADDTOLIST(CALLS, UNIT_CALLS)
72: end procedure
```

Limitations

During our experiments we observed that developers do not always follow guidelines on how to check which button has been clicked. Instead of comparing their `ids` they compare objects, class names or even check in which activity they are. Such cases rely on dynamic executions and cannot be resolved statically.

4.11 Evaluation

As the first step evaluation of SAFAND UI analysis module we aimed to compare its abilities against the latest version of GATOR (March 2017), which is the only currently available static framework to analyze Android UI elements and bind them to their corresponding callbacks. We first ran SAFAND and GATOR on a small set of synthetic examples that cover many different features of the Android UI. We present the results of our analysis in Section 4.11.1. We conclude our comparison by briefly presenting the results on the Health Tracker app in Section 4.11.2.

In Section 5.1.4 and Section 5.1.5 we will show how program features extracted with the help of SAFAND UI analysis module can be used effectively to identify UI errors.

4.11.1 Comparison on Synthetic Samples

To compare which features are supported by SAFAND and GATOR, we carefully reviewed the Android documentation and analyzed several real apps. We crafted a set of synthetic samples to validate capabilities of each tools. The sample apps cover the following features:

- UI elements challenge — checks whether the analysis can capture UI elements declared in the code.

- Layouts challenge — checks whether the analysis can correctly model the layout structure of the activity. Different types of layout hierarchies are present, created both statically, via xml tags, and dynamically, in the code. The samples include merged layouts, fragments, menus, and dialogs.

- Listeners challenge — checks whether the app can correctly bind UI elements to listeners in the following cases: 1) an instance of a class that implements a listener, 2) an anonymous inner class, 3) a variable declared as a type of interface, 4) an activity itself implements the listener interface (See Listing 4.19), 5) via an xml layout attribute onClick.

- Text label challenge — tests whether the analysis handles text connected to a UI element. We consider the following string types: plain text, static, non-static and global fields, which are defined in Android resource files. We also include examples of string concatenations via the StringBuilder.

- Style challenge — checks whether user-defined styles are supported. Labels and listeners are both defined via XML tags inside a style file.

- API challenge — tests whether the analysis correctly associates UI element with APIs they trigger. We include samples that require context-sensitive inter-procedural to obtain the correct information.

Table 4.6 and Table 4.7 report the results of our analysis. GATOR correctly identified menus of two types. However, it could not find *Pop-up menu* and mistakenly assigned the *Contextual menu* to a wrong button. Finally, it does not support Navigation-Drop-Down menus as well. Regarding listeners, instead, (see Table 4.7) GATOR tends to over-approximate in most cases assigning too many listeners to UI elements.

4.11.2 Comparison on the Health Tracker app

On the Health Tracker app, see Figure 4.2, GATOR reported *9 listeners* bound to the *Send User Data By Email* button. Actually, it assignes the exact same set of listeners to each of the 7 buttons in Health Tracker app. Two additional buttons come from the *Dialog* invoked by one of the buttons. This dialog was correctly identified by GATOR, but the list of listeners is over-approximated, and its layout is completely blundered with the elements of the underlying activity. Thus, for the Health Tracker app GATOR could produce only the rough model of the activity, missing many details. In contrast, SAFAND generated the precise

Table 4.6: User Interface Elements Support.

UI Element	**GATOR**	**BACKSTAGE**
<include>xml tag	✓	✓
LayoutInflater	✓	✓
<fragment>xml tag	✗	✓
FragmentManager	✗	✓
Option menu	✓	✓
Pop-up menu	✗	✓
Contextual menu	✓	✓
Navigation-Drop-Down menu	✗	✓
Drawer Layout	✓	✓
Tab View (via ActionBar)	✗	✓
ViewPager		
xml defined	✓	✓
via PagerAdaptor	✗	✓
AlertDialog	✓	✓
Toast	✗	✓
Notification	✗	✓

Table 4.7: Listeners Support.

Listener Type	**GATOR**	**BACKSTAGE**
Layout xml file	✗	✓
Style xml file	✗	✓
Anonymous inner class	✶	✓
Menu callbacks	✓	✓
Listener interface	✶	✓
Shared superclass	✶	✓
Listener assigned in a loop	✶	✗

✓— listener correctly identified, ✗— listener not found,
✶— too many listener found, over-approximation

```
1   public class Main extends Activity implements OnClickListener {
2   @Override
3   public void onCreate(Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.main);
6         findViewById(R.id.button1).setOnClickListener(
7               new OnClickListener() {
8                     public void onClick(View arg0) {...}
9               });
10        findViewById(R.id.button2).setOnClickListener(new HClick());
11        findViewById(R.id.button3).setOnClickListener(this);
12  }
13  public void onClick(View arg0) {...}
14
15        private class HClick implements OnClickListener {
16              public void onClick(View arg0) {...}
17        }
18  }
19
```

Listing 4.19: Ways to register an event listener

activity model. It correctly assigned all listeners to the corresponding buttons. Along with handler binding it supplied each button with its label and produced a list of APIs reachable from each listener.

The over-approximation in UI hierarchy construction and callback handler binding by GATOR is caused by the missing support of dynamic dispatch recognition. For instance, for a set of subclasses, it reports a union of all layouts assigned by a method defined in a shared superclass, whereas layout id is redefined inside each subclass. The same over-approximation also happens for listeners implemented by an anonymous inner class. Namely, GATOR can not distinguish such listeners, i.e. if two UI elements have different listeners it will assign both to each of them.

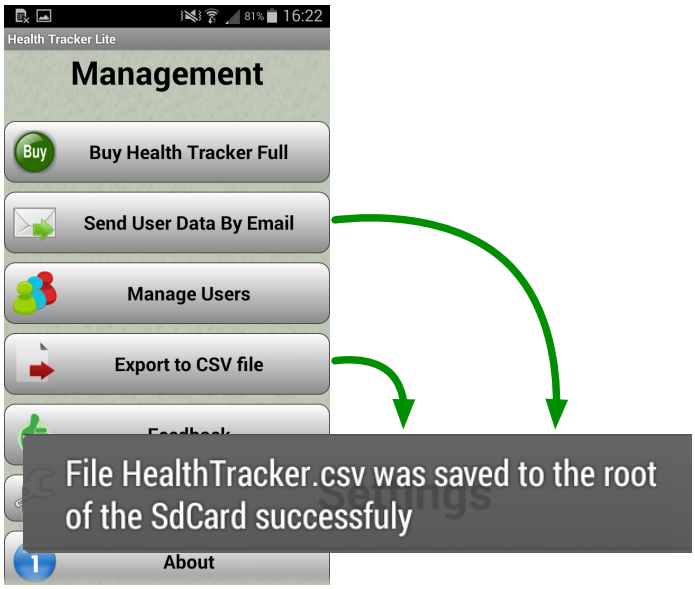

Figure 4.2: Health Tracker Lite app. The same message "File saved" is shown for export as well as for the email actions.

Chapter 5

Associating Graphical User Interfaces with Sensitive Behavior

5.1 Associating Graphical User Interfaces with Sensitive APIs

The next phase of the program analysis phase in SAFAND is to map the advertised behavior e.g., UI elements and their visible representation on the screen to the actual behavior that is triggered in the process of user interaction. In this section we use a notion of sensitive APIs (Section 2.3.2) as a proxy of the actual behavior.

5.1.1 Mining Reachable APIs

To mine APIs that are reachable from a corresponding callback, SAFAND performs the following steps:

1. SAFAND identifies *callbacks* from the UI analysis phase as discussed in Section 4.8, and sets them as entry points for the call-graph construction.

2. It builds the *call-graph* thanks to the Rapid Type Analysis algorithm (RTA), which limits the over-approximation by identifying the classes in the program that are possibly instantiated [15]. It should be mentioned that we did not write our own implementation of the algorithm and reused the built-in one version from SOOT. SAFAND, however, is fully configurable and supports Variable Type Analysis (VTA) [51] and Hierarchy Analysis (CHA) [17], as SOOT also supports them.

   Both VTA and CHA have their own advantages and disadvantages. CHA is quite fast in computing, but it over-approximates a lot and contains infeasible edges. VTA, in turn, is highly precise as it performs propagation of an abstract class object to represent all created objects of that class. Such heuristic of VTA, on the one hand, eliminates infeasible edges and makes call-graph more precise, but on the other hand it is time- and resource costly. Moreover, in the presence of missed libraries, VTA is quite conservative and truncates edges to them, as it cannot perform analysis for the missed code. People mostly use the stubbed version of Android platform classes which is shipped with Android SDK. Thus, all edges to the Android platform will be removed by VTA as there is no implementation of methods. The analysis of the whole Android platform is quite a resource consuming task and therefore is not applicable for real-world problems.

   RTA is between CHA and VTA in terms of the precision of the call-graph and time which is needed for its construction. That is why RTA has been chosen as a default algorithm, but one can easily change it.

3. For each callback it collects all *reachable Android API invocations* in the transitive closure of its call-graph by performing the context-sensitive analysis described in Section 4.10.

Indeed, the last item in the list above is not as simple as it sounds. During our experiments we found that call-graph construction algorithms contain some missing edges. The source of such misses is mostly related to the asynchronous events and complex chains of them. Yang et al. [59] discussed in their paper the problem of poor responsiveness of apps. And the main root-cause was the fact that all time- and resource consuming operations have to be excluded to background and asynchronous tasks. The current official Android documentation suggests the same [4]. That is why developers take care of the responsiveness of their apps as it directly correlates with the user satisfaction. The most critical case is shown in Listing 5.1. We do not include full implementation neither of Java Development Kit (JDK) nor of Android

platform. Therefore we cannot identify which methods will be invoked in the process of running `executeOnExecutor` method. The call-graph implementation of SOOT has the predefined knowledge for such cases, but this particular case is missed. SAFAND takes care of this case and manually points the analysis to `LongOperation:doInBackground` method. We omit for now methods like `onPreExecute` or `onPostExecute` for performance reasons and assume that they do not invoke any sensitive APIs and their job is merely to notify the user about the task. Such construction was used in *Tripwolf* app. Apart from `android.os.AsyncTasks`, SAFAND supports also classes that implement `java.lang.Runnable` and all kinds of their submits.

```
1  public class MyActivity extends Activity(){
2      public void onCreate(Bundle savedInstanceState){
3          runTask();
4      }
5      private void runTask(){
6          LongOperation longOp = new LongOperation();
7          longOp.executeOnExecutor(Executors.newSingleThreadExecutor(), "");
8      }
9
10 private class LongOperation extends AsyncTask<String, Void, String> {
11     @Override
12     protected String doInBackground(String... params) {
13         Log.w("MyTag", "Long operation done");
14         return "Executed";
15     }
16
17     @Override
18     protected void onPostExecute(String result){
19         update UI
20     }
21 }
```

Listing 5.1: Running asynchronous task by using Executor

The code included in apk files often includes libraries. As a consequence, many API invocations that SAFAND would identify with its analysis belong to third party libraries. Our initial manual evaluation of the analysis results showed that many of these library invocations are infeasible in practice. This is due to the over-approximation of static analysis as libraries usually have checks for some conditions. Those checks appear in the code due to wide audience of the libraries. Vendors release libraries which contains a lot of different functionality and each developer uses only the small part of it. That is why the libraries

code ensures that only desired functionality is triggered. Unfortunately, static analysis is unable to cope with these conditions and thus, such code gives more disadvantages than advantages. To diminish this problem, we decided to limit the analysis only to the application code, thus excluding libraries from the analysis.

To achieve this goal, we filter classes based on their package name. Therefore, for instance, when analyzing the Twitter app, we would focus only on classes belonging to the `com.twitter` package.

Furthermore, we also included a parameter to limit the depth of the call-graph analysis starting from entry points. In fact, the farther the code from the entry points, is the more likely it contains infeasible invocations. The default settings, which are what we used in our experiments, consider only invocations to the Android API that are in methods with the maximum depth of five calls from the corresponding callback.

Finally, after obtaining the mapping between the UI element and its APIs we conducted post-processing to resolve types of Intents (Section 3.2.2) and Sensitive Resources (Section 3.2.1) in the same way as by extracting sensitive data flows in Section 3.3. Precise values of Intents and Sensitive Resources provide more expressive information on what behavior is being triggered.

5.1.2 Dissolving Intercomponent Calls

As we discussed in Section 2.1, Android application consists of a set of screens and each screen usually performs a separate action. The typical example is a process of making an order. In one screen a user selects the product, then by clicking on *"Proceed to the Payment"* button he reaches the second screen with payment details and confirmation. Activities are fully isolated in Android and the only way to transfer the data from the first screen to the second one is to leverage the inter-component communication mechanism by means of sending Intents.

As we discussed in Section 3.2.2, intents can be explicit and implicit. Explicit intents are responsible for communication between activities of the same application. In this section we describe the analysis of explicit intents and the algorithm to dissolve them. By dissolving explicit intents we can identify the target screen and thus, gather APIs that are invoked in its lifecyle methods (Section 2.1.3). In other words, we gather APIs from others screens that are tight to the selected product and *"Proceed to the Payment"* button from the first screen.

In order to achieve this, we analyze the class name of the corresponding Intent API (Table 5.1) during API analysis. If the class name is the app's activity,

Table 5.1: Signatures of APIs that perform inter-component calls

android.content.Context: void startActivities(android.content.Intent[],android.os.Bundle)
android.content.Context: void startActivities(android.content.Intent[])
android.content.Context: void startActivity(android.content.Intent)
android.content.Context: void startActivity(android.content.Intent,android.os.Bundle)
android.content.Context: void startActivityForResult(android.content.Intent,int)
android.content.Context: void startActivityForResult(android.content.Intent, int,android.os.Bundle)
android.content.Context: void startActivityFromChild(android.app.Activity, android.content.Intent,int,android.os.Bundle)
android.content.Context: void startActivityFromChild(android.app.Activity, android.content.Intent,int)
android.content.Context: void startActivityFromFragment(android.app.Fragment, android.content.Intent,int,android.os.Bundle)
android.content.Context: void startActivityFromFragment(android.app.Fragment, android.content.Intent,int)
android.content.Context: void startActivityIfNeeded(android.content.Intent, int,android.os.Bundle)
android.content.Context: void startActivityIfNeeded(android.content.Intent,int)
android.content.Context: android.content.ComponentName startService(android.content.Intent)
android.content.Context: boolean bindService(android.content.Intent, android.content.ServiceConnection,int)

Table 5.2: Signatures of APIs that are responsible for triggering Activity and Service

android.app.Service: void onStart(android.content.Intent,int)
android.app.Service: int onStartCommand(android.content.Intent,int,int)
android.app.Service: android.os.IBinder onBind(android.content.Intent)
android.app.Service: void onRebind(android.content.Intent)
android.app.Activity: void onCreate(android.os.Bundle)

we manually jump to its `onCreate` method and continue API search. Each such jump increments the `depthComponentLevel` counter of further APIs. We do the same for services, i.e. if we reach an API that is responsible for interaction with the service, we extract the name of the service and emulate calls to its lifecycle methods (Table 5.2). We use here only those lifecycle methods that can receive an `Intent` and are very popular.

5.1.3 Limitations

The main limitation of the API mining phase is missed or overapproximated edges of the call-graph. Due to its nature, RTA algorithm fails to truncate infeasible edges. The current implementation of SOOT's call-graph does not assume all possible cases of Java Threads, Executors and their combinations.

5.1.4 Graphical User Interfaces and their APIs

The Android Apps Dataset and its UI Elements

BACKSTAGE needs a large number of apps in order to point out relevant outliers. Thus, we collected a big dataset that includes the top 600 Android apps in each category of the Google Play Store as displayed in the US in July 2016. We chose the US market in order to maximize the number of apps using English as the main language. Instead of crawling the Google Play store to download the app APKs, we retrieved them from ANDROZOO [2].

As a result of this step we collected 12,000 apps, which is less than one would originally expect. This is because either Google Play lists less than 600 top apps for some categories, or ANDROZOO does not have the desired APK.

Given that BACKSTAGE detects anomalies in the graphical user interface, we filtered out the apps with little to no GUI, as well as the apps where UI is mainly made of drawings on canvases (e.g. interactive games). As a heuristic to filtering out the apps we could not analyze, we computed the ratio between the *number of layout files* and the *number of activities* in the app. We ignored the apps with the ratio under 70%. The intuition behind this heuristic is that there is usually one layout file for each activity. When this is not the case, it means that the activity does not have a UI that can be analyzed with our approach (i.e. there are no labels with text, buttons, etc.). In essence, this heuristic led us to ignore all apps listed in the GAMES, ANDROID_WEAR, COMICS, and APP_WIDGET categories. As a result we removed from our dataset those apps that are out of the scope of this work (924 apps).

Moreover, given that BACKSTAGE semantically groups similar text labels, it requires on a dataset in one language. We chose English for its prevalence, and we relied on *LangDetect* [42] to detect the language of the GUI text. Similarly to the previous heuristic, we removed from our dataset the apps with less than 70% of the resources in English. 203 apps were removed in this step.

Analysis Settings

Experiments have been conducted on the machine with with 730 GB of RAM, 64 Intel Xeon CPU cores and Java 8. We run 4 instances of the analysis at once and each instance used 40 GB of RAM and 16 CPU cores.

In order to decrease the search space of APIs we decided to search for them only in the code that relates to the same package as the app (refer to Section 5.1.1 for more details).

Each UI mining pack (Section 4.2) as well as reachability analysis of each callback (Section 5.1.1) had a timeout of 1 minute. The overall timeout for the analysis of an app was 3 hours. We could not analyze 356 apps due to this timeout, and we thus removed them from the dataset too.

Analysis Results

Despite these filtering processes, our dataset remains significantly large: We analyzed in total 10326 apps and 292674 UI elements, which included 9375 *unique UI element types* (Button, ImageButton, RadioButton, ToggleButton, CheckBox, ImageView, EditText, and 9368 additional custom button types), and extracted 47677 unique text labels. Figure 5.1 shows the distribution of UI elements in our dataset.

We analyzed more than 373000 unique callbacks, where the analysis identified over 4.5 million API invocations, in which they refer to 3.618 unique APIs.

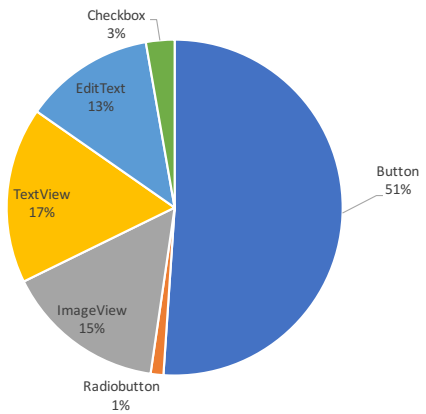

Figure 5.1: Distribution of UI elements in apps

Figure 5.2 motivates the need of such complicated and sophisticated UI analysis like BACKSTAGE employs. Over 85% of callbacks are defined in the code at runtime. It means that parsing layout XML files of activities gives us only 15% of overall callbacks, which is just a tiny portion of them.

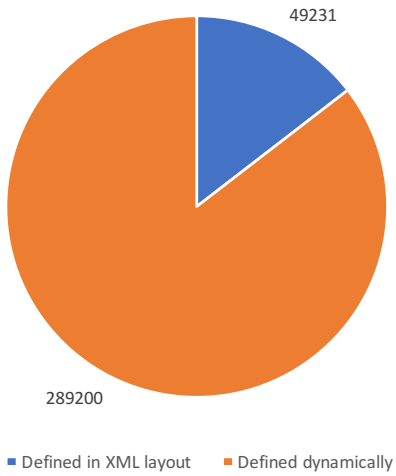

Figure 5.2: Ratio of callbacks defined in the XML layout file and dynamically in the code

A large number of unique UI element types is suprising as developers love to extend custom Android types as *Button* because of their limited functionality. However, such UI elements are still *Buttons*. Therefore we group UI types by their names and assign, for example, `com.starbucks.mobilecard.controls.SBButton` to the *Button* type. We do this for the most known standard UI elements: *Button, RadioButton, ImageView, TextView, EditText, CheckBox*. Figure 5.3 shows the distribution of callbacks per UI type after applying this heuristic.

As we discussed in Section 4.8, the most typical callback is *onClick*. However, it does not mean that it is prevalent in all UI types. The ratio of callbacks for the mentioned UI types is illustrated in Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7 and Figure 5.9. These ratios show that just tracking of *onClick* callback is not enough which motivates the list of callbacks that we presented in Section 4.8.

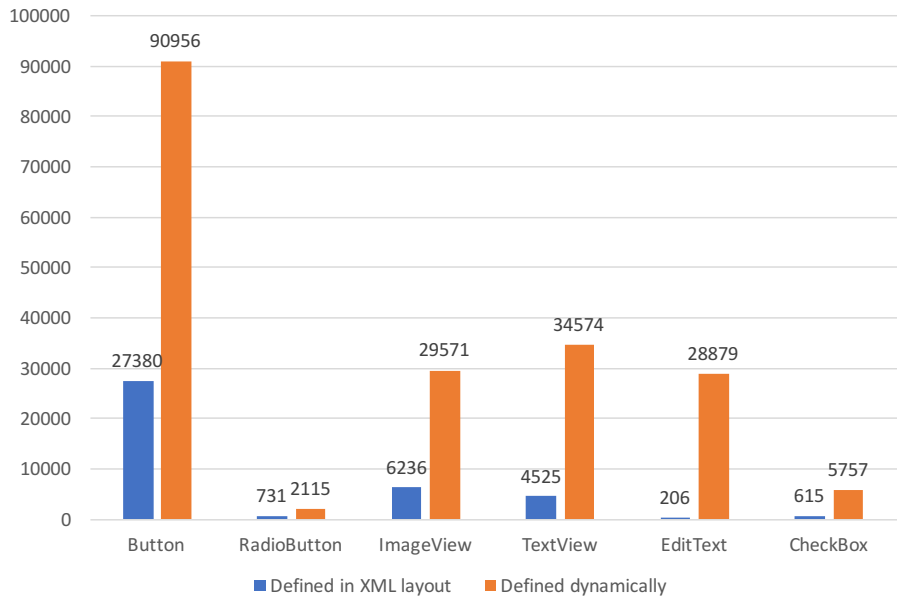

Figure 5.3: Ratio of callbacks defined in the XML layout file and dynamically in the code by UI element type

It is interesting to see that not only *Buttons* can invoke APIs. Figure 5.10 shows that buttons, indeed, do not take the first place in this. It turns out that *TextView* and *ImageView* invoke more APIs on average which clearly tell us that UI analysis should assume a big variety of UI elements as BACKSTAGE does.

As we already mentioned, the whole analysis of one app has been run on 8 CPUs and has been fully parallelized. The runtime performance of the UI analysis phase is illustrated in Figure 5.11. For most of the apps UI analysis phase has been terminated within 5000 seconds (83 minutes). UI analysis of Android Twitter app took about 10 hours. This app is the exception because of an complexity of its code.

In contrast to UI analysis, reachability analysis of APIs is very fast (Figure 5.12). The main bottleneck is building a call-graph. Once it is built, the analysis just walk over its edges and searches for APIs. The most time expensive app for the reachability analysis is Google Docs app and the analysis took about 100 minutes. The reachability analysis of the most apps has been terminated

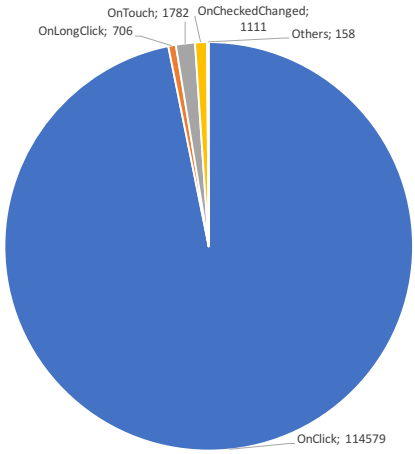

Figure 5.4: Ratio of callbacks for Buttons

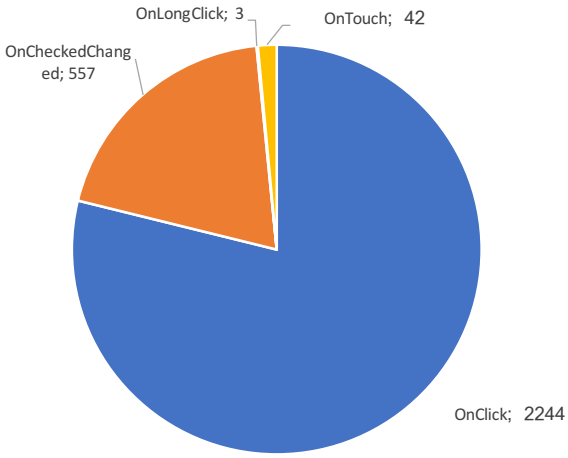

Figure 5.5: Ratio of callbacks for RadioButtons

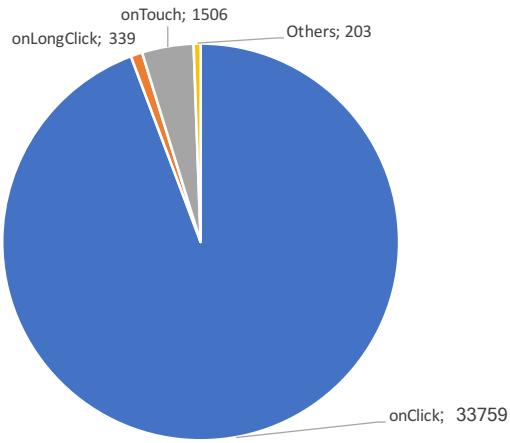

Figure 5.6: Ratio of callbacks for ImageViews

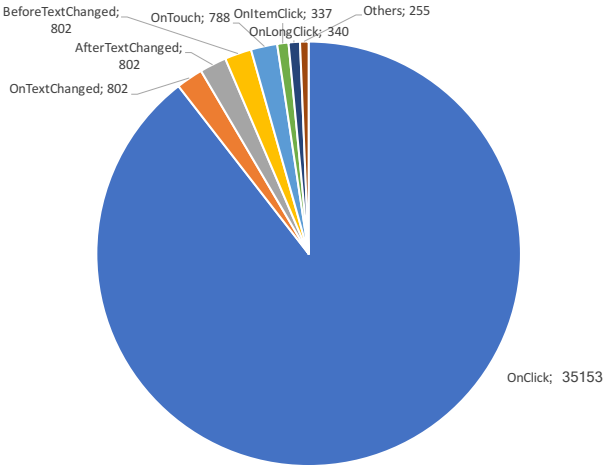

Figure 5.7: Ratio of callbacks for TextViews

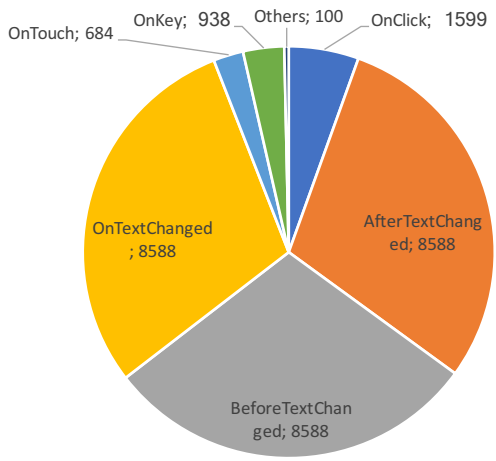

Figure 5.8: Ratio of callbacks for EditText

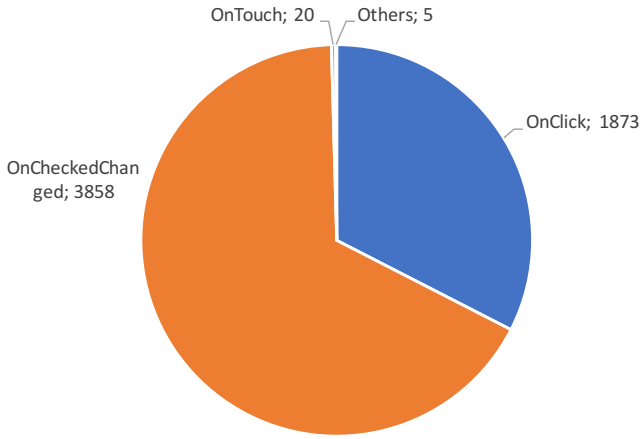

Figure 5.9: Ratio of callbacks for CheckBoxes

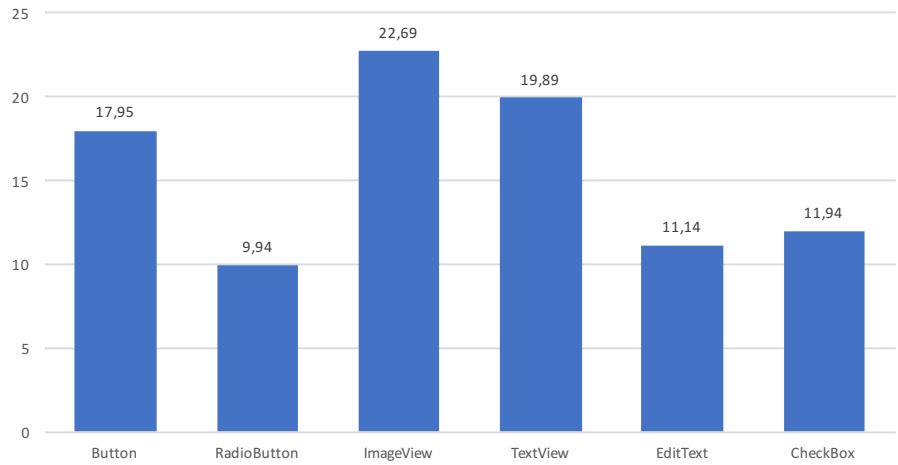

Figure 5.10: Average number of reachable API invocations per UI element

in 6 minutes, which is quite a good result.

The most important thing is that BACKSTAGE can easily parallelize the analysis. And by having access to very powerful servers one can easily analyze even such big apps in a few minutes.

Each callback usually leads to one or more Android APIs. As we discussed in the Section 5.1.1, we leverage the SUSI list of Android APIs and also their categorization. However, there are 108561 callbacks that do not have any API invocation from the list. It means that a particular UI element does not interact with the Android system but can perform some internal app-related operations. Figure 5.13 illustrates the top 50 API SUSI categories sorted by their occurrences. It should be mentioned that in case an API falls to NO_CATEGORY, we apply the following scenario: we extract the package name of the corresponding API and use it as a SUSI category. We assume that APIs from the same package should perform similar operations. Such categorization helps us to understand what behavior is being triggered in most cases.

Figure 5.13 shows that there are only 9 sensitive categories of APIs in the top 50. Moreover, DATABASE_INFORMATION and NETWORK are very general and usually it is very hard to understand just from statistics which data is being sent to the Internet or being stored in the database. That is why we deal not only with sensitive APIs there, but assume all kind of Android APIs.

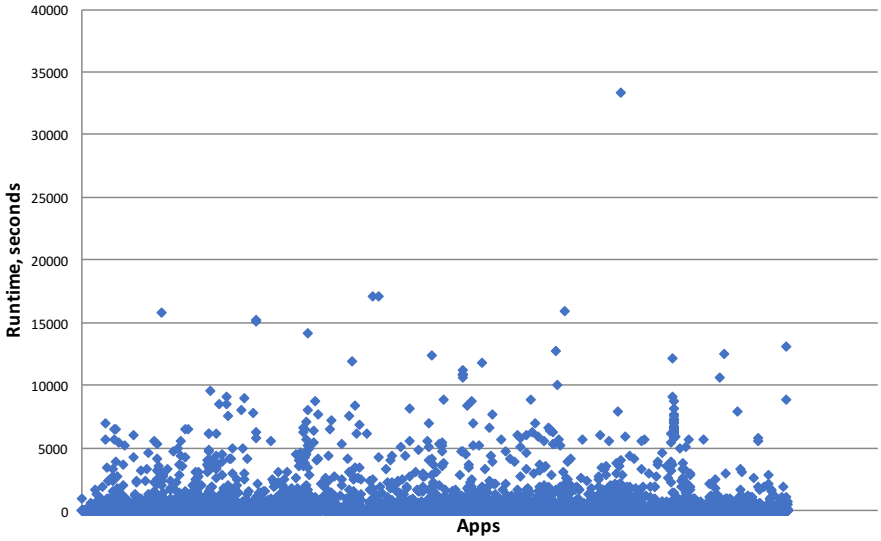

Figure 5.11: Runtime of UI analysis phase

5.1.5 Mining anomalies

In this section we will discuss whether APIs on itself can be used to identify applications with suspicious and malicious behavior (**RQ3**). Moreover we will provide the background why a combination of *declared* and *actual* behavior can be effectively used to identify abnormal apps. The content of the section is primarily based on the BACKSTAGE technical report by Avdiienko et al.[13].

The *declared* behavior of UI elements the typically constructed from what a user sees on the screen: If a UI element says "Print", "Save", "OK", "Close", or "Cancel", the experience with other programs using these labels gives users an idea of what to expect. The *actual* behavior is everything that happens after the user's interaction with the UI element: If the user clicked on the "Save" button, the actual behavior is "data has been saved to the database" and "the corresponding pop-up message has been shown".

To solve the aforementioned problem we have developed a tool called BACKSTAGE. SAFAND handles the program analysis work of BACKSTAGE and provides a possibility to identify the expected result of an UI element. The result could be either *visible* or *invisible* to the user. Opening another window or poping-

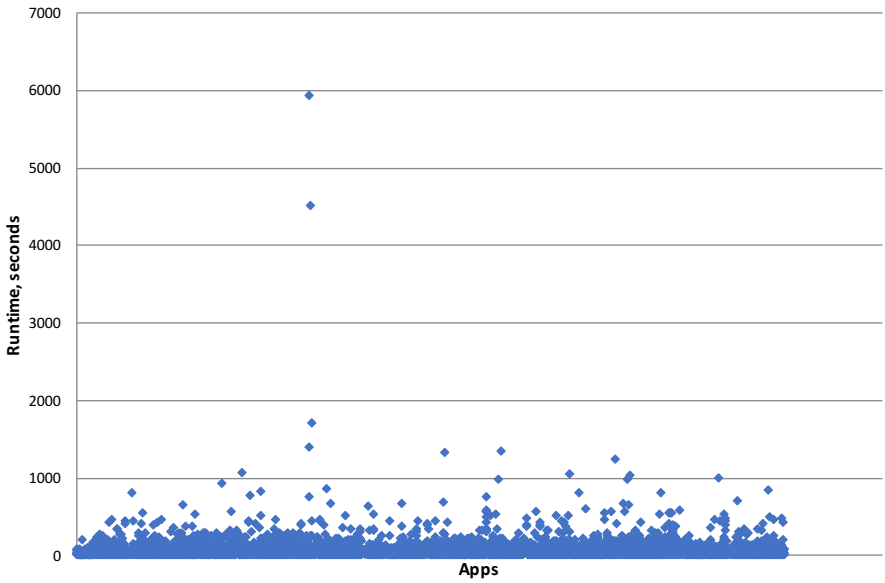

Figure 5.12: Runtime of Reachability analysis phase

up an alert dialog are examples of the *visible* result. Whereas all background activities, which are associated with the UI element, are typically considered as an instance of the *invisible* behavior. In other words, the user could say that they are running behind the scene, i.e., in a *backstage*.

After performing the described program analysis, we perform two additional steps in order to identify anomalies in GUIs:

Cluster Analysis. From associated text of all UI elements, we clustered their verbs and nouns into 250 *concepts*—clusters of words with the minimal *semantical distance* using the WORD2VEC model [39]. For each UI element, we determine the distance between its text and the concepts. A button named "Share", for instance, would be semantically close to the concepts of "friend" (to share) and "finances" (a share). Figure 5.14 shows the labels of all UI elements that form the "Signup" concept.

For each button, we also extract the APIs used. Figure 5.15 shows the ANDROID packages used by Signup buttons. The "normal" behavior of a

Signup function is to access the network via `android.net` or `org.apache.http`. Several signup functions also access `android.telephony` to access the country code of either the current network or the inserted SIM card. The `android.location` package also is frequently used—but only to access the current local time.

Outlier Detection. For each concept, we use outlier detection to identify the UI elements that invoke uncommon APIs, indicating differing and possibly unexpected behavior. Accessing the current precise location is rarely used in the Signup cluster—and thus, the TRIPWOLF Signup button is flagged as an anomaly.

Since our analysis uses most widely used apps, with a high level of maturity, visible GUI errors as those prevented by BACKSTAGE are typically quickly detected, reported, and fixed. We thus switch to a well-established scheme used to evaluate testing techniques. To evaluate how BACKSTAGE fares as it comes to *general* GUI mismatches, we devised a setting in which we would create *synthetic GUI errors* (mutations). Specifically, we would take existing buttons and change their labels such that they would no longer match the APIs used. Then we would then evaluate whether BACKSTAGE detects these mutations as anomalies.

In detail, we implemented the following mutations, modeled after *mutation* and *crossover* operations in genetic algorithms:

Label replace. Given a UI element e, we replace its label with a label from another concept. This label is chosen in two ways:

- *randomly*—that is, out of all the labels encountered across all apps. This simulates a random error in labeling a UI element.
- *high distance*—that is, from a concept that is semantically distant with respect to the original label.

Crossover. Given a UI element e, we would swap its label with the label of a random UI element $e' \neq e$ in the same app. This simulates the error of a developer confusing two UI elements, swapping callbacks to UI elements within the same application. The mismatches created by crossover would be more subtle, as they occur within the same range of app functionality.

All these mutations are applied on the data alone; we do not actually change the code of existing apps. In terms of the difference induced by the mutation, we would assume "high distance" mutations to be the easiest to detect, followed by random mutations, and finally crossover.

Table 5.3: BACKSTAGE accuracy for "random" label replace mutations.

	Classified as			
Input	**Abnormal**	**Normal**	**Total**	
Mutant	TP = 3369	FN = 1630	4999	*Precision* = 75%
Correct	FP = 1100	TN = 4056	5156	*Recall* = 67%
Total	4469	5686	10155	*Accuracy* = 73%
				Specificity = 79%

Table 5.4: Accuracy for "high distance" label replace mutations.

	Classified as			
Input	**Abnormal**	**Normal**	**Total**	
Mutant	TP = 3528	FN = 1471	4999	*Precision* = 76%
Correct	FP = 1096	TN = 4060	5156	*Recall* = 71%
Total	4624	5531	10155	*Accuracy* = 75%
				Specificity = 79%

The overall results of BACKSTAGE are summarized in the confusion matrices in Table 5.3, Table 5.4, and Table 5.5. Let us discuss random mutations first, and then contrast the results with the alternative mutation schemes.

As seen in Table 5.3, of the 4,999 mutants fed into BACKSTAGE, 3,369 are correctly classified as being abnormal, which results in the recall rate of 3369/4999 = 67%. As expected, high distance mutations (Table 5.4) have a higher chance (71%) of being detected. Even if the programmer confuses two buttons (Table 5.5), BACKSTAGE detects every second such mistake. All these results should be interpreted from the standpoint that to the best of our knowledge, there is no other approach which would detect such mismatches.

A high recall means little if the precision is low; that is, if the UI elements reported by BACKSTAGE contain many false positives. For "random" mutations (Table 5.3), the precision is 75% meaning that three out of four UI elements reported will actually be abnormal; high distance mutations fare even slightly better, with 76%. For "crossover" mutations (Table 5.5), BACKSTAGE still has the precision of 69%, a bit more than two out of three UI elements reported will be true anomalies. This high precision makes BACKSTAGE a practical tool and proves that associations between UI and its program behavior is a good indicator for incorrect behavior of apps.

Table 5.5: BACKSTAGE Accuracy for crossover label mutations.

	Classified as			
Input	**Abnormal**	**Normal**	**Total**	
Mutant	TP = 2290	FN = 2475	4765	*Precision* = 69%
Correct	FP = 1026	TN = 4121	5147	*Recall* = 48%
Total	3316	6596	9912	*Accuracy* = 65%
				Specificity = 80%

5.2 Associating Graphical User Interfaces with Sensitive Data Flows

The mere presence of a particular dataflow in the application does not necessarily means that the app has suspicious behavior. Moreover, it is a quite complicated task to understand why the dataflow is present in the application, and, finally, to understand which action triggers it. For example, if the SMS sending functionality is being triggered by the UI element (e.g., by the user interaction) than, most probably, such dataflow is expected. In Section 3.3 we already showed that flows of sensitive data is a good property to describe the behavior of the application. Indeed, it is important to know not only the presence of the dataflow in the application, but also its intention and its trigger. In other words, the location of the user is leaked to the Internet only when he presses the button with the label *"Locate"*. It is sometimes necessary to know how your location is used when you press this button. The location information can be used in multiple ways and the exact way is essential to identify whether the usage is legitimate or not. If user location is just obtained for saving her route and leaks only to the internal database for history purposes, it will not be assumed as a harmful. Moreover, associating UI elements and APIs (Section 5.1.5) is not so helpful either, as it can not tell the user in what way her location is used.

The workflow is described in Algorithm 13. SAFAND first starts with the set of callbacks obtained from the UI analysis phase (Section 4.8). For each callbacks it perform its context-sensitive analysis (Section 4.10) and extract all reachable source APIs. We again rely on the SUSI (Section 2.3.2) categorization of Android APIs to sources and sinks. Next, it saves the mapping of sources to their corresponding callbacks. SAFAND does not just save APIs as a sources, it stores all needed information to make APIs unique. It can differentiate invocations of `TelephonyManager:getDeviceId` inside different callbacks for example. Next, SAFAND invokes FLOWDROID and provides collected sources as starting points

Algorithm 13 Strategy of identifying data flows and mapping them to respective UI elements.

Require: Callbacks C, CallGraph CG
Ensure: Mapping of UI elements to data flows $UIFlow$

```
1: procedure DataflowAnalysis(T, C)
2:     CBS ← callbacks to sources
3:     AllS ← all sources
4:     for CB in C do
5:         S ←GetSourcesContextSensitive(CB,CG)
6:         put(CBS, CB,S)
7:         add(AllS,S)
8:     end for
9:     SubmitSourcesToInfoflowAnalysis(AllS)
10:    Flows ←RunInfoflowAnalysis
11:    UIFlow ←MapUiToFlows(CBS,Flows)
12:    return UIFlow
13: end procedure
14: procedure MapUiToFlows(CBS,Flows)
15:    UiFlow ← mapping of callbacks to flows
16:    for F in Flows do
17:        S ←GetSource(F)
18:        CB ←GetCallbackForSource(CBS,S)
19:        put(UiFlow,CB,F)
20:    end for
21:    return UIFlow
22: end procedure
```

of the dataflow analysis. FLOWDROID performs forward taint propagation and thus starts from sources and searches for their connections with sinks. When the dataflow analysis is done, SAFAND obtains the set of dataflows and for each dataflow it checks to which callbacks its source belongs. By using such heuristic SAFAND can precisely match dataflows to their corresponding callbacks, and consequently to UI elements that initiate them.

In this section we fully rely on FLOWDROID to obtain dataflows and do not alter it. The only modification is the set of sources to start and the post-processing phase which resolves values of sensitive resources and intents as explained above.

5.2.1 Evaluation

Practical evaluation of associations between UI and sensitive dataflows are out of scope of the thesis. As mentioned in Chapter 3, dataflow analysis is quite resource and time-expensive. This approach goes one level of granularity deeper in comparison with the approach from Section 5.1 as sensitive dataflows describe the behavior of an application more precise than just the presence of particular APIs.

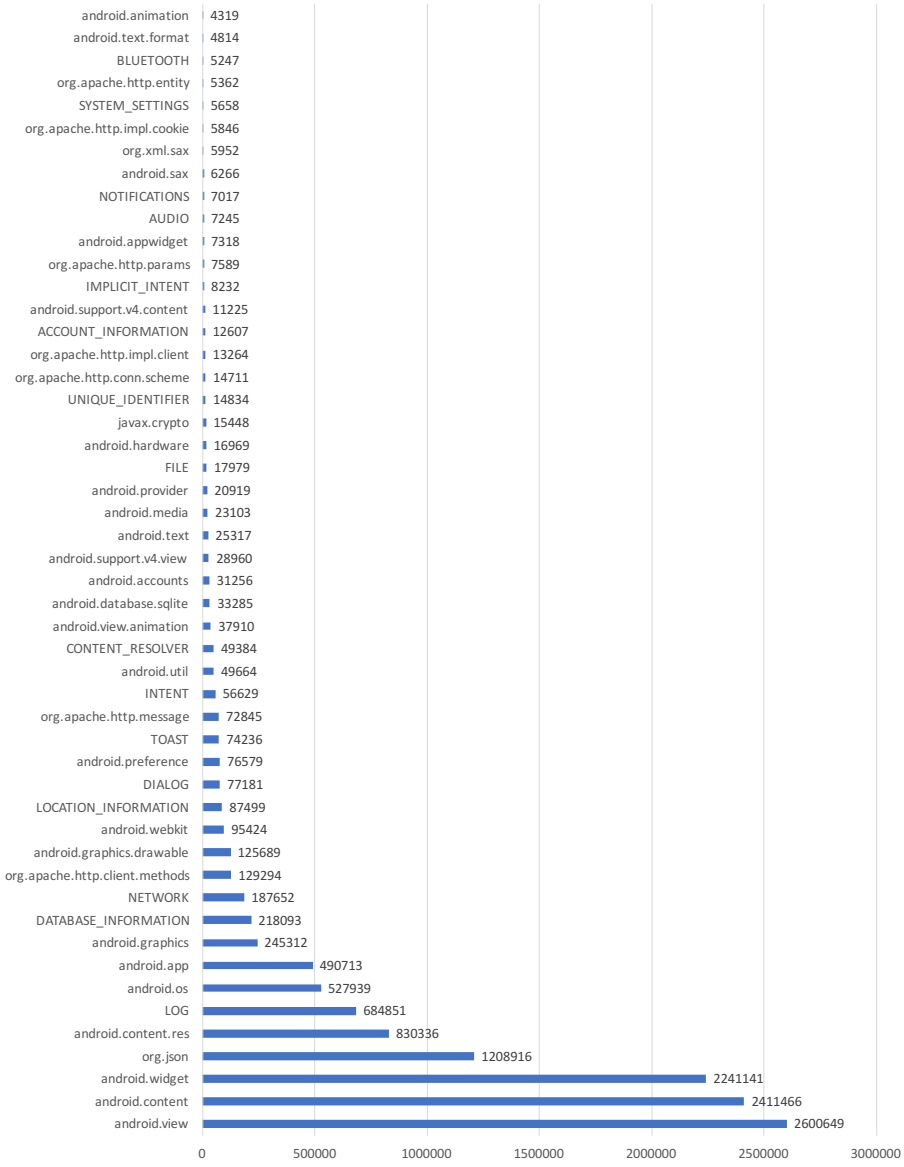

Figure 5.13: Top 50 APIs reachable from UI elements

Figure 5.14: Labels of semantically related UI elements, forming a "Login" concept. Based on its label and its context on the screen, the "Join TRIPWOLF" Signup button is associated with this concept.

Figure 5.15: The ANDROID API packages used by the functions triggered by the UI elements in Figure 5.14. "Normal" functions of login buttons include accessing the Internet and switching to a new screen.

Chapter 6

Conclusion and future work

Nowadays mobile devices are an essential part of people's life. Social networks, corporate and personal mailboxes, videos, photos and even payments are possible and accessible via modern pocket devices. Such impetuous break-through in technology indeed introduces a lot of security and privacy risks for end-users.

Each mobile device has dozens of installed apps and all of them are ubiquitous. End-users sometimes can not observe and control what these applications are actually doing on their devices. Therefore cybercriminals develop malicious payloads for well-known applications or create completely new and shiny apps which steal sensitive user data such as passwords, contacts and payment information.

In the thesis we addressed aforementioned issues and presented the set of static analysis approaches to analyze and describe behavior of Android applications called SAFAND. SAFAND encapsulates program analysis techniques of MUDFLOW and BACKSTAGE tools.

MUDFLOW analyses and learns "normal" flows of sensitive data from trusted applications to detect "abnormal" flows in possibly malicious applications. For example, it reports if an application accesses the address book of a device and this data goes to the network. The approach is effective in detecting novel attacks, learning from benignware only, as well as recognizing known attacks, learning from benign as well as malicious samples.

Despite data flow analysis of MUDFLOW being expensive for real-world apps, it has been shown that the flow of sensitive data is a useful abstraction not only for automatic classification, but also for end users to understand what an app does with sensitive data.

The presence of a particular dataflow does not necessarily mean that the app is malicious. Developers of modern applications leverage a huge set of Android APIs to make their apps more attractive for end-users. Thus, it is not easy to identify malicious applications just based on their flows of sensitive data.

What actually matters is whether the user is aware of such behavior and at the end agreed on it. To address this issue, we extended our analysis from just API to UI level aa well. The idea is to connect UI of Android applications with their corresponding behavior. For example, it is important to know whether the address book was accessed and the data was been sent to the network by clicking on a *Send* button or the action was executed somewhere in the background without user's notice.

BACKSTAGE is the first approach that generally checks the *advertised* functionality of UI elements against their *implemented* functionality. To this end, BACKSTAGE analyzes thousands of existing text and context UI elements shown to the user, clusters them by common concepts, and in each cluster detects *outliers*—that is, UI elements that use different APIs than the others. This approach is general and effective: In our evaluation, BACKSTAGE was able to effectively identify GUI behavior mismatches with high accuracy.

We proved that presented program analysis can be effectively used to identify applications with suspicious behavior.

Despite presented successes, there are still lots of opportunities for improvement.

Our own future work will focus on the following topics:

- To fool MUDFLOW, malware writers could use reflection, native code, self-decrypting code, or other features that challenge static analysis. Usage of such techniques in combination with sensitive data, however, would be unusual for benign apps. We are investigating analysis techniques that would detect such obfuscation techniques as anomalies.

- While static taint analysis across components and intermediate data storages is difficult, it is not fundamentally impossible. We want to design analysis techniques specifically tailored to app-wide and system-wide data flows as found in Android.

- Incorporating our earlier CHABADA work [24], we want to associate flows with app descriptions, detecting anomalies within specific application domains such as "travel", "wallpapers", and likewise.

- Where static analysis is challenged, combinations of automated test generation and dynamic flow analysis may prove to be helpful alternatives. We are investigating such combinations in conjunction with static analysis to combine the strengths of both static and dynamic flow analysis.

- Despite the ease of static analysis, we are considering using additional dynamic analysis and exploration to assess dynamic features. Most notably, we want to *validate* anomalies as reported by BACKSTAGE by creating test cases that demonstrate the actual API access.

- Detecting a mismatch of UI element label and its behavior allows for automatic suggestions of better fitting labels. One idea we are investigating is to identify labels of UI elements that use similar APIs and to suggest them as automatic repairs, e.g., "This button should be named 'Send'."

- Besides ANDROID apps, there are several other domains with programs whose GUIs could be mined, such as desktop applications.

- Besides looking for anomalies between text and behavior, one might also examine anomalies in visual presentation (for example, "The 'Send' button should be highlighted"), layout, process, or visual images. This would open a door to general automatic anomaly detection and recommendations for GUI design.

Bibliography

[1] An official android documentation of activity lifecycle. https://developer.android.com/reference/android/app/Activity.html.

[2] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. Androzoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (New York, NY, USA, 2016), MSR '16, ACM, pp. 468–471.

[3] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 258–261.

[4] Android documentation: Keeping your app responsive. https://developer.android.com/training/articles/perf-anr.html.

[5] Requesting permissions at run time. https://developer.android.com/training/permissions/requesting.html.

[6] Report: 97safe. https://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/.

[7] Providing resources. android developers documentation. https://developer.android.com/guide/topics/resources/providing-resources.html.

[8] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 259–269.

[9] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: Analyzing the android permission specification. In *Proceedings of CCS* (2012), pp. 217–228.

[10] Avdiienko, V. Mining patterns of sensitive data usage. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 891–894.

[11] Avdiienko, V., Kuznetsov, K., Calciati, P., Caiza Román, J. C., Gorla, A., and Zeller, A. Calappa: A toolchain for mining android applications. In *Proceedings of the International Workshop on App Market Analytics* (New York, NY, USA, 2016), WAMA 2016, ACM, pp. 22–25.

[12] Avdiienko, V., Kuznetsov, K., Gorla, A., Zeller, A., Arzt, S., Rasthofer, S., and Bodden, E. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 426–436.

[13] Avdiienko, V., Kuznetsov, K., Rommelfanger, I., Rau, A., Gorla, A., and Zeller. Detecting behavior anomalies in graphical user interfaces. Tech. rep., 2016.

[14] Backes, M., Bugiel, S., and Derr, E. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 356–367.

[15] Bacon, D. F., and Sweeney, P. F. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1996), OOPSLA '96, ACM, pp. 324–341.

[16] Comparing apples and googles: The app store vs. google play (infographic). http://venturebeat.com/2013/07/17/comparing-apples-and-googles-the-app-store-vs-google-play-infographic/.

[17] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (London, UK, UK, 1995), ECOOP '95, Springer-Verlag, pp. 77–101.

[18] Google play: Developer policy center. https://play.google.com/about/developer-content-policy/.

[19] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[20] Official android documentation of fragments. https://developer.android.com/guide/components/fragments.html.

[21] Gdata: Mobile malware report. https://www.gdata.nl/fileadmin/user_upload/Presse/Netherlands/Whitepaper_MMWR_EN_02-2014.pdf.

[22] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2012), TRUST'12, Springer-Verlag, pp. 291–307.

[23] Google play: an official android marketplace. https://play.google.com/store.

[24] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, June 2014), ICSE 2014, ACM, pp. 1025–1035.

[25] Hammer, C. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3.

[26] An official web-page of heros project. https://github.com/Sable/heros.

[27] Huang, J., Li, Z., Xiao, X., Wu, Z., Lu, K., Zhang, X., and Jiang, G. Supor: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), USENIX-SEC'15, USENIX Association, pp. 977–992.

[28] Huang, J., Zhang, X., and Tan, L. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (2016), FSE '16. to appear.

[29] Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B. Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1036–1046.

[30] How to use icons to support content in web design. https://www.smashingmagazine.com/2009/03/how-to-use-icons-to-support-content-in-web-design/.

[31] ios mobile platform. https://www.apple.com/ios/ios-10/.

[32] Just, S., Cleary, A., Shirley, B., and Hammer, C. Information flow analysis for javascript. In *Proc. of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients* (2011), PLASTIC '11, pp. 9–18.

[33] Klieber, W., Flynn, L., Bhosale, A., Jia, L., and Bauer, L. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (New York, NY, USA, 2014), SOAP '14, ACM, pp. 1–6.

[34] Kuznetsov, K., Avdiienko, V., Gorla, A., and Zeller, A. Checking app user interfaces against app descriptions. In *Proceedings of the International Workshop on App Market Analytics* (New York, NY, USA, 2016), WAMA 2016, ACM, pp. 1–7.

[35] Lam, P., Bodden, E., Lhoták, O., and Hendren, L. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)* (Oct. 2011).

[36] Li, L., Bartel, A., Klein, J., Traon, Y. L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. I know what leaked in your pocket: uncovering privacy leaks on Android apps with static taint analysis. *arXiv 1404.7431* (2014).

[37] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, ACM, pp. 756–761.

[38] Martin, W., Sarro, F., Jia, Y., Zhang, Y., and Harman, M. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering (TSE)* (2016).

[39] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[40] Implementation of the intent resolving algorithm. https://github.com/uds-se/soot-infoflow-android/blob/develop/src/st/cs/uni/saarland/de/AnalyzeIntents.java.

[41] Implementation of the uri resolving algorithm. https://github.com/uds-se/soot-infoflow-android/blob/develop/src/st/cs/uni/saarland/de/UriFinderSwitch.java.

[42] Nakatani, S. Language detection library for java, 2010.

[43] Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., and Wang, X. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), USENIX-SEC'15, USENIX Association, pp. 993–1008.

[44] An official android documentation of platform architecture. https://developer.android.com/guide/platform/index.html.

[45] An official web-page of pscout project. http://pscout.csl.toronto.edu/.

[46] Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium* (2014), NDSS'14.

[47] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95* (1995), pp. 49–61.

[48] Rommelfanger, I. Static analysis of graphical user interfaces of android applications. Master's thesis, Saarland University, 2016.

[49] Packs and phases in soot. https://github.com/Sable/soot/wiki/Packs-and-phases-in-Soot.

[50] Number of available applications in the google play store from december 2009 to june 2017. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[51] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical virtual method call resolution for Java. *j-SIGPLAN 35*, 10 (Oct. 2000), 264–280.

[52] An official web-page of susi project. https://github.com/secure-software-engineering/SuSi.

[53] Mobile malware evolution 2016: Cybercriminals continue their use of google play. https://securelist.com/analysis/kaspersky-security-bulletin/77681/mobile-malware-evolution-2016/.

[54] Vallee-Rai, R., and Hendren, L. J. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[55] Virusshare web-site. http://virusshare.com.

[56] T.j. watson libraries for analysis (wala). http://wala.sf.net/.

[57] 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* (New York, NY, USA, 2016), SOAP 2016, ACM, pp. 18–23.

[58] Windows mobile platform. https://www.microsoft.com/en-us/mobile/windows10/.

[59] Yang, S., Yan, D., and Rountev, A. Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the* (2013), IEEE, pp. 1–6.

[60] Yang, S., Yan, D., Wu, H., Wang, Y., and Rountev, A. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 89–99.

[61] Yang, W., Prasad, M. R., and Xie, T. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering* (Berlin, Heidelberg, 2013), FASE'13, Springer-Verlag, pp. 250–265.

[62] Yang, Z., and Yang, M. Leakminer: Detect information leakage on Android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering* (Washington, DC, USA, 2012), WCSE '12, IEEE Computer Society, pp. 101–104.

[63] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 95–109.