Static Flow Analysis for Hybrid and Native Android Applications

Claudio Rizzo

Submitted in fulfillment for the degree of
Doctor of Philosophy

Department of Computer Science
Royal Holloway, University of London

Declaration of Authorship

I, Claudio Rizzo, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Claudio Rizzo,
November 25, 2020

Acknowledgments

I wish to express my gratitude to my supervisors Johannes Kinder and Lorenzo Cavallaro. This thesis would not have been possible without your constant support and mentorship. I also want to thank you for always believing in my abilities and keeping me sane when tough situations presented themselves.

I am grateful for the REID scholarship from Royal Holloway and the financial support it provided during my Ph.D.

I wish to thank my lab mates, who supported me during this entire journey. I enjoyed all our fruitful scientific discussions and all the cheerful moments on and off-campus. In particular, I want to thank Duncan Mitchell and Emanuele Uliana for proofreading my thesis and always being there when I needed it. To Blake Loring and James Patrick-Evans, your help and support were crucial for the outcome of my work. To Roberto Jordaney, for a great time spent together in and out of the university.

I had some of the most exciting time playing CTFs with my team. I want to thank all of you for the challenges we solved together and for the time spent learning new technical skills. To Giovanni Cherubin, for coo-setting up the CTF team in the first place and the cheerful times we had inside and outside the campus. To Feargus Pendlebury, for all the binary exploits we wrote together and the time spent on research work. To Giulio De Pasquale for the good time spent together.

A very special thank you goes to you, Versha Prakash. Without you, writing this thesis would not have been possible. Your love and support kept me sane and gave me the strength to finish this journey. Thank you for the faith you had in me all the amazing time we spent together.

A big thank you for gratitude goes to my family: my mum Maria Grazia, my dad Raffaele and my sister Laura. This thesis and all my achievements would not have been possible without your constant support throughout the years. I am grateful for your faith in me and infinite support.

Abstract

Android applications consist of different components, interacting with each other, developed in different programming languages. While Java is at the core of an Android app, it may require to interact with the web or perform low-level Operating System (OS) operations. For example, Android Webviews are in-app browsers that expose interfaces to the JavaScript in the web page loaded to communicate with Java. Similarly, Android supports native code components that Java invokes via the *Java Native Interface* (JNI) framework. The ways of interaction of these components may introduce new security concerns the analyses need to address. Unfortunately, work so far has not addressed these mechanisms, compromising on precision and leaving potential security-critical bugs undiscovered.

In this thesis, we propose new techniques to enable existing analyses to consider the multi-language nature of an Android application. First, we focus on Android Webviews. To this end, we developed *BabelView*, a tool that uses information flow analysis to assess the security of Webviews. Our idea is that we can make reasoning about JavaScript semantics unnecessary by instrumenting the application with a model of possible attacker behavior – the BabelView. We evaluated our approach on a sample of 25,000 apps from the Google Play Store, finding 10,808 potential vulnerabilities in 4,997 apps, having over 3 billion installations worldwide. We manually validated BabelView on a sample of 50 apps and estimated our fully automated analysis achieves a precision of 81% at a recall of 89%.

Second, we focus on enabling analyses for Android native code. We created a new framework, *JniFuzzer*, which enables fuzzing for Android JNIs. We used JniFuzzer on real-world Android apps, finding potential vulnerabilities that we report as case studies. We then developed *TaintSaviour*, a Proof of Concept (PoC) tool which uses a black-box approach to generate summaries for JNIs. We implemented TaintSaviour as a plug-in of JniFuzzer, and we present a preliminary evaluation showing that our approach is viable and practical.

Contents

List of Figures

List of Tables

1 Introduction

Android is an operating system (OS) for mobile devices developed by Google. It allows users to run graphical applications, called apps. Apps are collections of different components interacting with each other. These components serve multiple scopes, from performing low-level OS operations to interacting with the web. Therefore, the programming languages used to build them vary accordingly. For example, consider Figure 1.1. The core of an Android app consists of Java code. However, native and web apps have segments written in C/C++ and HTML/JavaScript, respectively. Moreover, framework methods are stubs that the OS dynamically loads at runtime.

To assess the security of an app in this multi-language setting is hard. Static taint analysis, a data-flow analysis to track sensitive information flow of a program (detailed in Section 2.4), has been proven effective at evaluating the security of Android apps [1, 2, 3, 4, 5, 6]. For example, one can use taint analysis to detect privacy leaks, a threat that most of the Android malware poses [7, 8, 9]. Similarly, taint analysis can detect whether untrusted input reaches a sensitive method, threatening the security of of the app.

However, state-of-the-art taint analysis tools only focus on a single language (Java for Android apps). This is a problem because the analyzer has no visibility of large parts of the program. Security issues may depend on

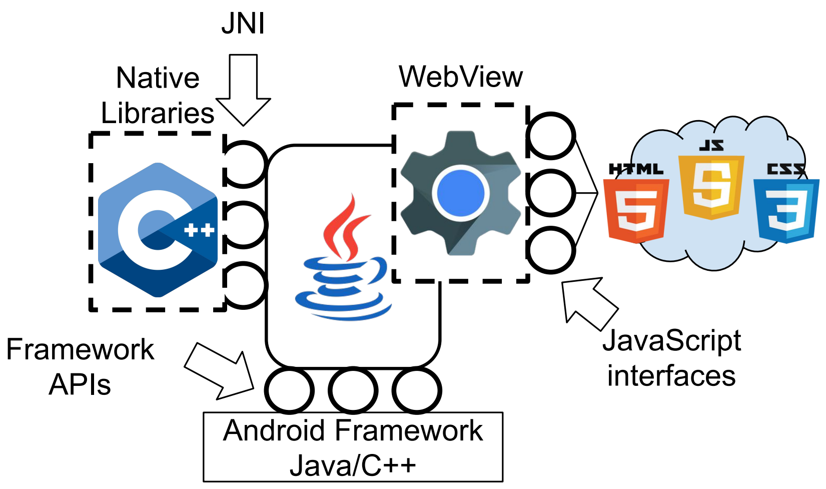


Figure 1.1: *APK as a multi-language app.* Android apps consist of several component written in different languages – e.g., C/C++ (JNIs) and JavaScript (Webviews).

how different language components interact. For example, the Android `WebView` class (detailed in Section 2.2) provides interfaces enabling an application to communicate with the web. Failing to consider them may leave potential security issues undiscovered. For instance, a banking app could provide access to account details when loading the bank's website in a Webview, or it could relay access to contacts to fill in payee details. Similarly, Android apps can use native code (detailed in Section 2.3) to execute performance-critical functionalities. Again, security issues can lurk in native code. For instance, memory corruption bugs can be introduced, leading to vulnerabilities such as buffer overflow. An analysis aware of these different language components (i.e., hybrid analysis) is then beneficial for exposing new security issues otherwise missed.

An hybrid analysis can also benefit malware detection, as malware authors persistently take advantage of the hybrid nature of Android applications, hiding their malicious behavior in those components ignore by analyses. For example, a study in 2017 showed a consistent increase in the use of native code as a means to include and hide malicious behaviors [10]. This trend has more recently been confirmed by another study, which highlights how nowadays applications pervasively use native code to evade analysis [11]. The authors showed how certain malware implements exploits that aim at obtaining root access on the device. The majority of these exploits are written in native code.

Native code is not the only vector. Android hybrid applications can hide malicious behavior in the web component [12, 13, 14, 15]. For example, apparently benign and legitimate apps may lead the user to click malicious links embedded into advertisements.

Security is not the only domain that would benefit from a hybrid analysis. Different approaches have been developed over the years to automatically test Android applications while maximizing code coverage and bug discovery [16, 17, 18, 19, 20]. These approaches rely on different techniques, including static and dynamic analysis and machine learning. However, none of them fully consider the hybrid nature of Android applications, and for example, they fail at targeting native code. Having an hybrid analysis would enhance the quality of the results, exposing new and undiscovered bugs.

In this thesis, we propose new approaches to enable hybrid taint analysis and to assess the security of Android apps. First, we investigate the security threats concerning Android Webviews. To this end, we developed *BabelView*, a tool that models the interaction between Java and HTML/JavaScript. BabelView enables existing taint analyses to consider the bridge across the two different languages.

Second, we consider a new approach to analyzing native code. In particular, we developed *JniFuzzer*, a tool capable of isolating and dynamically testing the security of native code interfaces. We successfully used

JniFuzzer to discover bugs with security implications (Section 5.3). We then built on top of JniFuzzer and developed *TaintSaviour*, a tool that uses a dynamic black-box approach to generate summaries for native methods. These summaries can then be reused by other taint analyses, enabling them to work across Java and C/C++.

Together, BabelView, JniFuzzer and TaintSaviour allow to expose potential bugs and vulnerabilities and possibly detect malicious behaviours in components that have traditionally been invisible to analysis.

1.1 Challenges for the Analysis of Hybrid Android Apps

In this section, we outline the most relevant challenges to our work. We divide them into three categories, namely:

1. Challenges for cross-language taint analysis (Section 1.1.1),
2. Challenges for Android Webview analysis (Section 1.1.2),
3. Challenges for Android native code analysis (Section 1.1.3).

1.1.1 Challenges for Cross-language Taint Analysis

Enabling Taint Analysis. Taint Analysis is effective at finding sensitive information leaks. It keeps tracks of sensitive data within a program, but tracking across languages is hard. There are two main strategies to work around this problem. In the first one, the analysis switches context and translates its current state from a language to another. In the second one, the analysis uses summaries of the method interfaced with the other language. Both cases are challenging for different reasons. A full context switch requires two different taint tracker engines, one per language.

When the taint reaches an interface method, the analysis must translate it for the new language and continue the analysis in the new context – e.g., passing from Java to JavaScript would require an engine that understand both languages' semantic. On the other hand, summaries generation needs to produce sound and meaningful summaries, which must be understood by the specific taint tracker. Moreover, the generation could, once more, require a taint analysis on the other language.

1.1.2 Challenges for the Analysis of Android Webviews

Android Webviews enable Android apps to render web pages. Analyses of apps with a Webview need to consider the mechanisms the app has to interact with HTML and JavaScript. In the following paragraphs, we present some of the challenges relevant to our work.

JavaScript. Webviews are full-fledged in-app web browsers, and as such, they render HTML and JavaScript to the end-user. An analysis aimed at assessing their security needs to consider JavaScript. JavaScript is a highly dynamic and asynchronous language and, therefore, is complex and expensive to analyze. For example, JavaScript's object model allows to create and delete object properties at runtime, increasing the analysis computation time and making it unpractical [21]. Additionally, most JavaScript applications rely on large libraries and frameworks, which in turn are written in a combination of JavaScript and native code [22]. In the context of Android Webview apps, JavaScript highly interacts with the Java counterpart. This interaction is crucial for thorough security analysis, but it comes with the challenge of finding solutions that integrate JavaScript analysis with Java. Indeed, this is not an easy task, as one needs to model the means by how the two languages interact and propagate information. Furthermore, a Webview may dynamically and remotely load JavaScript which therefore is not available for an analysis unless the app is running.

Threat Modeling. Webviews inherit all the threats that come from browsing the web. Moreover, the Java/JavaScript interaction mechanisms open up brand new scenarios of attacks (detailed in Section 2.2.4). For example, a Cross-Site Scripting (XSS) attack can be more powerful, as it can exploit the interfaces exposed by Webviews for the web. Therefore, performing an accurate threat modeling is hard, as we need to consider different factors specific to a certain app's layer.

Identifying Vulnerabilities. Taint analysis is not enough. There is a need to analyze the information flow results and deduce potential vulnerabilities out of them. This task is challenging as we may need to perform further inter-procedural analyses. For example, Android uses the same API to perform different functions (e.g., phone calls, email, calendar, etc.). The parameters provided dictate what task to perform. Therefore, these parameters need further investigation to understand what the API's functionality.

Multiple Webviews. Android apps can have multiple instances of Webviews. Therefore, the analysis of the custom hybrid interfaces must consider the Webview defining them. Maintaining a sound analysis, while keeping precision, is challenging. The analysis must at least consider different types of hierarchy of Webviews and associate them with the respective custom interfaces in order to be meaningful.

1.1.3 Challenges for the Analysis of Android Native Code

Android apps can use native code (C/C++) to perform computational intensive operation. A thorough analysis needs to consider the mechanisms apps have to invoke native code from Java and vice versa. In the following paragraphs, we outline some of the related challenges most relevant to our work on native code.

Underlining Native Framework. Android apps using native code do so via the Java Native Interface Framework (JNI, detailed in Section 2.3). This framework provides APIs to design interfaces acting as bridges between Java and native code. Therefore, a challenge when considering this sort of apps is to model how the underlying framework works. An analysis on such hybrid apps must be aware of how Java and Native code communicate, meaning we need to create abstraction on top of the two languages. Due to the differences between them, this is a challenging task. We will detail some of the challenges involved in the next paragraphs.

Methods Isolation. One way to assess the security of code is to execute the target program with random inputs to trigger unusual behaviors (i.e., fuzzing, detailed in Section 2.4). In Android apps, it is challenging to target the execution of a native interface. Approaches exploring the app's UI may never trigger native code, while directly executing the native method lacks the execution context.

Input Generation. When testing a program, a challenge is how to generate a meaningful input that exercises the program as much as possible. Moreover, native methods support a wide range of types, including complex structures to resemble Java Objects and Strings.

Lack of Source Code. Our goal is to analyze third party apps. Therefore, we cannot rely on source code. This deficit poses a serious challenge for the analysis of native code, as we enter the realm of binary analysis.

1.2 Goals and Overview

In this thesis, we propose new techniques for assessing the security of Android apps. The core of our work focuses on making existing analyses aware of the different programming languages present in a single app. We consider two categories: Android web apps using Webviews and apps using native code. To this end, we developed three different tools. In Figure 1.2, we list all the challenges discussed earlier (Section 1.1) and relate them to the respective tool. We cover each tool in a separate chapter of this thesis.

BabelView. As we discuss in Section 2.2, Android Webviews are powerful in-app browsers which bring about new security threats. For example, they allow developers to define custom *JavaScript interfaces* – i.e., bridges used via the loaded web content to use app and device specific functionalities. These bridges can poke holes into the browser sandbox, enlarging the attack surface in case malicious web content has loaded. With BabelView, our goal is to evaluate the impact of a possible attack against Webviews with respect to the JavaScript interfaces that they expose to the web. Differently from previous work, we do not flag all interfaces as dangerous, instead we rely on static analysis to understand their nature and provide meaningful feedback focusing on the most dangerous cases. To analyze these interfaces is challenging as they are only ever accessed by the JavaScript part of the application. Our key idea is that we can and should avoid reasoning about JavaScript, as it is hard to analyze. To this end, we model a general attack behavior that over-approximates the possible information flow semantics of an attack and embeds its logic in a specially crafted Webview, the BabelView. We then instrument the BabelView into the target app, replacing its Webviews and its descendants with BabelView, which simulates an arbitrary execution of the JavaScript inter-

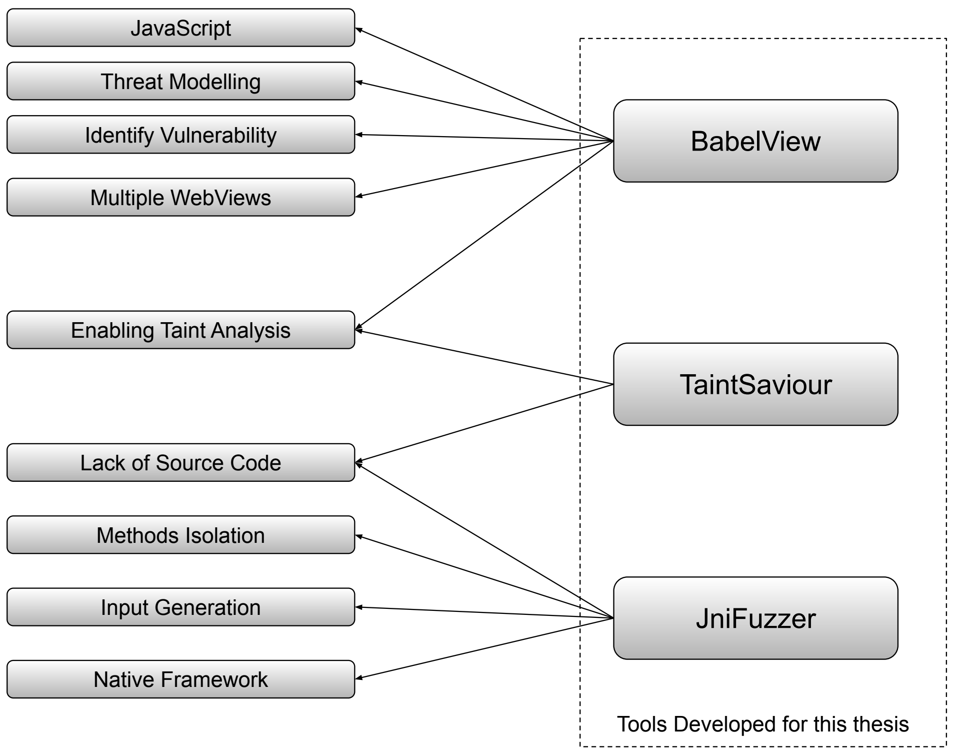


Figure 1.2: *Challenges and solutions.* Challenges for analysis of Android multi-language apps and proposed solutions.

faces. Because an app can have many Webviews, we generate a BabelView for each different type of Webview and bind all JavaScript interfaces, respectively. Subsequent taint analyses can now reason on the JavaScript interface detecting information flows in their context. BabelView further refines the taint analysis results projecting them to a potential vulnerability and its impact. We support our approach by a large scale experimental evaluation, presented in Chapter 4, which sheds light on the state of Webviews security in Android.

JniFuzzer. Static analysis of binary code is hard, so unsurprisingly there is no usable framework for analyzing Android native code. Our goal is to ease the effort to analyze native code and provide a tool that can expose security bugs lurking in native code. To this end, in Chapter 5, we present JniFuzzer, a native code analysis framework, based on fuzzing, that breaks the status-quo of Android apps' native code analysis. JniFuzzer effectively models the Android native framework and isolates and executes each of the native interfaces inside an Android app. JniFuzzer does not require the app's source code, and it can be easily extended as a plugin system to support different analyses. Moreover, we designed JniFuzzer to support state-of-art fuzzing tools in its analysis, such as AFL [23]. As we will show in Chapter 5, we successfully used JniFuzzer to discover real bugs in Android apps.

TaintSaviour. Android loads certain framework code at run time. Moreover, frameworks such as JNI invoke interface implemented in native code, posing a major challenge to static taint analysis. As a result, native code is usually excluded by the these analyses, preventing the discovery of possible bugs or malicious behaviors. In this thesis, our goal is to provide viable solutions to enable taint analysis to reason about native code. In Chapter 6, we present TaintSaviour, a tool aimed at solving these kind of problems.

TaintSaviour uses a black-box approach to generate summaries of methods, which we can reuse to instruct taint analysis on how to proceed when encountering one of the summarized methods. We built TaintSaviour as a plugin for JniFuzzer and used it to generate summaries for JNI methods. We support the validity of our approach with a preliminary experimental evaluation, showing that TaintSaviour can be practically used.

1.3 Thesis Contribution

As a summary, this thesis makes the following contributions to the state of the art:

- We provide a solution to enable static taint analysis to include JavaScript interfaces in Webviews in the analysis. We implemented this solution in BabelView, a tool assessing the security of Android Webviews, which is available as open source[1] (Chapter 3).
- We analyzed 25,000 applications from the Google Play Store to evaluate our approach and surveyed the current state of Webview security in Android. Our analysis reports 10,808 potential vulnerabilities in 4,997 apps, which together are reported to have more than 3 billion installations. We validated the results on a random sample of 50 applications and estimate the precision to be 81% with a recall of 89%, confirming the practical viability of our approach (Chapter 4).
- We implemented JniFuzzer, the first Android fuzzing framework to directly target applications' native libraries and JNIs and we make it available open source[2]. The JniFuzzer framework supports plugins, enabling analysts to implement custom analyses. Finally, we

[1]`https://github.com/ClaudioRizzo/BabelView`

[2]`https://github.com/ClaudioRizzo/JniFuzzerFramework`

present an initial exploratory evaluation, demonstrating how this framework can be used to detect real-world bugs and vulnerabilities in Android apps (Chapter 5).

- We introduce a new black-box approach to generate method summaries. These summaries can then be reused to improve the completeness of current static taint analysis tools. We implement this approach in TaintSaviour, a Proof of Concept (PoC) tool that we used for our preliminary experimental evaluation, showing that our approach is viable in practice (Chapter 6).

Chapter 3 and Chapter 4 of this thesis have been published as [24] (BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In: Bailey M., Holz T., Stamatogiannakis M., Ioannidis S. (eds) Research in Attacks, Intrusions, and Defenses. RAID 2018. Lecture Notes in Computer Science, vol 11050. Springer, Cham).

Background 2

This chapter provides the necessary information to understand the remainder of the thesis. First, in Section 2.1, we provide an introduction of Android. In Section 2.2, we explore Android Webviews. In Section 2.3, we then detail how native code is used in Android. In Section 2.4, we provide a general description of program analysis techniques relevant to the thesis. Finally, in Section 2.5, we discuss static analysis in the the Android scenario, discussing existing tools relevant to this thesis.

2.1 Android Framework

In this section, we provide a brief introduction to Android. In Section 2.1.1, we provide an overview of the Android Operative System. In Section 2.1.2, we describe the format of an Android app and finally, in Section 2.1.3, we describe its main software components.

2.1.1 Android Overview

Android is an open-source OS based on the Linux kernel and developed to work for several devices, and, in particular, on mobile phones. As we show in Figure 2.1, the Android stack consists of multiple layers.

At the bottom, we have the Linux kernel, which is the foundation of the

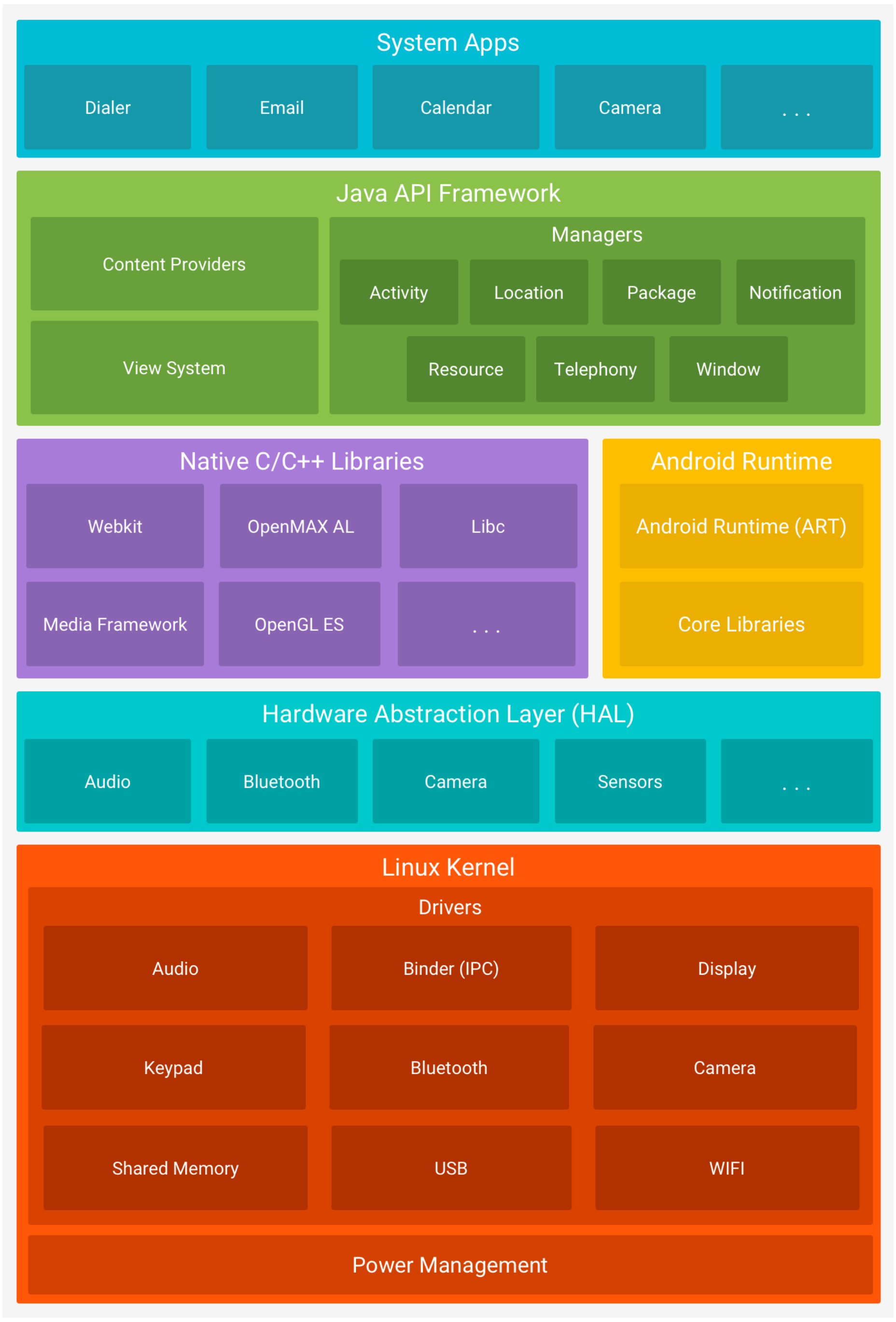


Image from `https://developer.android.com/guide/platform`

Figure 2.1: Android OS software stack.

OS. The Linux kernel is very well known and therefore device manufacturers can easily integrate their hardware with Android.

On top of the kernel, we find the Hardware Abstraction Layer (HAL). This layer provides interfaces that expose hardware capabilities to higher levels. The HAL consists of different modules, which implement an interface for a specific hardware component (e.g., camera, Bluetooth, etc.). Upon a framework API call to access the specific hardware, Android loads the library module for that component.

The next layer consists of Android Runtime (ART) and Native Code libraries. ART is the Android runtime environment since version 5.0 (API level 21). Android applications are compiled into DEX, a bytecode format designed for Android that is optimized for minimal memory footprint. The ART environment converts the DEX bytecode into machine code, which then executes. Before Android version 5.0, Dalvik was the Android runtime. One of the main improvements of ART over Dalvik is ART's support for ahead-of-time compilation, which improves execution performances.

The native code libraries sit next to ART. Many core Android systems components build on native code and therefore require C/C++ libraries. Moreover, Android provides Java framework APIs to expose the functionality of some of these native libraries to apps.

On top of ART and native libraries, we find the Java API Framework. The entire feature set of the Android OS is available through Java API. For example, all Android apps in the app layer are developed using this set of APIs.

2.1.2 Android App Packaging

Android apps come in the Android App Packaging (APK) format. You can think of an APK as a ZIP file containing all the relevant resources an app

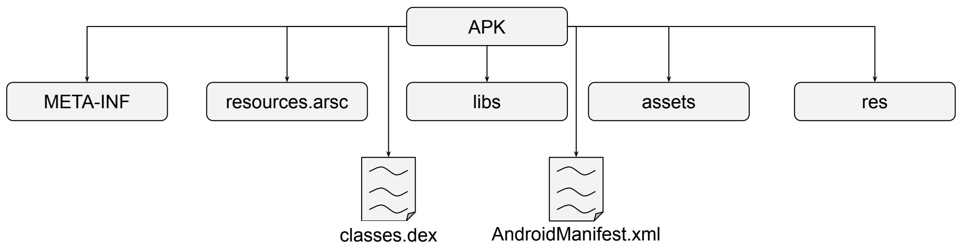


Figure 2.2: *APK structure.* This figure shows the main components inside an APK.

requires to run on a device. In Figure 2.2, we show the APK's structure.

Android Manifest. The Android manifest file presents essential information about the application that the Android system must have beforehand to run the app. For example, the manifest contains the app package name, which serves as the application's unique identifier. It describes the app's components (e.g., Activities, services, etc.) and specifies the app's permissions. For a more detailed explanation of the AndroidManifest, we refer to the official documentation [25].

Lib Folder. The `lib` folder contains any native libraries that developers used for their apps. Each library serves different Industry Standard Architectures (ISA), and, therefore, the `lib` directory contains a sub-folder for each different architecture supported. For example, a library compiled for `x86_64` is stored in the `libs/x86_64` folder. This folder is of particular interest for us because, in these native libraries, we find the implementation of the JNI methods (see Section 2.3 and Chapter 5).

Assets Folder. The assets folder contains the application assets. Consider, for example, mobile web application. These apps combine web technology with the Java API Framework of Android. They can load and render HTML/JavaScript files, which they can find within the assets directory. We will further discuss mobile web applications, as we analyze Android Webviews (see Section 2.2 and Chapter 3).

Res Folder. The `res` folder contains the app's resources. These resources vary from the app's layouts (i.e., UI of part of the application) to images and strings. The compiled version of these files is located in `resources.arsc`.

DEX File. The DEX file is where the byte code of the application lies. Android Runtime uses this file to run the app.

2.1.3 Android Mobile Applications

Mobile applications differ from desktop counterparts in that they have multiple entry points. For example, if one opens an email app from the home screen, a list of emails – i.e., the email app's main view – will show. However, if one starts the email app, say, from a social media app, the entry point will be a view for typing an email.

The Android Activity is designed to support this paradigm. One can think of an Activity as a specific screen displayed to the user. Activities define a layout, determining where UI components appear on the screen. Every Activity must specify a layout to display content. Each layout can be specified as an XML file in the `res` folder, or programmatically in the Activity code.

To make an Activity more modular, Android provides fragments. A fragment is a modular section of an Activity and has its own life cycle. For example, multiple fragments can be combined in a single activity to build a multi-pane UI. Moreover, fragments can be reused in different Activities.

Whether an application uses only Activities or Activities with fragments, it ultimately displays views. A view is a UI basic component and is responsible for drawing the UI and handling events. The Java API framework of the Android OS provides several different views to use. For example, `WebView`, which we will further discuss in Section 2.2, is a subclass of `View` that acts like a browser. Within the same activity of fragment, you can have multiples views.

An Android mobile app consists of a set of Activities and fragments which act as containers for view components. Activity, fragments, and views, all sit at the Java API framework layer and therefore interact with the OS via the Java API. However, certain APIs are interfaces to execute functionality outside the Java API framework layer. For example, this is the case for Webviews (Section 2.2) and JNIs (Section 2.3).

2.2 Android Webviews

Webviews are customized in-app browsers. Their powerful mechanisms for interaction between the application Java code and the rendered web content make them valuable resources to develop portable and performant mobile applications. In the Android ecosystem, `WebView` is a subclass of `View` (i.e., the basic block for a UI component) that can be used to display web pages as part of an Android `Activity` layout [26].

In this section, we provide an in-depth overview of how Webviews can be implemented in an Android application. In Section 2.2.1, we describe how `WebView` can be defined and inflated[1] into the application view. In Section 2.2.2, we provide an overview on how JavaScript can be enabled and used within `WebView`, describing the security impact it

[1]In the Android ecosystem, to *inflate* means to insert a specific component, e.g., a Webview, into a layout.

```
<android.support.constraint.ConstraintLayout
    .../>
    <WebView
        android:id="@+id/webview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</android.support.constraint.ConstraintLayout>
```

Listing 2.2.1: Declaring WebView in XML

brings about. In Section 2.2.3, we discuss the Same Origin Policy (SOP) with respect to Webviews and, finally, in Section 2.2.4, we describe some of the attacks that can occur on `WebView`.

2.2.1 Define a Webview

As any other `View`, `WebView` can be added dynamically in an application activity, or statically as part of the activity's `XML` layout file.

Adding a `WebView` dynamically happens directly in the `Activity` code as part of the `onCreate` life cycle method. A `WebView` is first instantiated and then added to the activity view. The following is an example of how this can be achieved:

```
...
WebView mWebView = new WebView(this);
setContentView(mWebView);
...
```

The static alternative requires the activity layout `XLM` file to include a `WebView`. Consider as an example the `XML` in Listing 2.2.1, which defines the main activity layout. Here, `WebView` is declared as a child of its container `ConstraintLayout`. Since `WebView` is a subclass of the more general class `AbsoluteLayout` (which in turn extends `ViewGroup` and `View`), it inherits all the style attributes these classes have. They can be used by adding them as `XML` attributes. Notice that an ID needs to be specified to later look up the `WebView`. Once this process of declaration and styling is completed, the resulting layout needs to be inflated and the Webview retrieved. Again, this happens in the `onCreate` method by setting the view content:

```
...
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
WebView mWebView = (WebView) findViewById(R.id.webview);
...
```

Differently from the dynamic case, here the `WebView` constructor is selected at run time by the Android framework depending on the layout and style attributes that have been defined [27, 28].

Custom Webviews are also supported. Developers who wish to create custom Webviews can do so by declaring a class inheriting from `WebView`. This gives developers more flexibility as they can override various callbacks as required. Once declared, the custom Webview can be inflated with one of the two methods described above. However, extra care needs to be taken when using the static approach. Developers need to include the proper constructors that will be used by the layout inflater to render and instantiate the Webview. For more detail, we refer the reader to a comprehensive tutorial on the Android Developers documentation [29].

Once the `WebView` has been instantiated, web pages can be loaded with the `loadUrl` API.

2.2.2 Enabling JavaScript

When a `WebView` object is created, it obtains a default set of settings. All these settings are wrapped in the `WebViewSettings` class and can be obtained by invoking `getSettings` method of `WebView`.

By default, only plain `HTML` can be rendered and no JavaScript is allowed. However, JavaScript is required for any non-trivial application.

```
...
WebViewSettings mSettings = mWebView.getSettings();
mSettings.setJavaScriptEnabled(true);
...
```

In the hybrid application context, JavaScript is not only used for common web tasks, it is key for interaction between the application native code (i.e., Java) and the web content in both directions.

Java code can execute arbitrary JavaScript code in the Webview by passing a URL with the "`javascript:`" pseudo-protocol to the `loadUrl` method of a Webview instance. Any code passed in this way is executed in the context of the current page, as if it were typed into a stand-alone browser's address bar. For the other direction, to let the JavaScript code in the Webview call Java methods, the Webview allows developers to create custom interfaces. Any methods of an object (the *interface object*) tagged with the `@JavascriptInterface` annotation[2], passed to the `addJavascriptInterface` API of `WebView`, are exported to the global JavaScript name-space in the Webview. Developers make wide use of interface methods in different ways. For instance, consider Listing 2.2.2; in Line 1 and Line 2, the interface object `JSUtils` is instantiated and then added to `mWebView`. In this interface object, two interface methods, `getLocation` and `notify` are defined at Line 12 and Line 17. The first method can be used by JavaScript loaded in `mWebView` to query the location of the phone. this is a common use case: web pages cannot directly access the phone's internal resources. The second method is instead used to notify `mWebView` with a certain message: developers often implement callbacks mechanisms due to the asyn-

[2]The `@JavascriptInterface` annotation was introduced in API level 17 to address a security vulnerability that allowed attackers to execute arbitrary code via the Java reflection API [30].

```java
JSUtils jsUtils = new JSUtils(mWebView);
mWebView.addJavascriptInterface(jsUtils);

public class JSUtils {
  private WebView webView;

  public JSUtils(WebView webView){
    this.webView = webView;
  }

  @JavascriptInterface
  public String getLocation(){
    return getLocationAsString();
  }

  @JavascriptInterface
  public void notify(String message){
    pushToWebView(message);
  }
}
```

Listing 2.2.2: JSUtils Interface Object

chronous nature of JavaScript. These callbacks are passed as arguments for interface methods such as `notify`, where `pushToWebView` at Line 18 make use of `loadUrl` to run JavaScript in `mWebView`.

Enabling JavaScript exposes the Webview to `XSS` and JavaScript injections via Man In The Middle (MITM). As we shall see in Section 2.2.4, the impact of such attacks is wider for hybrid applications than it is for a common web app.

2.2.3 WebView and Same Origin Policy

Pages loaded in a Webview must follow the Same Origin Policy [31]. Same Origin Policy is a mechanism that restricts Javascript running within a page to access resources outside this page origin. The origin of a page is defined as the triplet composed of the scheme, the host and the port of its URL. Consider for example a page loaded from `http://example.com`. If Javascript in this page tries to load content from `https:///example.com`, the SOP will stop this from happening as the schemes don't match and, therefore, the origin is different. Unfortunately, SOP is not always enough to ensure user security. For example, when a Webview loads a page from the file system, its Javascript has virtually access to the whole file system, as it will be from the same origin. This behavior might expose the user to different attacks if the application is not properly designed.

Two different Webviews cannot directly interact with each other and one cannot execute JavaScript within the other. You can think of them as two different tabs in a web browser. However, by poorly designed JavaScript interfaces, this behavior can be altered, resulting in dangerous attacks that can bypass SOP.

In Section 2.2.4, we will discuss the details of some of these attacks.

2.2.4 Attacks on Android Webviews

Webviews introduce new security threats, which change the security landscape of mobile web browsers. The interface methods (see Section 2.2.2) poke holes into the browser sandbox and they can result in Same Origin Policy (SOP) violations. As a result, there is a wide range of possible attacks that can be carried on Webviews [13, 32, 33, 34, 35, 36, 37, 38, 39].

Abusing JavaScript Interfaces. The Webview's JavaScript interface mechanism enforces a policy specifying which Java methods are available from the JavaScript context. Developers of hybrid apps are left to decide which functionality to expose in an interface that is more security-critical than it appears. As an example, recall the `JSUtils` interface object we saw in Listing 2.2.2. Its `getLocation` method allows JavaScript running within `mWebView` to query the phone's current location[3]. This seems reasonable if one assumes that the application is self-contained and the web content is trusted. Unfortunately, this is not always the case: for example, an attacker could forge a `URL` and trick users to load it in the Webview, stealing their location.

Another vector of injection is through `iframes`, which are embedded containers running an external web application. For example, ads often use `iframes` to display their content. If the Webview has an interface object available, then any child of the loaded page can access its interface methods, including JavaScript running within the iframe. In our example scenario, a malicious advertisement would be able to read the user location.

[3]We assume that `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` are granted

```
@JavascriptInterface
public void cacheData(String key, String data){
  doStore(key, data);
}

@JavascriptInterface
public String getCacheData(String key){
  return getData(key);
}
```

Listing 2.2.3: JSUtils Interface Object - Extension

Attacks from Frame Confusion. A more convoluted attack scenario involving `iframes` is known as *frame confusion* [13]. As we mentioned in Section 2.2.2, one common use case for interface methods is to function as callbacks. This is exactly the purpose of `notify` in `JSUtils`. Suppose a page is composed of a web page (main frame), which contains some `iframes` (child frames). If a child frame invokes `notify`, we would expect it to run in the child frame context. Instead, the JavaScript runs in the main frame, introducing a channel among the frames. For instance, `notify` pushes JavaScript coming from the Webview back into the Webview. If this is not resilient to JavaScript injection, frame confusion can occur. Malicious code from child frames can be used in this manner to control the main frame, in the manner of `XSS`.

Same Origin Policy Violations. Consider now Listing 2.2.3, which extends `JSUtils` with `cacheData` and `getCacheData`, two (generic) methods that the web page can use to cache data for improved performance. The issue here is that once the data is out of the Webview, it is no longer protected by the browser sandbox SOP. Any page inside the Webview can use those interface methods to read and modify that data, bypassing SOP.

TLS Errors. The Android Software Development Kit (SDK) provides mechanisms that developers can use to protect Webviews. Webviews fully support TLS and therefore HTTPS. If an error happens during the TLS handshake, the Webview drops the connection, always treating the error as fatal (even non-fatal ones, e.g., expired certificates). However, developers can customize this behavior and easily introduce mistakes. A common one is to bypass certificate validation by blindly accepting all possible certificates. In other cases, developers do accept invalid certificates (e.g., a certificate valid for `example.com` only, while connecting to `google.com`).

Before Android 9.0 was released in August 2018, HTTPS was not enforce by default, opening to mixed content vulnerabilities (i.e., a mix of HTTP and HTTPS traffic) and SLL-stripping attacks. From Android 9.0 HTTPS is enforced [40]. While this limits the scope for errors, the problem is still there. Developers often wekean their applications security by copy pasting vulenerable network policies from the Web or adopting libraries forcing them to downgrade their security [41].

Origin Validation Threats. Another threat comes from lack of origin validation on the resources loaded in a Webview. For example, consider a situation where an XSS triggers an HTTP request to, say, `http:`*`//evil.com`*. If the Webview does not specify a whitelist of safe origins for the downloaded content, the malicious website is loaded. Android Webviews provides two APIs that can mitigate that scenario: `houldInterceptRequest` and `shouldOverrideUrlLoading`. The first API allows developers to control each resource accessed by the web page, while the second one only allows for URLs validation.

Lack of Input Validation. Webviews can access local resources with some restrictions, which vary depending on the choice of API. Developers can change Webviews default permissions by means of certain APIs. The main ones are as follows:

- *setAllowContentAccess* True by default, it allows the Webview to access content providers.
- *setAllowFileAccess* True by default, it allows the Webview to load local files from the file system using the `file:///` schema.
- *setAllowFileAccessFromFileURLs* False by default since API 16 (True for older APIs). This setting allows JavaScript from local HTML pages (loaded with `file:///`) to access resources in the file system.
- *setAllowUniversalAccessFromFileURLs* False by default since API 16 (True for older API). This setting allows local JavaScript to access resources from any origin.

Granting Webviews unrestricted access to resources is not necessarily a problem. However, care must be taken to avoid private data leaks. Consider the following code as an example:

```
mWebView.loadUrl("file:///android_assets/www/users/"+
    getUserName());
```

Here, the username is user-controlled and therefore an untrusted user can manipulate it. For example, if the user name is a string like `"../../storage/emulated/0/Pictures/pic000.jpg"`, then the malicious user would be able to load and possibly exfiltrate the image `pic000.jpg`. This scenario is possible because of the lack of user input validation.

Conclusion. It is easy for a developer to erroneously assume the JavaScript interface to be a trusted internal interface shared only between the Java and JavaScript portions of the same app. In reality, it is more akin to a public API, considering the relative ease with which malicious JavaScript code can make its way into a Webview (Section 3.1.2). Therefore, care must be taken to restrict the interface as much as possible and to secure the delivery of web content into the Webview. In Chapter 3, we provide a way for developers and app store maintainers to detect applications with insecure interfaces susceptible to abuse; our study in Chapter 4 confirms that this is a widespread phenomenon.

2.3 Android Native Components

Android provides the Native Development Kit (NDK) [42], a set of tools that allow developers to write C and C++ code in their applications, providing platform libraries that can be used to access the physical layers of the device. We call code written with the NDK *native code*. Native code has many uses, for example, performance requirements for computationally intense applications (e.g., graphics rendering or physics simulation) or design requirements for integrating bundled third-party code (e.g., advertising).

In this section, we provide an overview of how developers can include and use native code into their applications. In Section 2.3.1, we introduce the JNI framework. In Section 2.3.2, we discuss how variables are accessed and referenced between Java and native code. In Section 2.3.3, we detail how strings and arrays are managed. Finally, in Section 2.3.4, we describe how the JNI framework deals with Java exceptions.

2.3.1 Java Native Interface: Overview

The Java Native Interface [43] acts as a bridge between code that runs in a Java Virtual Machine (JVM) and compiled code written in another programming language, such as C or C++. Importantly, the JNI does not impose any restriction on the implementation of the underlying JVM. This enables JVM vendors to add support for the JNI with support for the JNI

```
1   package example;
2
3   public class MyActivity extends Activity {
4    ...
5    public native int nativeM(int);
6    static {
7     System.loadLibrary("libnative.so")
8    }
9    public int getSum(int a, int b) {return a+b;}
10   ...
11  }
```

Listing 2.3.1: Loading Native Methods

without affecting any other part of their JVM. Android itself comes with its implementation of the JNI, which is part of the ART [44].

JNI functions are available to native code, and are used to access JVM features and are available through an *interface pointer*, a pointer to an array of function pointers. This organization is similar to a C++ virtual function table and comes with the benefit that the JNI name-space is separate from the native code. Each native method receives a pointer to the JNI interface pointer as an argument. The JVM passes the same pointer to native methods within the same Java thread.

Native methods are packaged as part of a shared library which must be loaded by the application. Consider Listing 2.3.1; `System.loadLibrary` (Line 7) is used to load a platform-specific native library, where the native method `nativeM` is defined. Developers can choose whether all native methods are contained in a single library; the JVM keeps an internal reference of all loaded libraries for this situation.

Native methods are resolved by the dynamic linker, or they can be registered by the programmer by calling the JNI function `RegisterNatives`. The dynamic linker resolves native methods based on their names. For

```
1  static JNINativeMethod methods[] = {
2     {"getMessage", "()Ljava/lang/String", (void*)
         NativeMessage},
3  };
4
5  JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved) {
6     JNIEnv* env = ...;
7
8     jclass my_clazz = env->FindClass("example.MyActivity");
9     env->RegisterNatives(myclazz, methods, 1);
10    ...
11 }
```

Listing 2.3.2: Statically Linked Native Methods

example, `nativeM` is mapped to `Java_example_MyActivity_nativeM`. It consists of a concatenation of the prefix `Java_`, a fully qualified class name, and a method name.

This approach is simple, but leads to verbose native method names and fails if the underlying operating system does not support dynamic linking (not the case for Android). To overcome this, developers can statically link each Java native method to a chosen function pointer via `RegisterNatives`. Consider Listing 2.3.2, which implements the `JNI_OnLoad` callback, which is triggered when the native library is being loaded. In this callback, developers can use `RegisterNatives` (Line 9) to register an array of `JNINativeMethod`, whose signature is found in `myclazz`(Line 1). Each `JNINativeMethod` is composed of the Java name of the interface, parameters and return types, and a function pointer to the native implementation. The function pointer must point to a function with a compatible signature.

Each native method has the JNI interface pointer as the first parameter, which is of type `JNIEnv`. The second argument depends on whether the native method is static or not. If static, the argument is a reference to

Java Type	Native Type	Size
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bit
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	N/A

Table 2.1: *Primitive Types Mapping.* Java types mapped to native code types.

its Java class. Otherwise, it is a reference to the instantiated object. The remaining parameters correspond to the regular Java method arguments. The result of a native method call is passed back to the caller via its returned value.

To enable this mechanism, Java types need to be matched to their respective native types. For primitive types, the JNI framework redefines all native code primitive types as `j<primitive_type>` (see Table 2.1). Several reference types are also included by JNI. They correspond to all different kinds of Java objects and have a similar hierarchical organization. This hierarchy and the mapping to Java types can be seen in Figure 2.3. `jobject` is the most general type and is mapped to any Java object – i.e., `java.lang.Object`. `jstring`, `jclass`, `jarray` and `jthrowable` are all subclasses of `jobject`; these map to the obvious Java types. Moreover, the arrays are further extended to support specific types, including arrays of `Object` and arrays of primitive types.

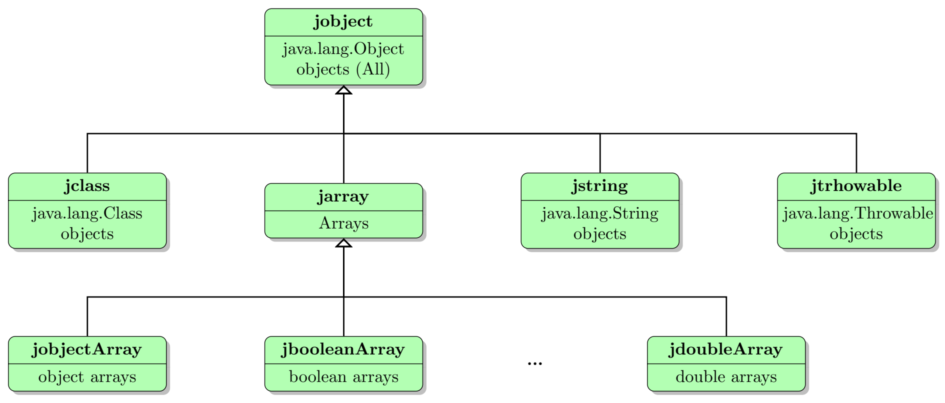


Figure 2.3: *JNI types hierarchy.* This figure show the JNI type system which maps Java type to C/C++ types. In each block, we show the native type at the top and the Java respective at the bottom.

2.3.2 Referencing And Accessing Java Objects

While primitive types (e.g., integer, chars, etc.) are copied between Java and native code, complex Java objects are passed by reference. The JVM keeps track of all the objects passed to native code and prevents them from being garbage collected. Similarly, native code has ways to inform the JVM that an object is no longer needed (e.g., with `DeleteLocalRef`). There are two types of references used by native code: local and global. Local references to an object are valid for the duration of the native method, and will be garbage collected once the method returns. In contrast, global references are valid until explicitly freed.

All objects passed to native methods are local references, as well as all Java object returned by JNI functions. Local references are stored in registries by the JVM, which maps them to their respective Java object, preventing the object from being garbage collected. The JVM creates a registry for each transition of control from Java to native code and all the object passed to the native method are added to it.

```
...
jclass my_clazz = env->GetObjectClass(obj);
jmethodID getSumID = env->GetMethodID(my_clazz, "getSum", "
    (I)II");
jint result = env->CallIntMethod(obj, getSumID, 42, 10);
...
```

Listing 2.3.3: Statically Linked Native Methods

The JNI provides a set of API functions to access global and local references. This ensures that the implementation of native methods is not affected by the underlining JVM. These functions can be used by native code for both accessing fields and invoking methods of Java objects. For example, to invoke the method `getSum` in Listing 2.3.1 from a native method, this can be accomplished by code similar to that provided in Listing 2.3.3. Here, `obj` is a reference to an instance of `MyActivity`. It is used to obtain its class definition, which is then used to lookup a `jmethodID`. This ID and `obj` are finally used to invoke `getSum` and obtain the result. This may differ depending on whether the method is a static or an instance method. In the former case, there would not be any need for an instance reference. In the case of instance methods, `obj` can be obtained in different ways depending on the situation. A full description of JNI functions is found in the documentation [45].

2.3.3 Array and String Management

Accessing objects through opaque references via JNI functions brings about higher overheads compared to directly accessing C/C++ data structures. While this is usually tolerable for general common objects, the overhead is unacceptable for large objects such as strings and arrays. For example, calling a function to access each element of an array in a for loop, is inefficient. The JNI framework solves this problem with *pinning*, i.e., enabling native methods to ask the JVM to pin the content of the array to directly address it.

The drawback of this approach is that the garbage collector must support pinning and that the JVM must layout arrays contiguously in memory. The JNI framework adopts a compromise that helps to overcome both these problems. First, a set of functions is provided to allow copying between a Java array and a memory buffer. Second, functions are given for retrieving a pinned version of the array element. However, it depends on the JVM implementation whether these functions copy the content of the array. If the garbage collector supports pinning and the layout of the array is as expected by the native method, then no copy operation is performed. Otherwise, the array is copied to the heap and a pointer to it, is returned. Finally, the JNI interface provides functions to inform the JVM that the array elements are no longer needed by the native method.

As arrays of characters, strings are dealt with pinning. However, strings are encoded in particular formats. The JNI specification uses modified `UTF-8` [46] strings, which are the same as those used by the JVM. Modified `UTF-8` strings are encoded so that they never contain null bytes. However, all Unicode characters can be represented, including `U+0000`, allowing such strings to be processed by traditionally null-terminated string functions. Notice that certain characters are encoded using two or more bytes. For example, `U+0000` is encoded as `\xc0\x80`.

The JNI interface provides many API functions to manage strings [45]. For example, to create a new Java string, one can use `NewStringUTF` or `NewString` functions. The difference is that the first one creates a string from a modified `UTF-8` characters buffer. Instead, the second creates a new string starting from a Unicode characters buffer. The two orthogonal functions are `GetStringUTFChars` and `GetStringChars`. They can be used to obtain a buffer of characters from a Java string. There are more APIs that one can use. All of them have a version for Unicode and modified `UTF-8` characters.

2.3.4 Exception Management

The JNI framework allows for the handling and generation of custom exceptions [47]. JNI functions report errors in different ways. The most common is to return an error code and throw a Java exception. In this scenario, one needs to first check for the error code and then call `ExceptionOccurred` to obtain the exception object containing details of the error. However, there are cases where it is not possible to check for an error code. For example, when the JNI function returns a result from a Java method invocation or a JNI function that throws `ArrayIndexOutOfBoundsException` or `ArrayStoreException`. Developers must call `ExceptionOccurred`, `ExceptionCheck` and similar APIs to check for possible exceptions that occurred during the execution of the Java method.

To handle an exception, developers have two possibilities. First, they can decide to handle the exception directly in native code. In this case, they must check for the exception, clear it (e.g., using `ExceptionClear`) and then handle it. Alternatively, one can propagate the exception to Java code and handle it there. To do so, the native code needs to return after an exception has occurred. The JVM then propagates the unhandled exception back to the Java method that invoked the native method causing the error.

2.4 Program Analysis Techniques

In this section, we provide an overview of program analysis techniques most relevant to this thesis. Program analysis is the science of automatically analyzing a computer program to determine properties of that program (e.g., correctness, robustness or safety). We begin with Section 2.4.1,

```
1 String path = getHttpVariable("path"); // path is tainted
2 String url0 = "http://example.com/0/"+path; // path -> url0

3 string url1 = "http://example.com/1/"+path; // path -> url1

4 url1 = "http://example.com/1/default" // url1 loses taint
5 loadUrl(url0); // information flow detected
6 loadUrl(url1); // no information flow
```

Listing 2.4.1: Taint Analysis Example.

where we discuss taint analysis, a static analysis technique. In Section 2.4.2, we describe the technique of fuzzing and finally, in Section 2.4.3 we discuss metrics to evaluate the performance of an analysis.

2.4.1 Information Flow Analysis: Taint Analysis

Taint analysis is a data-flow analysis, which tracks sensitive information flows within a program. For this thesis, we only consider inter-procedural taint analysis. We define an information flow as the transfer of information gathered from a source to a sink method. Source and sinks are defined by the analyst and they usually are chosen as sensitive APIs that retrieve and use sensitive data, respectively. For example, consider the code in Listing 2.4.1. This code uses `getHttpVariable` to get the value of an HTTP variable, which is user controlled (Line 1), and assign it to `path`. Two URLs, based on `path`, are then created (Line 2 and Line 3) and loaded via `loadUrl`. If an analyst were interested in whether a user can control loaded URLs, `loadUrl` would be a source and `getHttpVariable` a sink. In this scenario, a taint analysis would report a tainted (i.e., sensitive) information flow from the source `getHttpVariable` (Line 1) to the sink `loadUrl` (Line 5). The user controlled variable, `path`, is tainted and used to create `url0`, which is then used to load a URL. A sensitive infor-

```
String x = "foo";
sink(x);
x = source();
```

Listing 2.4.2: Flow Sensitivity Example

mation is therefore detected. In contrast, no information flow involving `url1` is found, as it `url1` is assigned a constant value, independent from the user input.

Taint analyses rely on point-to analyses [48] to determine information on the value of pointer variables. This analysis can have different and custom levels of sensitivities, which determine the analysis precision. For an object-oriented language, and relevant to this thesis, the most common are flow, context, object, and field-sensitive.

Flow-sensitivity. A flow-sensitive analysis considers the order of execution of the program statements. Consider, for example, the code in Listing 2.4.2. A flow-sensitive analysis keeps full precision of the snippet, as it understands that the call to `sink` happens before the one to `source`. On the other hand, a flow-insensitive analysis would, erroneously, report an information flow.

Context-sensitivity. One challenge for inter-procedural analysis is that the behavior of each procedure depends on the context in which it is called. For example, consider Listing 2.4.3. The procedure `convert` is invoked at two different call sites: Line 7 and Line 8. The string `s1` is first tainted, in Line 5, and then used as an argument of `convert` at Line 7. The string `s2`, is not tainted and is passed to `convert` at Line 8. A tainted and an untainted upper case string are returned, respectively. Finally, only the untainted variable reaches the sink in Line 9.

```
1  String convert(String low)
       {
2    return low.toUpperCase();
3  }
4  void example(){
5    String s1 = source();
6    String s2 = new String();
7    String c1 = convert(s1);
8    String c2 = convert(s2);
9    sink(c2);
10 }
```

Listing 2.4.3: Context sensitivity example.

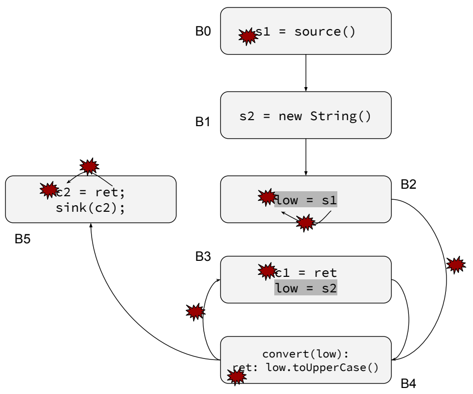


Figure 2.4: Extended control-flow graph of Listing 2.4.3

A context-insensitive analysis does not consider the calling context and treats each call and return statement as "goto" operations. This analysis uses an extended control-flow graph, as the one we show in Figure 2.4 for Listing 2.4.3. Block `B4` is the function `convert`. Block `B2` contains the call site at Line 7; it sets the parameter `low` to `s1` and jumps to the beginning of `convert`, at `B4`. Similarly, `B3`, which is reached from the end of `convert` (block `B4`), contains the call site at Line 8. In this case, we take the return value from `convert` and assign it to `c1`. We then set the parameter `low` to `s2` and call `convert` again, by jumping to `B4`. Finally, block `B5` represents the return from the second call and the final invocation of sink.

If we treat this graph as if it were a single procedure, we would conclude that coming into `B4`, `low` can both be tainted or not, but we do not know for sure. Therefore, the return value (`ret`) can be both tainted or not. In order to preserve soundness, the analysis assumes the return value to always be tainted if there is at least a tainted value that reaches the call site. In this example, `s1` is tainted, and reaches the call site in Line 7 (block `B2`), which causes the return value in block `B4` to be tainted. Therefore, `c2` in `B5` is

```
1 A a = new A();
2 taint(a);
3 a = new A();
4 sink(a);
```

Listing 2.4.4: Object Sensitivity Example

wrongly tainted causing the false positive in Line 9.

On the other hand, a context-sensitive analysis overcomes this limitation by separating the results for each calling context. Therefore, it can understand that `c1` is tainted while `c2` is not.

Object-sensitivity. An object-sensitive analysis distinguishes between different instances of objects of the same type. For example, consider Listing 2.4.4. We instantiate `a`, a new variable holding a reference to an object of type `A` (Line 1), which we then taint (Line 2). Following the taint operation, we reassign `a` to a new instance of the same type `A` (Line 3) and pass it to a sink (Line 4). An object-sensitive analysis distinguishes between different instances of `A` and therefore does not report any information flow. On the other hand, a non-object-sensitive analysis only knows that an object of type `A` is tainted and therefore assumes that `a` is tainted at the sink, introducing a false positive (Line 4).

Field-sensitivity. In an object-oriented language, such as Java, classes can specify different attributes or fields. Every field can, in turn, be a complex object with more attributes, and so on recursively.The initial object from where the recursion begins is known as the base object. A taint analysis must consider this structure when it propagates a taint. A conservative approach would always taint the base objects as a whole, even when only one of the fields was tainted. Instead, a field-sensitive analysis distinguishes different fields of the same objects. Consider the code

```
1 A a = new A();
2 A a1 = new A();
3 a.fld1 = source();
4 sink(a.fld2)
```

Listing 2.4.5: Field Sensitivity Example (1)

```
1 A a = new A();
2 A a1 = new A();
3 a.fld1.subf1 = source();
4 sink(a.fld1.subf2)
```

Listing 2.4.6: Field Sensitivity Example (2)

in Listing 2.4.5.

Without the field sensitivity, the analysis lumps all the fields together. In particular, the whole base object `a` is tainted. The analysis will falsely report Line 4 as flow to the sink. Conversely, a field-sensitive analysis distinguishes different fields of the same object. This prevents the false positive as now only `a.fld1` is tainted and not `a` as a whole. In this example, there is only one level of depth when accessing `a` fields. However, deeper nesting occur. For example, in Listing 2.4.6, the analysis needs to consider at least two levels of depth, or a false positive occurs in Line 4. To keep track of multiple dereferences, and therefore, to retain precision, one needs an abstraction that can capture the base object plus a sequence of field. This abstraction is called an *access path*. An access path of length zero taints the whole base object, while an access path of length N, considers fields up to the N-th field. If the sequence of accessed fields is greater than N, then the whole base object is tainted. For Listing 2.4.6, an access path of two would prevent the false positive.

2.4.2 Fuzz Testing

Fuzz testing, or simply *fuzzing*, is a dynamic technique to analyze computer programs [49] by repeatedly executing a program with invalid, unexpected, or random inputs. The execution is then monitored for crashes, memory leaks, or built-in exceptions. This can expose coding flaws that can potentially lead to security vulnerabilities. For this analysis to be effective, the generated inputs need to be semi-valid, i.e., able to pass the parser, but invalid to trigger corner cases and program exceptions. From a security perspective, fuzz testing can be successfully used to find serious software vulnerabilities (e.g., buffer overflows, memory leaks, etc.).

Tools for fuzz testing are called fuzzers. There are three main categories of fuzzers: black-box, white-box, and gray-box fuzzers. Black-box fuzzers are designed to work with no knowledge of the internals of the program under test. Their goal, given a specification for a program, is to find inputs which cause the program behavior to violate that specification. These specifications describe, for example, the structure of input or output (such as, their types, or simply memory safety). Because there is no knowledge of the program internals, to be exhaustive, this approach needs to try every possible input condition as a test case. Even if possible in principle, it is almost always unachievable in practice.

White-box fuzzers have a systematic knowledge of the program code. They combine this information with information gathered by executing the program. These fuzzers leverage code knowledge to find information about the input to use with other fuzzing strategies. This usually consists of a preliminary static analysis aimed at enhancing the effectiveness of fuzzing.

Other white-box fuzzing approaches are based on dynamic symbolic execution [50], a variation of symbolic execution [51]. The program is executed with a well-formed input and, during its execution, constraints on inputs are collected from conditional statements. These constraints can then be negated and solved with a constraint solver, generating new inputs that exercise new control-flows in the program.

Similar to black-box fuzzing, white-box fuzzing cannot always exhaustively test a program. The main problem is that exploring all the possible executions (i.e., different runs of the program) can be unfeasible. There can be a number of different paths, leading to path explosions and preventing the analysis from ever terminating. Moreover, compared to the black-box fuzzers, white-box ones suffer from a higher overhead. This is because their implementations often require dynamic instrumentation and Satisfiability Modulo Theories (SMT) solving.

Gray-box fuzzers are a middle ground. They obtain some information on the internals of the program under test. Unlike white-box fuzzers, gray-box fuzzers do not reason on the whole program semantics. Instead, they leverage light static analysis and/or program execution to gather knowledge of the program. For example, they may lightly instrument the code to obtain information on code coverage. This can prove very effective in deciding how to generate inputs, as the fuzzer can avoid to generate inputs leading to already explored paths. One of the most successful fuzzers in this category is the American Fuzzy Lop (AFL) [23].

Fuzzing has been successfully used to find vulnerabilities on Android applications [52, 53, 17, 54, 19, 16, 18, 55]. These fuzzers aim is to navigate an application UI as comprehensively as possible, to trigger as many unexpected behaviors as possible. There are two approaches to stimulate a UI: random and non-random. The random approach simply generates a random series of UI events [55]. Fuzzers of this kind belong to the black-box category, as no knowledge of the application internals is known. On the other hand, non-random approaches need a certain amount of knowledge of the application internals. These fuzzers usually fall in white and gray-box categories.

In this thesis, we use fuzzing (Chapter 5) to dynamically generate models for native components of Android applications (Chapter 6). Due to its simplicity, we adopt a black-box approach. This comes with some advantages:

1. We can directly target the native component, without stimulating the

UI.

2. We avoid to perform binary analysis, which a white- or gray-approach requires.

2.4.3 Performance Metrics

As we saw in the previous sections, analysis techniques might be incorrect. For example, we discussed how different levels of sensitivity for taint analysis impact the quality of the results and how different fuzz testing techniques are more effective than others at discovering bugs. Therefore, an analysis is meaningless if it is not accompanied by an evaluation of its performance.

To place an evaluation into context, we must define the two classes of positive and negative outcomes. In this thesis, we apply fuzzing and taint analysis with the goal of finding potential security vulnerabilities and, therefore, define these two classes as follow:

Definition 1 (Positive Class)**.** Code paths that results in a potential security vulnerability.

Definition 2 (Negative Class)**.** Code paths that do not results in a potential security vulnerability.

Notice that these definitions are not conventional to classic program analysis where the analysis aims to verify the correctness of a program.

We then evaluate a tool based on its ability to correctly detect all positive cases and correctly reject the negative ones. To this end we can introduce the following definitions:

Definition 3 (True Positive (TP) and True Neative (TN))**.** We define as TP all the cases belonging to the Positive Class that the were correctly reported. As dual, we define TN as all the cases belonging to to the Negative Class that were correctly not reported.

Definition 4 (True Negative (TN) and False Negative (FN))**.** We define as TN all the cases belonging to the Negative Class that were correctly rejected. As dual, we define as FN all the cases belonging to the Negative Class that were wrongly rejected.

We can then define the following two metrics to evaluate our analysis:

Definition 5 (Precision)**.** The percentage of the positive reported results that are actually true:

$$P = \frac{TP}{TP + FP} \tag{2.1}$$

Definition 6 (Recall)**.** The percentage of positive results that are actually being reported.

$$R = \frac{TP}{TP + FN} \tag{2.2}$$

The perfect analysis would maximize both precision and recall, ideally having them equals to 1. Unfortunately, it is not possible to achieve the perfect result and compromises need to take place. For example, a security bug discovery tool implementing static analysis techniques presents an higher number of FPs as opposed to a tool implementing a dynamic approach.

Static analyses often perform over-approximations to include all possible cases at the cost of including spurious ones. However, they also perform under-approximations in certain cases. For example, certain part of the code might be left out the analysis (e.g., native code in Android).

In contrast, dynamic analysis techniques take the opposite approach. At the extreme, they only ever report a result if they can validate it and be sure of its validity. However, the validation step is expensive and under-approximations are needed. This results in analyses with virtually no FPs but with an higher number of FNs.

The natural question to ask is then whether you should optimize an analysis to achieve higher precision or higher recall? The answer is that it depends on a case to case bases and it comes down to how dangerous is a FP as opposed to a FN. For example, consider the scenario where we want to deploy a security program analysis in the life cycle of a large application. In this case it is critical to minimize the number of FPs. Even 1% of FPs scaled on billions line of code is high. In fact, the result analysts are not necessarily experts in the security domain and in the best case scenario they would seek help, while in the worst they might entirely disregard the analysis and introduce a security flaw.

Consider now the scenario where we want to use the tool to perform penetration testing of a specific application. In this case the analyst is a security expert and would not require any extra help to understand the result of the analysis. Therefore, having more FPs would be tolerable as it would enable us to increase recall and expand the search scope.

As a final example, imagine a firm dealing with sensitive data needs to install new software and wants to be sure that software is not malware. In this scenario, a FN could have a catastrophic impact as opposed to a FP. Therefore, that firm would probably go for an approach that would reduce the number of FNs, accepting a higher number of FPs.

In this thesis, we present different tools mainly targeted to an expert domain. However, there is room for our techniques to applied in a different context.

2.5 Android Static Analysis

In this section, we discuss the status of Android static analysis. In Section 2.5.1, we introduce the existing tools that can be used to carry out reversing of Android application. In particular, we discuss Soot [56, 57], which is the base framework for our analysis in Chapter 3. We then conclude in Section 2.5.2 with a discussion on Android taint analysis. Here, we focus our attention on Flowdroid [3], a taint analysis tool for Android that we adopt as part of our analysis in Chapter 3.

2.5.1 Android Reversing Tools

Among the many Android reversing tools available, in this thesis we use Soot [56, 57]. Initially, Soot was born as a Java compiler and then it evolved into a static analysis framework and transformation tool for Java and Android applications.

Soot transforms Java byte-code into an intermediate representation, Jimple. While Java applications are natively supported, Android support came later with Dexpler [58], a decompiler that converts Dalvik byte-code to Jimple. Jimple (Java sIMPLE) consists of only 15 instructions and it is a stack-less and three addresses representation of the byte-code. Methods in Jimple consists of a body and a list of local variables. The body of a method is effectively a graph of statements that represents the behavior of the method.

The Soot framework has been extended by the community across the years and it implements various analyses [59]. For example, it can reconstruct an application's call graph, perform data flow and point-to analyses, etc. Moreover, one can use soot to automate the instrumentation of applications byte-code. Different research tools base their analysis on Soot, including Flowdroid [3] a state-of-the-art taint analysis engine for Android applications.

Soot is not the only existing tool/framework to reverse Android applications. For example, Androguard [60] is a python tool that allows to analyze and reconstruct the call graph of Android applications. However, Androguard does not provide an out of the box instrumentation. More tools exists that transform Dalvik byte-code to an intermediate representation and vice-versa. For example, Dex2Jar [61] uses the Jasmin [62] disassembler to transform Dalvik to Java byte-code, which is then suitable for loading by a Java runtime system. Dex2Jar can be used to disassemble and re-assemble an Android applications and, therefore, one can use it for instrumentation. Another tool is apktools [63], which uses Smali [64] as an intermediate representation. Likewise, one can use apktools to instrument an Android app. However, differently from Soot, these tools lack of modules to automate the instrumentation process and to perform known static analyses.

In this thesis, we decided to use soot as framework for our analyses. The main reason for this choice is that Soot enable us to automate the instrumentation process. Moreover, we feel that the large open source community and support is an advantage as it can speedup the development process.

By using soot we inherit its limitations. For example, recent research [58] shows that Soot intermediate representation is not always the most accurate when compared to other tools, such as apktools. These inaccuracies can alter the semantic of the re-assembled application, introducing imprecisions in the final analysis. As we discuss in Chapter 4, we use Soot to instrument an Android application. However, in our evaluation Chapter 4 we did not experience imprecision due to Soot re-assembling process.

2.5.2 Android Taint Analysis Tools: Flowdroid

Performing data-flow analysis on Android applications presents with its own set of challenges [65, 66].

1. Android applications lack a clear entry point. Instead, they have several entry points which are called by the framework at runtime. This is usually mitigated by inspecting all the possible entries and constructing separate call graphs for each of them.

2. Android's components (e.g., Activities, Services, etc.) have their life cycle. Indeed, each of them implements certain life cycle methods that are invoked by the framework at runtime. This has to be modeled.

3. User and system events are all handled through a callback system. A callback can be triggered at any time, making it hard for static analysis to exhaustively model the system.

4. Android allows inter-component communication (ICC). Similarly to callbacks, ICC methods are processed by the system at runtime. This hinders static analyzers, so we must rely on heuristics.

On top of this, Android applications are Java-based, inheriting all the challenges of the analysis of Java Programs e.g., reflection, multi-threading, exceptions, etc. Different work has been carried on the topic, resulting in a number of tools that aim at performing data-flow analysis on Android application [67, 4, 3, 2, 6]. These tools are tuned toward different aspects, compromising on others.

In this thesis, we decided to build upon FlowDroid [3] to find potential vulnerabilities in Android Webviews (Chapter 3). Flowdroid is a context-, flow-, field- and object-sensitive taint analysis tool for Android applications which is built on top of Soot and Dexpler. Flowdroid, precisely models the Android life-cycle and can handle data propagating via callbacks. Moreover, it provides models for most of the underlying Android framework methods and adopts a conservative strategy where these model are not present. This strategy taints all the parameters and return values of methods that are invoked with at least one tainted parameter. The tool has limited support to Java reflection, considering only reflective calls with constant string arguments. Also, recent versions of Flowdroid have support for implicit flows. Finally, the tool handles exception taking advantage of the underlying Soot framework. While Flowdroid alone cannot handle ICCs, IccTA [1] has been developed as a Flowdroid extension (often referred as Flowdroid+IccTA) which enable ICC analysis. Unfortunately, neither of Flowdroid and Flowdroid+IccTA can handle native code, introducing possible inaccuracies.

A recent benchmark [65] shows how Flowdroid is among the highest accuracy tool. Its closest competitor is Amandroid [4]. While there is not one single "best tool", Flowdroid is reported to have the average lowest execution time. This is confirmed in the benchmark, where Flowdroid seems to be the best option for large scale evaluations. These result is also confirmed in other evaluation [68], where Flowdroid+IccTA was the fastest and the highest accurate tool on the benchmark.

Our choice of using Flowdroid is not only based on its performance. One key advantage of using this tool is that it is actively maintained and improved and it is supported by the same community of Soot. This is not necessarily the case for the competing tools. For example in certain cases (e.g., DidFail [67]), they do not even run on newer applications. There are three main benefits of choosing FlowDroid: (i) thanks to the community support, we could reduce the development time of our analyses, (ii) we are reassured that bugs in the tool will be addressed and fixed, resulting in our analysis to be improved and (iii) FlowDroid is easy to integrate in our analysis as it is based on the same framework, Soot.

3 Information Flow Analysis on Webviews: BabelView

As we largely discussed in Section 2.2, Webviews are customizable in-app web browsers that expose their users to new security threats. In 2015, 85% of the apps on Google's Play Store contained a Webview [69]. Their popularity led to the development of cross-platform frameworks, such as Apache Cordova, which allow apps to written entirely in HTML and JavaScript. Even otherwise, apps not based on any specific framework often embed Webviews for displaying login screens or additional web content.

Unfortunately, Webviews bring new security threats (Section 2.2.4). One of the main security concerns for Webviews is their ability to intentionally poke holes in the browser sandbox. This functionality enables JavaScript web code to access app- and device-specific features via a JavaScript interface.

Understanding the implications of the JavaScript interface is necessary to assess the overall security of an app. When designing these interfaces, developers think of the functionalities required by their trusted JavaScript. However, there are several ways an attacker can inject malicious code into the Webview [70, 69] and, therefore, access the interfaces.

Concerns about the security of the exposed interfaces were raised in previous work [39, 71, 72]. However, not all the interfaces are dangerous or offer meaningful control to the attacker. Therefore, we focus on assessing the risk posed by every single interface, focusing on the most dangerous cases, and providing meaningful feedback to developers.

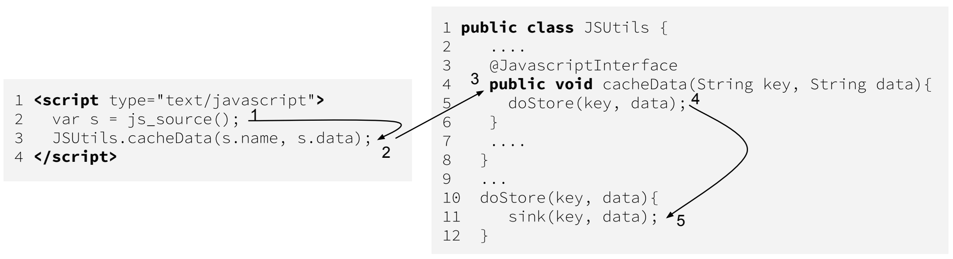
We rely on static analysis to evaluate the potential impact of an attack against Webviews, concerning the nature of the JavaScript interfaces. Our key idea is that we can instrument an app with a model of potential attacker behavior that over-approximates the possible information flow semantics of an attack. In particular, we instrument the target app and replace its Webviews (and its descendant) with a specially crafted *BabelView*, a Webview that arbitrarily interacts with the JavaScript interfaces. A subsequent taint analysis on the instrumented app then yields new flows made possible by the attacker model.

Instrumenting the target application allows us to build on existing mature tools for Android flow analysis. This design makes our approach particularly robust, which is important on a quickly changing platform such as Android. In addition, since the BabelView over-approximates JavaScript interactions, we inherit any soundness guarantees offered by the flow analysis used.

In the remainder of this chapter, we detail our approach (Section 3.1) and the BabelView implementation (Section 3.2). Finally, in Section 3.3, we discuss work related to ours and conclude, in Section 3.4, discussing BabelView's limitations and applications. We leave the evaluation of BabelView to Chapter 4.

3.1 Data-flow Analysis for Hybrid Android Applications

In this section, we set the scene for our work on Android Webviews. We start by explaining the problematic concerning data-flow analysis for Android JavaScript interfaces (Section 3.1.1). We then outline our attacker



```
<script type="text/javascript">
  var s = js_source();
  JSUtils.cacheData(s.name, s.data);
</script>
```

```
public class JSUtils {
  ....
  @JavascriptInterface
  public void cacheData(String key, String data){
    doStore(key, data);
  }
  ....
}
...
doStore(key, data){
   sink(key, data);
}
```

Figure 3.1: *JavaScript to Java Flow.* A variable is tainted in JavaScript (on the left) and reaches a sink in Java (on the right).

model (Section 3.1.2), which enables us to avoid reasoning on the JavaScript codebase (Section 3.1.3 and Section 3.1.4).

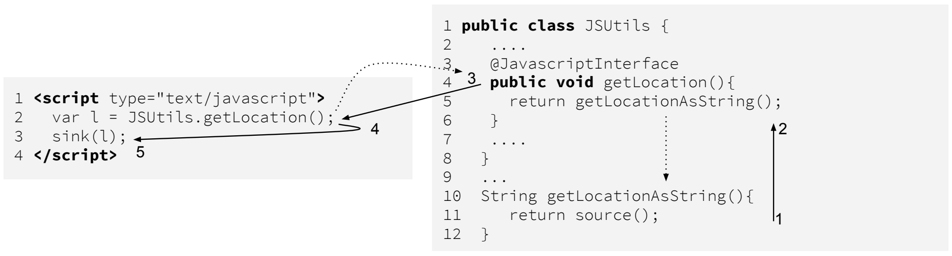
3.1.1 The Problem of JavaScript Interfaces

JavaScript interfaces are only invoked by the JavaScript loaded within the Webview. Performing data-flow analysis without considering the apps' web component leads the analysis to miss all information related to the JavaScript interfaces. There are two cases of interest:

1. A source in JavaScript reaches a sink into Java via a JavaScript interface.

2. A source in Java reaches a sink to JavaScript via a JavaScript interface.

JavaScript to Java Consider Figure 3.1, where `JSUtils` is the same interface object we previously introduced in Listing 2.2.2 and Listing 2.2.3. In the JavaScript snippet, the interface `cacheData` is invoked with inputs coming from a source. This information flows to the `doStore` method and eventually to the Java sink. A data-flow analysis unaware of the JavaScript would miss the flow.



```
1 <script type="text/javascript">
2   var l = JSUtils.getLocation();
3   sink(l);
4 </script>
```

```
1 public class JSUtils {
2    ....
3   @JavascriptInterface
4   public void getLocation(){
5     return getLocationAsString();
6   }
7   ....
8  }
9  ...
10 String getLocationAsString(){
11    return source();
12 }
```

Figure 3.2: *Java to JavaScript Flow.* A variable is tainted in Java (on the right) and reaches a sink in JavaScript (on the left).

Java to JavaScript Consider Figure 3.2 as an example, where `JSUtils` is the same interface object we previously introduced in Listing 2.2.2 and Listing 2.2.3. The JavaScript code invokes the JavaScript interface `getLocation` (Line 2 of JavaScript snippet), getting the user location from Java. Since this information comes from a source method (Line 11), we have a flow. Unfortunately, excluding JavaScript from the analysis would fail to find this flow.

The analysis of Java interface objects comes with the burden of analyzing JavaScript. However, we show that we can relax this constraint by considering an appropriate attacker model. We introduce this attacker model in Section Section 3.1.2 and, in Section 3.1.3, we detail how we can avoid analyzing JavaScript code.

3.1.2 Attacker Model

Our overall goal is to identify vulnerabilities in Android applications. Our insight is that injection vulnerabilities are difficult to avoid with current mainstream web technologies. For example, Android web applications are susceptible site-specific XSS attacks [72, 39, 33]. While these attacks seriously compromise users safety, their severity and impact might increase if JavaScript interfaces are available. Malicious JavaScript can exploit these interfaces to gain capabilities that would have not had otherwise. For example, it could access the phone location if there was an interface with that functionality.

Other possible injection vulnerabilities exist. Any standalone browser that allows loading content via insecure HTTP has this vulnerability (while calling this a "vulnerability" may be controversial, it clearly has security implications and has led to an increasing adoption of HTTPS by default). The ubiquity of advertisement libraries in Android apps further increases the likelihood of foreign JavaScript code gaining access to JavaScript interfaces.

Following this insight, we aim to pinpoint the risk of using a Webview embedded in an app. To this end, we assess the *degrees of freedom* an attacker gains from injecting code into a Webview with a JavaScript interface, which determines the potential impact of an injection attack.

For our analysis, we consider an attacker model consisting of arbitrary code injection into the HTML page or referenced scripts loaded in the Webview. To abuse the JavaScript interface, the attacker then only requires the names of the interface methods, which can be obtained through reverse-engineering. Note that even a MITM attack becomes more powerful with access to the JavaScript interface: the interface can allow the attacker to manipulate and retrieve application and device data that would not normally be visible to the adversary. For instance, consider a remote access application with an interface method `getProperty(key)`, which retrieves the value mapped to a key in the application's properties. Without accessing the interface, an attacker may only ever observe calls to `getProperty` with, say, the keys `"favorites"` and `"compression"`, but the attacker would be free to also use the function to retrieve the value for the key `"privateKey"`.

Adopting this attacker model help us to evaluate the BabelView findings. While our analysis is capable of flagging dangerous JavaScript interfaces, it is not always possible for an attack to target them. By adopting a MITM attacker model, we can automate the process of injecting JavaScript into insecure connections and evaluate the feasibility of an injection (Section 4.1.5). Adopting a different attacker model, such as one base on XSS, this task would have been harder as we would require to analyze all the possible web contented that a WebView could load.

3.1.3 Instrumenting for Data-flow

Our approach is based on static information-flow (or taint) analysis. We aim to find potentially dangerous information flows from injected JavaScript into sensitive parts of the Java-based app and vice-versa. At first glance, this appears to require expensive cross-language static analysis, as recently proposed for hybrid apps [73, 74]. However, we can avoid analyzing JavaScript code because our attacker model assumes that all JavaScript code is controlled by the attacker. Therefore, we want to model the actions performed by *any possible JavaScript code*, and not that of developer-provided code that is supposed to execute in the Webview.

To this end, we perform information flow analysis on the application instrumented with a representation of the attacker model in Java, such that the result is an over-approximation of all possible actions of the attacker (we discuss alternative solutions in Section 3.4). We replace the Android `WebView` class (and custom sub-classes) with a *BabelView*, a Webview that simulates an attacker specific to the app's JavaScript interfaces. We then apply a flow-, field-, and object-sensitive taint analysis [3] to detect information flows that read or write potentially sensitive information as a result of an injection attack.

The BabelView provides tainted input sources to all possible sequences of interface methods and connects their return values to sinks, as shown in Algorithm 1. Here, `source()` and `sink()` are stubs that refer to sources

Algorithm 1: Information flow attacker model

```
1 while true do
2     choose iface ∈ JS-interfaces;
3     result ← iface(source(), source(), ...);
4     sink(result)
```

and sinks of the underlying taint analysis. The non-deterministic enumeration of sequences of interface method invocations is necessary since we employ a flow-sensitive taint analysis. This way, our model also covers situations where the information flow depends on a specific ordering of methods; for instance, consider the following modification to `getLocation` in `JSUtils`:

```
String location;

@JavascriptInterface
public void getLocationAsString() {
  this.location = locationToString();
}
@JavascriptInterface
public String getLocation() {
  return this.location;
}
```

Here, a call to `getLocationAsString` must precede any invocation of `getLocation` to cause a leak of sensitive information (the user location). The flow-sensitive analysis correctly distinguishes different orders of invocation, which helps to reduce false positives. In the BabelView, the loop in Algorithm 1 coupled with non-deterministic choice forces the analysis to join abstract states and over-approximate the result of all possible invocation orders. We do this to avoid enumerating all the invocations.

Figure 3.3 illustrates our approach. We annotate certain methods in the

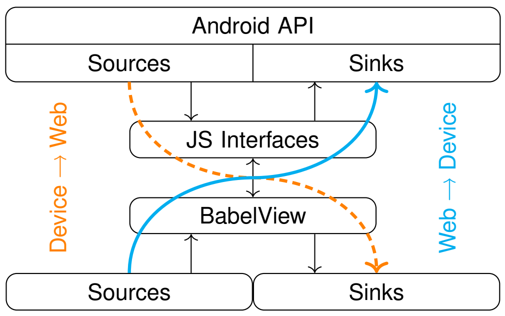


Figure 3.3: *BabelView Approach.* BabelView models flows between the attacker and sensitive sources and sinks in the Android API that cross the JavaScript interface.

Android API as sources and sinks (Section 3.2.4), which may be accessed by methods in the JavaScript interface. The BabelView includes both a source passing data into the interface methods and a sink receiving their return values to allow detecting flows both from and to JavaScript. The source corresponds to any data injected by the attacker, and the sink to any method an attacker could use to exfiltrate information, e.g., a simple web request.

3.1.4 Preserving Semantics

Our instrumentation eliminates the requirement to perform a cross-language taint analysis and moves all reasoning into the Java domain. However, we must make sure that, apart from the attacker model, the instrumentation preserves the original application's information flow semantics. In particular, we need to integrate the execution of the attacker model into the model of Android's application life cycle used as the basis of the taint analysis [3]. We solve this by overriding the methods used to load web content into the Webview (such as `loadUrl()` and `loadData()`) and modifying them to also call our attacker model (Algorithm 1). This is the earliest

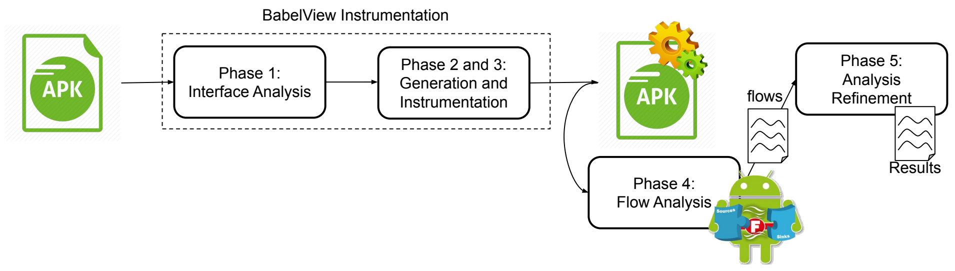

BabelView Instrumentation
APK
Phase 1: Interface Analysis
Phase 2 and 3: Generation and Instrumentation
APK
flows
Phase 5: Analysis Refinement
Results
Phase 4: Flow Analysis

Figure 3.4: *BabelView Phases.* In Phase 1, a preliminary static analysis is performed. In Phase 2 BabelView is generated and then instrumented into the target app in Phase 3. In Phase 4 a taint analysis on the resulting app is performed and in Phase 5 the results are analyzed.

point at which the Webview can schedule the execution of any injected JavaScript code. The BabelView thus acts as a proxy simulating the effects of malicious JavaScript injected into loaded web content.

As the BabelView interacts only with the JavaScript interface methods, it does not affect the application's static information flow semantics in any other way than an actual JavaScript injection would. Obviously, this is not necessarily true for other semantics: for example, the instrumented application would likely crash if it were executed on an emulator or real device.

3.2 BabelView Internals

In this section, we present our implementation of BabelView. Figure 3.5 provides an high level overview: in Phase 1 (Section 3.2.1), we perform a static analysis to retrieve all interface object and methods, and associate them to the respective Webviews. In Phase 2 (Section 3.2.2), we generate the BabelView, and, in Phase 3 (Section 3.2.3), we instrument the target application with it. In Phase 4 (Section 3.2.4), we run the taint analysis on

```
WebView mWebView = new WebView(this);
mWebView.addJavaScriptInterface(new JSUtils(mWebView), "
    Android");
```

Listing 3.2.1: Adding Interface Object to WebView

the resulting application and, finally, in Phase 5 (Section 3.2.5), we analyze the results for flows involving the BabelView.

We implemented our static analysis and instrumentation using the Soot framework [56]; our taint analysis relies on FlowDroid [3]. Overall, BabelView adds about 6,000 LoC to both platforms.

3.2.1 Phase 1: Interface Extraction and Webview Pairing

As the first step of our analysis, we statically analyze the target application to gather information about its Webviews and JavaScript interfaces. We are interested in mapping each Webview class with the class of interface objects added to them.

Using Soot, we can generate the application call graph and precisely resolve callers and callees. We iterate through all classes and methods, identifying all calls to `addJavascriptInterface`, from where we then extract Webviews that will hold interface objects. We illustrate our approach on the code in Listing 3.2.1. The interface object `JSUtils` (recall Listing 2.2.2) is added to `mWebView`. We process the call and extract the type of `mWebView` from the base object and the type of the interface object from the parameters. We then create a mapping between them. Continuing on the example, we would derive the following:

$$WebView \rightarrow JSUtils :: [getLocation, notify]$$

where `JSUtils` holds a reference to all the methods annotated with the

```
public class JSUtilsFile extends JSUtils{
  public JSUtils(WebView webView){
    super(webView);
  }

  @JavascritpInterface
  public void writeToFile(String fileName, String content){
    ...
  }
}
...
WebView mWebView = new WebView(this);
mWebView.addJavaScriptInterface(new JSUtils(mWebView), "
    Android");
```

Listing 3.2.2: Interface Hierarchy

`@JavascriptInterface` annotation.

Hierarchical JavaScript Interface Object JavaScript Interface Objects are classes in the Java domain. As such, they can have a custom hierarchy, which persists in the JavaScript domain. Consider Listing 3.2.2, where instead of `JSUtils`, we add to `mWebView` its subclass `JSUtilsFile`. The analysis as described in Section 3.2.1 would generate the following mapping:

$$WebView \rightarrow JSUtilsFile :: [writeToFile]$$

However, in the JavaScript domain, one can invoke all the annotated JavaScript interfaces. Moreover, one can also invoke the ones defined in the interface object's super-classes. Therefore, our analysis must also include these methods, or our final result will be missing these JavaScript interfaces. To this end, our analysis examines the hierarchy of each interface object extracted. It collects all the exposed interfaces for the current object and

```
1 public class MyWebView extends WebView {...}
2
3 void initInterface(WebView aWebView, JSUtils jsBridge) {
4   aWebView.addJAvascriptInterface(jsBridge, "Android");
5 }
6 ...
7 MyWebView mWebView = new MyWebView(this);
8 initInterface(mWebView, new JSUtilsFile(mWebView));
```

Listing 3.2.3: Polymorphic Webview and Interface Object

all its super-classes. For the example in Listing 3.2.2, our analysis would also consider part of `JSUtilsFile` all the exported interfaces defined in `JSUtils`:

$$WebView \rightarrow JSUtilsFile :: [getLocation, notify, writeToFile]$$

Handling Polymorphism To prevent result loss, our analysis must also consider polymorphism. Consider, for example, Listing 3.2.3. The code is adding an instance of `JSUtilsFile` to `mWebView`, which is an instance of `MyWebView`. However, it happens via a framework method, `initIntrerface`, where the invocation of `addJavaScriptInterface` takes place (Line 4). Our analysis would locate this calling place and would consider `aWebView` as of type `WebView` and `jsUtilsBridge` as of type `JSUtils`. Two problems follow:

1. The paired Webview is of type `WebView`, causing our instrumentation to fail to instrument MyWebView (Line 7)
2. The extracted interface object would miss some annotated interfaces (e.g., writeToFile)

For the Webview, we must process all descendants of its declared class to include the types of all possible instances. For `aWebView`, this means we must instrument all descendants (including anonymous classes) of `WebView`, i.e., `WebView` and `MyWebView`. Similarly, we are interested in the real type of `aBridge`. Again, we must iterate over all sub-classes of its declared type `JSUtils` to ensure capturing the bridge added at runtime. However, since `addJavascriptInterface` is of the unconstrained type Object, this could potentially include all classes. Therefore, we restrict processing to just those sub-classes that contain at least one `@JavascriptInterface` annotation. As a result, we obtain a superset of all interface objects that can be added by this method, i.e., `JSUtils` and `JSUtilsFile`:

$$\begin{aligned} WebView &\rightarrow \{\mathsf{JSUtilsFile} :: [\mathsf{getLocation}, \mathsf{notify}, \mathsf{writeToFile}], \\ &\qquad \mathsf{JSUtils} :: [\mathsf{getLocation}, \mathsf{notify}]\} \\ MyWebView &\rightarrow \{\mathsf{JSUtilsFile} :: [\mathsf{getLocation}, \mathsf{notify}, \mathsf{writeToFile}], \\ &\qquad \mathsf{JSUtils} :: [\mathsf{getLocation}, \mathsf{notify}]\} \end{aligned}$$

3.2.2 Phase 2: BabelView Generation

We generate a BabelView class for each Webview in the mapping. Following up on the example in Listing 3.2.3, in Listing 3.2.4, we provide the resulting BabelView for the Webview `mWebView` of type `MyWebView`. We will use it as a reference through the section for our explanation.

Each BabelView defines a subclass of its Webview (`MyWebView` in this case) and overrides all its parent's constructors so that it can be used as a drop-in replacement. The interface objects are class attributes, which we utilize to invoke the JavaScript interfaces. We initialize these attributes in the `addJavaScriptInterface` method, which we override to extract the reference from its first parameter (Line 10).

To implement the attacker model, the BabelView needs to override all methods that load external resources and could thus be susceptible to JavaScript injection. In particular, we override `loadUrl`, `postUrl`, `loadData`

```
public class BabelView extends MyWebView {
  private JSUtilsFile if0;

  public void BabelView(Context ctx){ super(ctx); }
  // All other constructor of WebView are here

  public void addJavaScriptInterface(Object obj, String
      name){
    super.addJavaScriptInterface(obj, name);
    if(obj instanceof JSUtilsFile) {
      this.if0 = (JSUtilsFile) obj;
    }
  }

  public void loadUrl(String url) {
    super.loadUrl(url);
    while(True){
      switch(random()){
        case 1:
          if0.writeToFile(source());
          break;
        case 2:
          leak(if0.getLocation());
          break;
        case 3:
          if0.notify(source());
      }
    }
  }
  public void leak(){ // stub method }
  public Object source() { return new Object(); }
}
```

Listing 3.2.4: BabelView Example

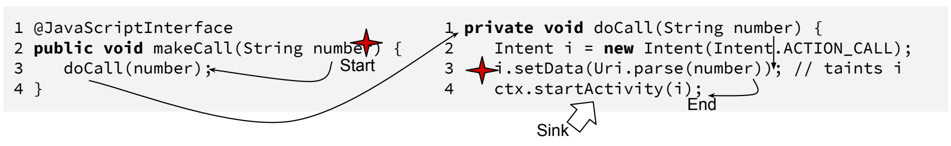
, and `loadDataWithBaseURL`. In the example, we show the implementation of `loadUrl` (Line 14). We invoke its super implementation followed by an implementation of the attacker model in Algorithm 1 (Line 16 to Line 26). Here, the switch case emulates a random pick of JavaScript interface to execute. Depending on the signature of the interface, we leak its return value. Similarly, we provide tainted inputs. To this end, the BabelView is equipped with two stub methods, leak (Line 29), and source (Line 30), representing a tainted sink and a tainted input, respectively.

3.2.3 Phase 3: Instrumentation

We instrument the application to replace its Webviews with our generated BabelView instances. The instrumentation is case-dependent on how the Webview is instantiated (see Section 2.2). If it is created via an ordinary constructor call, that constructor is replaced with the corresponding constructor of its BabelView class. If the Webview is created via the Activity XML layout, our instrumentation searches for calls to `findViewById` , which the app uses to obtain the Webview instance (e.g., to add the JavaScript interface to it). To identify the calls to `findViewById` returning a Webview, our instrumentation identifies explicit casts to a Webview class. Because we do not parse the XML layout itself, we arbitrarily choose one of the constructors of the BabelView. While this could potentially be a source of false positives or negatives, it would require a highly specific and unconventional design of the Webview class that we never seen during our evaluation in Chapter 4.

3.2.4 Phase 4: FlowDroid Data-flow Analysis

We perform a static information flow analysis on the instrumented application to identify information flows involving the attacker model. Since our approach relies on instrumenting the application under analysis, it



```
@JavaScriptInterface
public void makeCall(String number) {
   doCall(number);
}
```

```
private void doCall(String number) {
   Intent i = new Intent(Intent.ACTION_CALL);
   i.setData(Uri.parse(number)); // taints i
   ctx.startActivity(i);
```

Figure 3.5: *TaintWrapper Intent Flow.* The method `Uri.parse` is called with `number`, which is tainted. Because `Uri.parse` is part of the TaintWrapper, its return value is also taint. Similarly, `setData` is modeled in the TaintWrapper causing the intent `i` to be tainted. A flow is then detected to `startActivity`.

is agnostic to the specific flow analysis. We decided to rely on the open source implementation of FlowDroid [3], inheriting its context-, flow-, field-, and object-sensitivity, as well as its life cycle-awareness. Sources and sinks are selected corresponding to sensitive information sources and device functions, modified from the set provided by SuSi [75]. We further include the sources and sinks used in the BabelView classes. The information flow analysis abstracts the semantics of Android framework methods. FlowDroid uses a simple modeling system (the TaintWrapper), where any method can either (i) be a source, (ii) be a sink, (iii) taint its object if any argument is tainted and return a tainted value if its object is tainted, (iv) clear taint from its object, (v) ignore any taint in its arguments or its object. We extended the TaintWrapper with several models that were relevant for the types of vulnerabilities we were interested in, e.g., to precisely capture the creation of Intents from tainted URIs.

Finally, information flows indicating that sensitive functionality is exposed via the JavaScript interface are identified, triggering an *alarm* showing a potential vulnerability. An alarm consists of a detected dangerous information flow enhanced with its semantics. For instance, consider the flow in Figure 3.5, where the JavaScript interface `makeCall` is used to perform phone calls. The variable `number` comes from the BabelView source and, therefore, it is tainted. The flow goes to `doCall`, where an Intent is created. After the intent creation, `number` flows to `Uri.parse` whose return value is an input of the intent `setData`. These two methods are from the android framework and, therefore, we include them into FlowDroid TaintWrapper. The intent `i` is then tainted and flows to the sink `startActivity`, indicating that an attacker can perform calls on behalf of the user.

3.2.5 Phase 5: Analysis Consolidation

We consolidate the data-flow analysis by further analyzing preferences and intents.

Preferences Taint analysis cannot distinguish between individual key-value pairs in a map. `Preferences` are a commonly used map type in Android apps that often store sensitive information as a key-value pair. After the information flow analysis, we consolidate our results by statically deriving values of keys for access to preferences. Our definition of sources and sinks allows to identify both flows from and to the `Preferences` . Given two flows, one inserting and the other retrieving values from `Preferences`, we are interested in understanding whether (i) the value is of the same type and (ii) the access key is the same. If these conditions are met, we have identified a potential leak via Preferences. To determine the key values, we modeled `StringBuilder` and implemented an intra-procedural constant propagation and folding for strings. Finally, if an interface method allows web content to interact with a preferences object, BabelView reports all keys used to access it, since preferences can be used to store sensitive values. This allows us to inspect flows to or from preferences entries, even if these values are not dependent on a specific source in the Android API. We match key names against a list of suspicious entries, which can highlight potential leaks of sensitive app-specific information

(Section 4.2.2). In the same manner, we also highlight suspiciously named interface methods.

Intents Flow analysis can detect situations where Intent creation depends on tainted input. However, it tells nothing about the type of the Intent created, as this depends on specific parameters, e.g, those provided to its `setAction` method. For interpreting results, it is important, however, to know the action of an `Intent` that can be controlled by an attacker.

For any flow that reaches the `startActivity` sink, we perform an inter-procedural backward dependency analysis to the point of the initialization of the `Intent`. If the `Intent` action is not set within the constructor, we perform a forward analysis from the constructor to find calls to `setAction` on the `Intent` object. The analysis may fail where actions are defined within intent filters (XML definitions) or through other built-in methods. As an example, consider Figure 3.5. From the sink `startActivity`, our Intent analysis would backtrack to the Intent creation. The Intent constructor creates an intent with the `Intent.ACTION_CALL` parameter. Therefore, we conclude that an attacker can make phone calls. To increase precision in our inter-procedural analysis, we ensure that the call-stack is consistent with an invocation through the interface method; i.e., the interface method that triggered the flow must be reachable.

3.3 Related Work

We now review work on vulnerabilities and attacks against Webview (Section 3.3.1), discuss related work on policies and access control (Section 3.3.2), and contrast with work on instrumentation-based modeling (Section 3.3.3).

3.3.1 Webview: Attacks and Vulnerabilities

Webview vulnerabilities have been widely studied [13, 76, 37, 71, 69, 72]. Luo et al., give a detailed overview of several classes of attacks against Webviews [13], providing a basis for our work. Neugschwandtner et al. [37], were the first to highlight the magnitude of the problem. In their analysis, they categorize as vulnerable all applications implementing JavaScript interfaces and misusing TLS (or not using it at all). For further precision, they analyzed permissions and discovered that 76% of vulnerable applications requested privacy critical permissions. While this is a sign of poorly designed applications, the impact of an injection exploit very much depends on the JavaScript interfaces, motivating the work of this thesis.

A step forward toward this was made by Bifocals [71], a static analysis tool able to identify and evaluate vulnerabilities in Webviews. Bifocals looks for potential Webview vulnerabilities (using JavaScript interfaces and loading third party web pages) and then performs an impact analysis on the JavaScript interfaces. In particular, it analyzes whether these methods reach code requiring security-relevant permissions. However, JavaScript interfaces can pose an (application-specific) risk without making use of permissions. At the same time, not all JavaScript interfaces that make use of permissions are dangerous: for example, an interface method might use the phone's IMEI to perform an operation but not return it to the caller.

The means by which malicious code can be injected into the Webview have been discussed in previous work [39, 33]. Having to interact with many forms of entities, HTML5-based hybrid applications expose a broader surface of attack, introducing new vectors of injection for cross-site-scripting attacks [33]. While these attacks require the user to directly visit the malicious page within the Webview, Web-to-Application injection attacks (W2AI) rely on intent hyperlinks to render the payload simply by clinking a link in the default browser [39]. Both discuss the threat behind JavaScript interfaces, but stop their analysis at the moment where the malicious payload is loaded, without analyzing the implication of the attacker executing the JavaScript interfaces.

A large scale study on mobile web applications and their vulnerabilities was presented by Mutchler et al. [69], but did not study the nature of the exposed JavaScript interfaces. Li et al. [38], studied a new category of fishing attacks called *Cross-App WebView infection*. This new type of attacks exploits the possibility of issuing navigation requests from one app's Webview to another via Intent deep linking and other URL schemata. This can trigger a chain of requests to a set of infected apps.

Most closely related to our work is the concurrently developed *BridgeScope* [35], a tool to assess JavaScript interfaces based on a custom static analysis. Similar to our work, BridgeScope allows to detect potential flows to and from interface methods. BridgeScope uses a custom flow analysis, whereas our approach intentionally allows to reuse state-of-art flow analysis tools. While BridgeScope's flow analysis performs well on benchmarks, there appears to be no specific treatment of Map-like objects such as `Preferences` of `Bundle`.

In recent work, Yang et al. [36], have combined the information of a deep static analysis with a selective symbolic execution to actively exploit event handlers in Android hybrid applications. In *OSV-Hunter* [34], they introduce a new approach to detect Origin Stripping Vulnerabilities. These type of vulnerabilities persist when upon invocation of `window.postMessage`, it is not possible to distinguish the identity of the message sender or even safely obtain the source origin. This is inherently true for Hybrid applications, where developers often rely on JavaScript interfaces to fill the gap between web and the native platform.

3.3.2 Webview Access Control

There have been several proposals to bring origin-based access control to Webviews [77, 78, 79, 80, 81]. Shehab et al. [79] proposed a framework that modifies Cordova, enabling developers to build and enforce a page-based plugin access policy. In this way, depending on the page loaded, it will or will not have the permission to use exposed Cordova plugins (i.e., JavaScript interfaces).

Georgiev et al. presented NoFrank [77], a system to extend origin-based access control to local resources outside the web browser. In particular, the application developer whitelists origins that are then allowed to access device's resources. However, once an origin is white-listed, it can access any resource exposed. Jin et al. [80] propose a fine-granular solution in a system that allows developers to assign different permissions to different frames in the Webview.

Tuncay et al. [78], increase granularity further in their Draco system. Draco defines a policy language that developers can use to design access control policies on different channels, i.e., the interface object, the event handlers and the HTML5 API. Another framework allowing developers to define security policies is HybridGuard [81]. Differently from Draco, HybridGuard has been entirely developed in JavaScript, making it platform independent and easy to deploy on different platform and hybrid development framework. Both Draco and HybridGuard could provide an interesting solution to the problem of securing an interface BabelView is rising an alarm for, without unduly restricting its functionality.

3.3.3 Instrumentation-based Modeling

Synthesizing code to trigger specific function interfaces is not a new problem and traces back to generating verification harnesses, e.g., for software model checking [82, 83]. On Android, FlowDroid [3] uses a model that invokes callbacks in a "dummy main" method, taking into account the life cycle of Android activities. While the problems share some similarity, JavaScript interfaces and Webviews are inherently varied and app-specific. Therefore, we require a static analysis and cannot rely on fixed signatures. Furthermore, because our model represents an attacker instead of a well-defined system, calls can appear out of context anytime web content can be loaded in the Webview, i.e., after a `loadUrl`-like method.

3.4 Limitation and Discussion

This section starts with a discussion of the benefit of BabelView instrumentation (Section 3.4.1). We then present the limitation of BabelView core (Section 3.4.2) and feasibility (Section 3.4.3) analyses. We then conclude discussing how to mitigate the potential vulnerabilities that BabelView finds (Section 3.4.4).

3.4.1 Avoiding Instrumentation

In principle, we could avoid instrumenting the application by summarizing interface methods with an inter-procedural taint analysis. However, to achieve the same precision, the analysis would have to be computationally expensive: on method entry, any reachable field in any reachable object (not just arguments of the interface method) would have to be treated as carrying individual taint. On method exit, the effects on all reachable fields would have to stored, before resolving the effects among all interface method summaries. Our instrumentation-based approach not only avoids this cost, but also allows us to factor out flow analysis into a separate tool, a design choice that improves robustness and maintainability.

3.4.2 Analysis Limitations

Our system, together with the underlying flow analysis, is subject to common limitations of static analysis and hence can fail to detect Webviews and interfaces instantiated via native code, reflection, or dynamic code loading. In principle, this currently allows a developer intent on doing so to hide sensitive JavaScript APIs. However, we focus on benign software and vulnerabilities that are honest mistakes rather than planted backdoors. Still, we note that BabelView would automatically benefit from future flow analyses that may counteract evasion techniques.

A potential source of false positives is that BabelView does not distinguish Webview instances of the same type and will conservatively join the JavaScript interfaces of all instances. Furthermore, our analysis loses precision when reporting indirect leaks via `Preferences` or `Bundle`. As mentioned in Section 3.2.4, we connect sensitive flows into the application preferences with flows from the preferences to the instrumented sink method in BabelView. While this is sound and will conservatively capture any information leaks via preferences, it is not taking into account any temporal dependencies between storing and retrieving the value. A different treatment of this would be a potential source of false negatives, since preferences persist across application restarts.

3.4.3 Attack Feasibility

In our feasibility analysis, we actively try to inject JavaScript code into a Webview, aiming at identifying whether the reported interface object is present in the Webview. The presence of the interface object means that all its interface methods are available to use, including the one BabelView reported. However, we do not actively invoke these methods and thus we cannot be sure of their exploitability.

3.4.4 Mitigating Potential Vulnerabilities

To avoid giving potential attackers control over sensitive data and functionality, developers can follow a set of design principles. First of all, Webview contents should be exclusively loaded via a secure channel. Second, as mentioned in the Android developer documentation, Webviews should only load trusted contents. External links have to be opened with the default browser. For also protecting against malicious ads or cross-site-scripting attacks, JavaScript interfaces should offer an absolute minimum of functionality and avoid arguments as far as possible. Finally, recent work also introduced novel mechanisms to enforce policies on hybrid applications (see Section 3.3.2).

4 BabelView Evaluation

In this chapter, we present our evaluation of BabelView. In Section 4.1, we show the results we obtained in our large scale analysis of the Android Play Store. We then conclude with Section 4.2, where we show some interesting case studies and how BabelView could help to find real vulnerabilities.

Unfortunately, we were unable to conduct a direct comparison with BridgeScope [35], the work most closely related to ours. Despite helpful communication, the authors were ultimately unable to share neither their experimental data nor their implementation with us. In the spirit of open data, we make all our code and data available[1].

4.1 Play Store Large Scale Analysis

In this Section we present the results of our study of vulnerabilities in Android applications. Below, we explain our methodology (Section 4.1.1) and ask the following research questions to evaluate our approach:

1. **Can BabelView successfully process real-world applications?** We conduct a study on a randomly selected set of applications from the

[1]`https://github.com/ClaudioRizzo/BabelView`

AndroZoo [84] dataset and provide a breakdown of all results (Section 4.1.2).

2. **What are the precision and recall of our analysis?** We manually validate a random sample of apps, estimating overall precision and recall (Section 4.1.4).

We also shed light on the current state of Webview security on Android with the following questions:

3. **How frequent are different types of alarms?** We report results per alarm, which provides an insight into the prevalence of potential vulnerabilities (Section 4.1.3).

4. **Are there types of potential vulnerabilities that are likely to occur in combination?** We compute the correlation between alarms raised by our analysis and analyze our findings (Section 4.1.6).

4.1.1 Methodology

We obtained our dataset from AndroZoo [84], using the list of applications available on July 22-nd, 2016, when it contained about 4.4 million samples. We downloaded a random subset of 209,069 apps, and then filtered our dataset for applications containing a Webview, a call to `addJavascriptInterface` , and granting permission to access the Internet. As a result, we obtained 62,674 total applications. Finally, from the obtained sample, we randomly extracted 25,000 applications found in the Google Play Store, which we used for our analysis.

We ran BabelView on five servers: one 32-core with 250GB of RAM and four 16-core with 125GB of RAM. Each application took on average 180 seconds to complete. The high precision of FlowDroid's information flow analysis can lead to long processing time in the order of hours. Moreover, BabelView instrumentation exposes new paths for the subsequent flow analysis which, therefore, might take longer to complete. For our analysis to be practical we then had to set a time limit. Given enough time, having an higher limit would increase the accurancy of our results as, in principle, we would be able to analyse more applications. However, for a large scale evaluation with limited resources, this would have required an impractical long time. Therefore, we performed preliminary experiments to tune the time limit of our analysis. In this process, we found that apps taking longer than 15 minutes would often go over an hour.

A positive effect of our instrumentation-based approach is that we benefit from improvements in the underlining flow analysis. Indeed, over the duration of this project, we saw a noticeable accuracy enhancement from the constant improvements on FlowDroid.

Each application underwent three main phases: (i) BabelView instrumentation, (ii) FlowDroid analysis on the instrumented app and (iii) analysis of the resulting flows to identify suspect flows and raise alarms. On the reported applications, we performed a feasibility analysis. We searched the app for plain `http://` URLs and assess the resilience of the app against injection attacks.

4.1.2 Applicability

We summarize the outcome of running our tool chain in Figure 4.1. Running our tool chain on the 25,000 target applications resulted in 1,286 general errors and 3,837 flow analysis timeouts. The remaining 19,877 apps were successfully analyzed and we obtained the following breakdown: 832 applications had no interface objects at all or no interface methods in case the target API was version 17 or above; 14,048 applications had no flows involving our attacker model; and 4,997 were reported as dangerous, i.e., containing flows due to the attacker behavior. This amounts to a

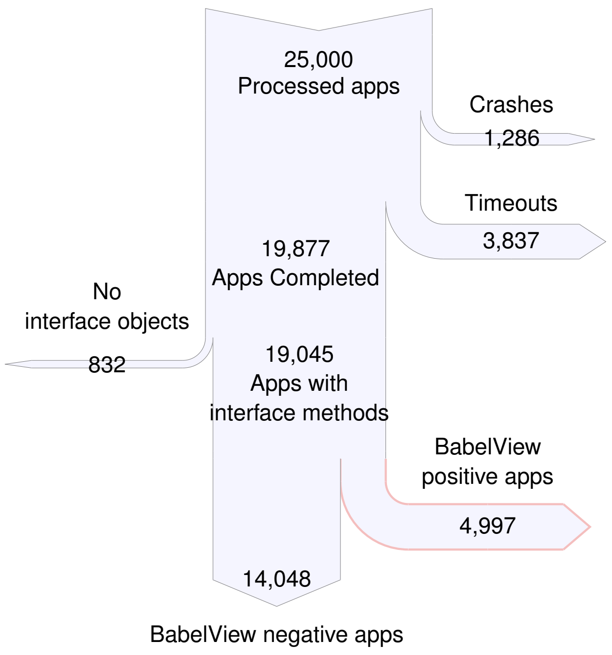


Figure 4.1: *Processed Apps Breakdown.* Breakdown of processed applications and analysis results.

rate of 26.2%. We investigated the reasons for the crashes, and most happened either due to unexpected byte code that Soot fails to handle or while FlowDroid's taint analysis was computing callbacks.

Among applications with interface objects, we also considered those targeting outdated versions of the Android API, since this is still a common occurrence [85, 86, 87]. When using Webviews prior to API 17, any app is trivially vulnerable to an arbitrary code execution disclosed in 2013[2]. Despite targeting an old API version, if compiled with a newer Android SDK, these applications can still use the `@JavascriptInterface` annotation. While the annotation itself does not provide extra security, these apps may target newer APIs in future releases [32].

4.1.3 Alarms Triggered

We successfully used BabelView to examine 19,877 applications. We found that 4,997 of them triggered an alarm (i.e., our analysis reported a potential vulnerability), meaning that the interface methods could be exploited by foreign JavaScript from injection or advertisement. Table 4.1 shows a breakdown of all the alarms we observed in our analysis. Among the most common alarms, we observed the possibility of writing to the File System (Write File), capability to start new applications (Start App), violation of the Same Origin Policy (Frame Confusion) and the possibility of exploiting the old reflection attack due to Android API prior to v17.

Writing File capabilities show the developers' need for storing app-external data usually coming from an app-dedicated server. We also observed that many applications implement advertising libraries which need to open a new application, usually Google Play Store, to allow the user to download or visualize some information. Unfortunately, the package name of the

[2]https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/

Alarm	#Apps	Alarm	#Apps	Alarm	#Apps
Open File	385	Write File	1,444	Read File	593
TM Leaks	39	Pref. TM Leaks	4	Pref. Connectivity Leaks	4
SQL-lite Leaks	136	SQL-lite Query	438	Pref. SQL-lite Leaks	11
GPS Leaks	43	Pref. GPS Leaks	1	Directly Send SMS	6
Directly Make Calls	19	Call via Intent	314	Email/SMS via Intent	778
Take Picture	7	Download Photo	317	Play Video/Audio	378
Edit Calendar	357	Post to Social	293	Start App	1,321
API prior to 17	1,039	Unknown Intent	1,107	Frame Confusion	1,039
Fetch Class	85	Fetch Constructor	0	Constructor init	13
Fetch Method	85	Method Parameter	622		

Table 4.1: *Number of Apps per BabelView's Alarm Category.* Pref. stands for indirect leaks via a Preference object; TM stands for Telephony Manager.

application to open is given as input to an interface method, enabling a possible attacker to control which app to start. Same-Origin-Policy violations are also very common: this is the case when a `loadUrl` is invoked with input from the interface methods, controlling what is loaded in to a frame. As described by Luo et al. [13], JavaScript executing in an `iframe` runs in the context of the main frame, violating the SOP.

Many applications still target an API version prior to 17 [85, 86, 87], often due to backward compatibility or simply due to confusion in declaring the SDK version. Other alarms involve the possibility to prompt the user with an email or a text message to send, directly sending an SMS or performing a phone call; prompting the user with the call dialer; posting content to social network; interacting with the calendar by creating or editing an event; playing videos or audio; leaking sensitive information like the device ID or phone numbers (i.e., TM Leaks), GPS position, SQL information, etc.

Finally, we shed light on the possible use of Java Reflection inside interface methods. Fetch Class, Fetch Constructor, Constructor init, Fetch Method and Method Parameter are all signs that an attacker controls input used to execute methods via Java reflection. Although these are rare situations and often hard to exploit, they are extremely high reward for an attacker as they can potentially allow to circumvent the `@JavascriptInterface` annotation, leading to arbitrary code execution. We manually analyzed some applications presenting these alarms and in some cases an attacker could take control of a method and its parameters, leading to remote code execution.

4.1.4 Manual Validation

We used manual validation to estimate the accuracy of our analysis. In particular, we sampled and manually analyzed (i.e., reversed and decompiled) 50 applications. We evaluated two aspects:

1. How accurate is BabelView with respect to each individual alarm it raises?
2. Does BabelView function as an effective alarm system for hybrid apps?

We reviewed all the alarms[3] for each app and we established whether an alarm was correctly triggered or correctly not triggered. To this end, we looked at BabelView's output and reversed the respective application to validate the results. BabelView's output consists of a json where the keys represent the alarms, and the values are boolean indicating whether an alarm has occurred. For example, consider the json in Listing 4.1.1. Each alarm, apart from "Intent Control", can either be 1 or 0. In the former case, BabelView raised an alarm for the alarm represented in the corresponding key. The latter case suggests that no alarms were reported. For the "Intent Control" alarm, the analysis is more fine grained, telling the user what

[3]We recall that an alarm consists of a flow enhanced with its semantic, see 3.2.4

```
{
    "Preferences DB Query Exec": 0,
    "Pref DB Leak": 0,
    "Pref TM Leak": 0,
    "Pref Connectivity Leaks": 0,
    "Pref Location Leaks": 0,
    "SQL-lite Leaks": 0,
    "TM Leaks": 0,
    "Connectivity Leaks": 0,
    "Location Leaks": 0,
    "SQL-lite Query Exec": 0,
    "Intent Control": [
      "android.intent.action.VIEW"
    ],
    "File Opening": 1,
    "File Writing": 1,
    "File Reading": 1,
    "Send SMS": 0,
    "Open Socket": 0,
    "Reflection": 0,
    "Frame Confusion": 1,
    "Fetch Class": 0,
    "Fetch method": 0,
    "Constructor Instance": 1,
    "Method Parameter": 0,
    "Fetch Constructor": 0
}
```

Listing 4.1.1: BabelView Output

		Expected	
		Positive	Negative
Reported	Positive	42	10
	Negative	5	1,494

Table 4.2: *Manual Validation Confusion Matrix Per Alarm Base.*

kind of intent action can be triggered via the interface. In the validation process, we checked each one of these alarms to be accurate with respect to what we saw in the decompiled code of the application under analysis.

We present a confusion matrix of our manual validation in Table 4.2. Among all alarms triggered for the 50 applications, we observed 42 TPs (True Positives), 10 FPs (False Positives), 1,494 TNs (True Negatives) and 5 FNs (False Negatives). From this, we can compute a precision of 81% and a recall of 89% for our analysis.

The results obtained are in line with our expectations. Our instrumentation does not alter the semantics of applications other than adding a model of attack behavior. Therefore, our precision depends on the underlining flow analysis. However, more false positives could be introduced due to the object-insensitivity of our instrumentation—i.e., we distinguish types but not instances of Webviews. Similarly, a very low false negative rate is common for data flow analysis; however, FNs are still possible, mainly due to incomplete Android framework.

To evaluate BabelView on a per-app basis, we consider a true positive the case where an app contains at least one potential vulnerability and at least one alarm is raised. True negatives and false positives/negatives follow accordingly. In Table 4.3, we report our results. We observed 19 TPs, 2 FPs, 29 TNs, and 0 FNs, which yields a precision of 90% and a recall of 100%. This suggest that BabelView performs well as an alarm system for potentially dangerous applications. Even if individual alarms can be false positives, the correlation of dangerous interfaces appears to leads

		Expected	
		Positive	Negative
Reported	Positive	19	2
	Negative	0	29

Table 4.3: *Manual Validation Confusion Matrix Per App Base.*

to highlighted apps being problematic with high probability. The false negatives that are present when taken per vulnerability disappear when analyzed on a per app basis.

4.1.5 Feasibility Analysis

To better understand the feasibility of exploiting potential vulnerabilities highlighted by BabelView, we measured the difficulty of performing an injection attack. To this end we use a three-step process: (i) we check the application for TLS misuse using MalloDroid [70]; (ii) we search for hard-coded URLs beginning with `http://`, suggesting that web content could be loaded via an insecure channel; and (iii) we actively injected JavaScript code into Webviews.

MalloDroid reported 61.5% of applications using TLS insecurely and 98.7% of apps were found hard-coding HTTP URLs. In order to actively inject JavaScript, we stimulated each reported application with 100 Monkey[4] events and actively intercepted the connection (using Bettercap[5]), trying to execute a JavaScript payload. Moreover, we set up our own certificate authority and also tried SSL strip attacks. The goal of the injection was to determine whether the reported interface methods were present in the Webview. To this end, we generated JavaScript code checking for the presence of the interface objects reported by the BabelView analysis.

[4]`https://developer.android.com/studio/test/monkey.html`
[5]`https://www.bettercap.org`

We were able to inject JavaScript in 1,275 applications and in 482 cases we confirmed the presence of the vulnerable interface object.

4.1.6 Correlation of Alarms

We were interested in finding correlations among the alarm categories we identified. This does not only account for common patterns of functionality, but also identifies single alarms that taken together could increase the attack capabilities, e.g., combining opening and writing of a file results in writing of arbitrary files.

We can see in the correlation matrix in Figure 4.2 that alarms involving related functionality tend to be positively correlated (in red). For example, opening and writing a file; SQL queries and leaks; and operations involving intents such as call via intent, send email, edit calendar, play video, post to social, and download pictures. While some correlations are evident, some appear incidental, such as intent calls and playing of videos. Based on manual inspection (see Section 4.2), we found that these categories of alarms often appear together in apps using common libraries, e.g., for advertisements.

4.2 Case Studies

In this section, we present some interesting case studies, proving that BabelView can expose real vulnerabilities. We start discussing a banking application, which exposes dangerous JavaScript interfaces (Section 4.2.1. We then show how we could exploit a sports app's JavaScript interface to leak the username and password of a user (Section 4.2.2). We finally conclude presenting our evaluation of the JavaScript interfaces exported by a commonly used Ads library (Section 4.2.3).

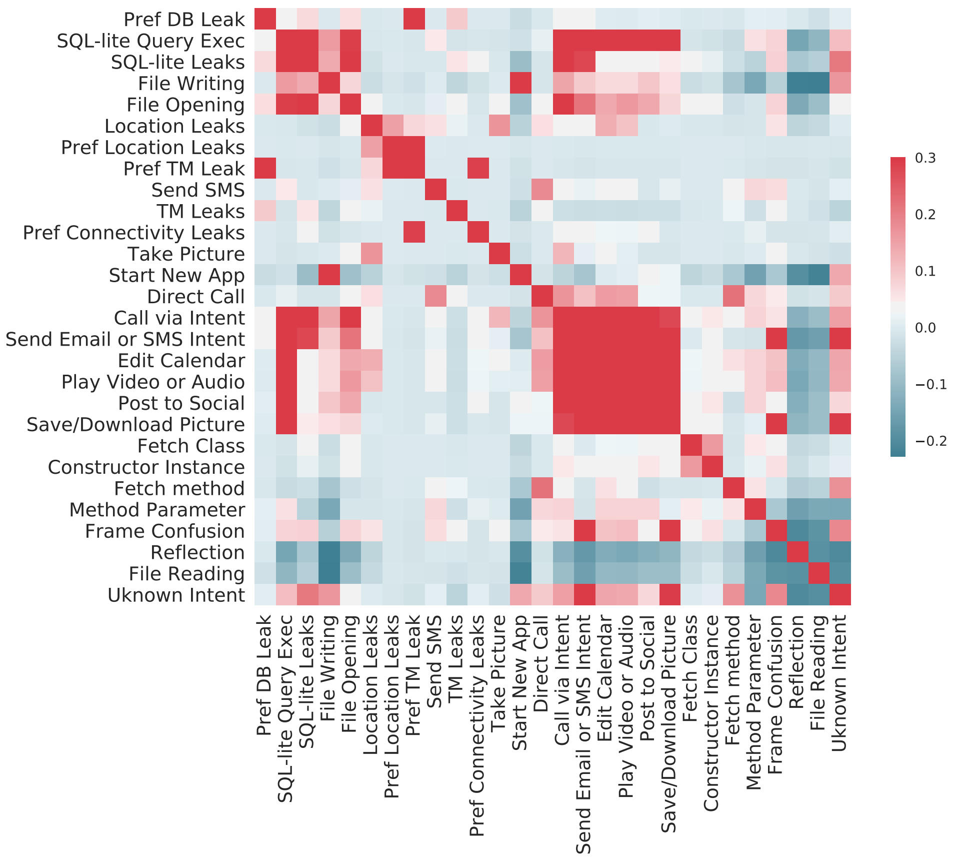


Figure 4.2: *Correlation matrix of alarms.* Red blocks indicate high correlation while blue one low correlation.

4.2.1 MAB Mobile Banking

The JavaScript interface of this hybrid application exports several sensitive methods. The information flow analysis with BabelView flagged it as susceptible to SQL-lite query execution, SQL-lite leaks, file writing, telephony manager leaks, and intent control. We manually reverse-engineered this application and were able to confirm all the alarm arise. In particular, an exploit against the JavaScript interface would not only allow an attacker to place calls to arbitrary numbers and write into the file system, but also to leak messages and initiate payments. The following are some of the interface methods exposed by the application (we expand `callPhone` for illustration):

```
@JavascriptInterface
public void callPhone(String num)
{
  Intent i = new Intent( "android.intent.action.CALL", Uri
      .fromParts("tel", num, null));
  startActivity(i);
}

public String payFriend(...) { ... }
public String payBill(...) { ... }
public String listInbox(...) { ... }
```

We could not actively confirm the exploitability of the application in a test run, since (apart from legal reasons) the interface becomes available only after authenticating. However, from our manual analysis it is apparent that the web content displayed in the Webview is dynamically loaded.

4.2.2 SwingAid

Among our results, we found a game application ("SwingAid Level up Golf") that uses several Webviews and JavaScript interfaces leading to different alarms: SQL-lite leaks via preferences, frame confusion, and telephony manager Leaks. Moreover, we discovered the value *loginPwd* among preferences keys accessible from a JavaScript interface. We were able to manually confirm all alarms as true positives. Interface methods accessible when creating an account creation within the game include `getAccountEmail`, `getPhoneNumber`, and `getUserPwd`. We successfully performed a man-in-the-middle attack and injected JavaScript to access all three methods. The account e-email and phone number are accessible immediately upon attempting to create an account. The password is stored in a local database, cached in the preferences and accessible with the *loginPwd* key. When the user visits the account creation page a second time, the password can be stolen via the interface method.

The underlying problem is twofold and representative for many Webview vulnerabilities: first, the Webview loads data via an insecure channel, and second, the JavaScript interface makes sensitive data available (a plaintext password). Even if the password would otherwise not be sent via the insecure channel, a JavaScript injection attack is able to retrieve it through the interface and extract it directly. Since our discovery, all issues have been resolved in a newer version of the application (version 2.6).

4.2.3 Ads Library InMobi

During the evaluation, we discovered an advertising library, used by 353 of 4,997 applications, which implements a Webview exposing many sensitive interface methods. In particular, a successful JavaScript injection would allow an attacker to perform different actions, including downloading/saving of pictures, sending email or SMS by manipulating `Intents`, playing audio or videos on the victim's phone, opening new applications, creating calendar events, and posting to social networks.

Another library, used by 1,507 applications, allows an attacker to start new applications on the phone, controlling the `Intent` extras provided to the `Activity`.

4.3 Work Outcome and Discussion

With BabelView, we introduced a new solution for the taint analysis of hybrid applications (Section 1.2). Specifically to the Webviews scene, one of the biggest challenges is the presence of web code (i.e., HTML and JavaScript), which is notoriously hard to analyze (Section 1.1.2). With our approach we successfully tackled this challenge by switching our focus on a generic threat model. With this approach we could effectively enable taint analysis for Android Webviews' JavaScript interfaces. However, our key goal was to perform a security analysis and evaluate the impact of an attack against these interfaces (Section 1.1.2). BabelView tackles this problem by classifying flows based on their semantic. Moreover, in the consolidation phase (Section 3.2.5), BabelView provides a more fine grained output on the nature of intents and preferences flows.

With this evaluation we proved that BabelView works in practice, and we shed light on the current state of WebView security in the Android echo system. The results shows that there is still confusion on how to safely integrate WebViews in Android applications. In fact, there are still instances where developers assume JavaScript interfaces as internal methods and they fail to understand the risks involved in exposing them. The problem can be even more severe if commonly used libraries implement potentially vulnerable WebViews. In our correlation analysis (Section 4.1.6), we showed and discussed how the alarm raised can be part of common libraries.

In our evaluation, we also found interesting case studies (Section 4.2), which showed once more the severity of the problem. While we could not provide a working exploit for all the case studies we showed, there is no doubt that the exposed interface can pose a threat to the end user if a JavaScript injection is successful. In all cases, we followed a responsible disclosure process. We did get in touch with the developers of the advertisement library. However, we reached a dead end and eventually could not get back any answer. For the game and the mobile banking cases, we saw that, in both cases, the authors updated their applications. We re-run BabelView on them and found out that in both cases the problems were resolved.

5 JniFuzzer: Fuzzing Android Java Native Interfaces

One of the biggest obstacles for the analysis of Android apps is Android's support for native components written in C and C++ to perform CPU-intensive tasks. Developers widely use this feature: over one-third of apps on Google Play include native code. Different libraries have been developed (e.g., advertisement libraries [88]), increasing the likelihood that exploits crafted against one app will be transferable to another app sharing the same vulnerability [89]. Given this wide adoption of native code and the risks associated with it, it is necessary to include native components in security analysis.

There is no defined limit to the size native code can assume within an Android application. The guideline on the official Android documentation [90] suggests that developer keeps the number of native code interfaces to a low number and to minimize marshalling of resurces across the native layer. However, these are only suggestions and developers are ultimately in charge of these aspects. In our experiments, we observed both applications with standalone and self contained native interfaces and with interfaces wrapping larger libraries. In both cases, external native library can be used, increasing the scope of analyses that would need to model these libraries.

The analysis of native code for Android apps is still challenging. The engineering overhead when building analyses for native code is remarkable, and not surprisingly, it is hard to reuse existing approaches to cover native components. For example, fuzzing (Section 2.4.2) has been successfully deployed for Android, from stimulating the UI [53, 55, 16, 19, 91, 54, 17, 20] to testing Intents [92, 93], and exposing vulnerabilities in Internet of Things (IoT) devices[52]. However, these tools rely on analyses not supporting native code, resulting in the inability to exhaustively test native code.

In this chapter, we propose JniFuzzer, a novel fuzz-testing framework targeted at Android apps' native components. To the best of our knowledge, there is no other work to directly enabling fuzzing for Java Native Interfaces. We designed JniFuzzer to be practical and extensible, and to support different analyses on native code. To this end, we implemented JniFuzzer as a plugin system, where new analyses can be easily integrated. In particular, in Section 5.1, we describe the implementation of the components at the core of JniFuzzer. In Section 5.2, we illustrate how we scaled and distributed our framework to support multiple and parallel analyses. In Section 5.3, we show how JniFuzzer can be successfully used on real world application. In Section 5.4, we discuss work related to ours. We then discuss limitations and future work in Section 5.5, and finally conclude with a discussion on the work outcomes in Section 5.6.

5.1 JniFuzzer

With JniFuzzer, our goal is to fuzz native methods. Our insight is that we do not need to stimulate the user interface, but we directly extract a function pointer to the fuzzing target.

In this section, we detail the implementation of JniFuzzer. In Section 5.1.1, we show how JniFuzzer creates mocks for the JNI Environment. In Section 5.1.2, we describe how JniFuzzer extracts a function pointer to the

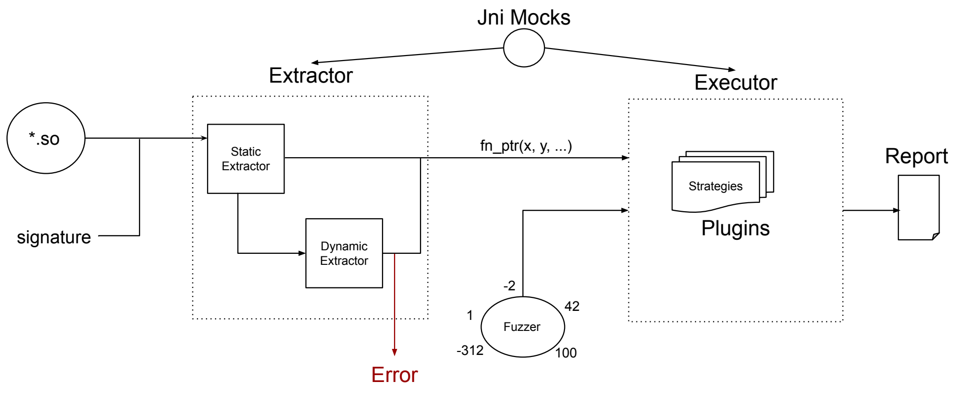


Figure 5.1: *JniFuzzer Design Diagram.* Given a signature and a library, the extractor extracts a function pointer of the target method. The function pointer is then executed with inputs generated by a fuzzer.

native method and, finally, in Section 5.1.3, we discuss how this pointer is executed.

5.1.1 Mocking The JNI Environment

We ultimately need to execute the target native method and therefore we must recreate a valid running environment. Recall, from Section 2.3.1, each native method receives a pointer to `JNIEnv`, which contains an interface pointer, which in turn, points to an array of function pointers. Then, we must recreate an instance of `JNIEnv` and mock all the functions exported and used by the native method.

The `jni.h` header defines approximately 240 functions[1]. In principle, one should mock all these functions to have a fully functional tool. Unfortunately, this would require a considerable effort given our limited resources. However, we note that one does not necessarily need to provide

[1]`http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/javavm/export/jni.h`

```
void GetMockedEnv(JNIEnv **env) {
    JNIEnv *my_env = (JNIEnv *) malloc (sizeof(JNIEnv));
    JNINativeInterface *env_funcs = (JNINativeInterface *)
        malloc (sizeof(JNINativeInterface));

    env_funcs->RegisterNatives = RegisterNatives_Mock;
    env_funcs->FindClass = FindClass_Mock;
    env_funcs->NewStringUTF = NewStringUTF_Mock;
    env_funcs->GetStringUTFChars = GetStringUTFChars_Mock;
    ...
    my_env->functions = env_funcs;
    *env = my_env;
}
```

Listing 5.1.1: Mocking the JNI Environment.

a mock for each of these functions. In fact, it is unlikely that an application uses more than a few of them. Therefore, we adopted an incremental and iterative approach to mock the functions more relevant to our analysis. We achieved this by reversing a few applications in our dataset and see what functions they were using. We started mocking these functions at first and then use our prototype on other applications and see where it failed. From there we iteratively mocked more functions.

To illustrate how we recreate a valid `JNIEnv`, consider `GetMockedEnv` in Listing 5.1.1. Here, we first allocate memory for a `JNIEnv*` and a `JNINativeInterface*`. The latter is the interface pointer, which is pointed to by the former. We recreate a custom implementation of all the functions pointed by the interface pointer, and finally, we let the `JNIEnv` point to it. Effectively, we hook each relevant API, overriding its behavior. Furthermore, certain methods require an instance of `JavaVm*` to run. `JavaVm*` has the same interface pointer structure as `JNIEnv*` and therefore we follow a similar approach to mock it.

To support basic string operation, we also mocked functions that we

```
class JString : public _jstring
{
   private:
   char *buff_;
   int size_;

   public:
   JString(char *str);
   JString();
   char *getString();
   int GetSize();
};
```

Listing 5.1.2: JString Mock Header

considered relevant such as (but not limited to) `NewStringUTF` and `GetStringUTFChars`. Since strings are a complex type (`jstring` in `jni.h`), we also provide a mock implementation of it (Listing 5.1.2). In particular, we define the `JString` class as a subclass of `jstring`. Finally, `JString` wraps a fixed size buffer of chars that can be accessed via a getter.

As we mentioned, we do not provide a full mock of `jni.h`. However, our prototype provides enough mocks to show its effectiveness (Section 5.3).

5.1.2 Function Pointer Extraction

As discussed in Section 2.3.1, there are two ways by which developers can register native methods. In the first scenario, Java signatures are matched on the native side. Consider a native method `example.MyActivity.nativeM(int x)` with return type **`int`**. This corresponds to the following C++ signature: `intJava_example_MyActivity_nativeM(JNIEnv *,jobject, int)`. Provided a signature in Java format, we parse and translate it into its native equivalent. The native signature can be used to extract a pointer

```
static JNINativeMethod methods[] = {
    {"getMessage", "()Ljava/lang/String", (void*)
        NativeMessage},
};

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = ...;

    jclass my_clazz = env->FindClass("example.MyActivity");
    env->RegisterNatives(myclazz, methods, 1);
    ...
}
```

Listing 5.1.3: Statically Linked Native Methods

from the native libraries in the APK.

Unfortunately, this approach does not work when developers statically map each JNI signature. They can do so by overriding `JNI_OnLoad` and calling the `RegisterNatives` API, registering an array of `JNINativeMethod`. Recall from Section 2.3.1 that `JNINativeMethod` is a structure containing the name of the method to bind as declared in Java, its (partial) signature, and a function pointer to the respective C/C++ implementation. Consider Listing 2.3.2, which we report in Listing 5.1.3 for ease of reference. Our goal is to extract `NativeMessage`. To this end, we provide a mock implementation of `RegisterNatives`, which iterates through the array of `JNINativeMethod` and saves them into a list we control. We then trigger its execution by invoking `JNI_OnLoad`. First, we lookup its symbol across the native libraries and extract a function pointer to it. Second, we generate valid inputs and execute it.

5.1.3 Execute Target Native Method

The goal of the executor is to invoke the native method, providing a valid input to it. To do so, it follows these steps:

1. It obtains a valid function pointer from the extractor.
2. It reads the input generated by a fuzzer.
3. It converts and passes the input to the native method.

We implemented the executor as an abstract class, enabling analysts to develop their analyses as plugins. Consider Listing 5.1.4; here, we show the header and a pseudo implementation of the base executor class. In particular, the abstract method `ReadFuzzerInput` must implement the logic to read the input, which is processed by the other abstract method `Execute` to run the native method. Finally, `Run` is a method implemented in the base class, which starts the execution chain. Consider now Listing 5.1.5, showing a pseudo implementation of a simple fuzzing strategy plugin. In this example, we implement `ReadFuzzerInput` to read from `stdin`. In `Execute`, we partition the input, convert it to respect the signature as provided in `parameter_types`, and finally, we execute the native method. Please note that the plugin should not need to implement `Run`, as its implementation is given in the base executor class. The advantage of this implementation is that we can use any fuzzer, as soon as it writes to `stdin`. In our prototype, we use AFL [23].

5.2 A distributed Fuzzing Framework

One of the gap in Android analysis is the lack of native code support. Existing tools either ignore it or require a great effort to function. With JniFuzzer, we introduced a new approach to effectively and specifically

```
// Header
class Executor {
   public:
      Executor();
      virtual char* ReadFuzzerInput() = 0;
      virtual bool Execute(JNINativeMethod method,
                     std::vector<std::string>
                        parameter_types,
                     JNIEnv* env,
                     char* input) = 0;
      void Run(std::string& signature, std::vector<std::
         string>& libraries);
};

// implementation pseudo code
void Executor::run(std::string& signature,
              std::vector<std::string>& libraries) {

   JNINativeMethod method = extractSignatureOrThrow(
      signature, method);

   std::vector<std::string> parameter_types =
      GetParameterTypes();

   char* buff = readFuzzerInput();
   execute(method, parameter_types, buff);
}
```

Listing 5.1.4: Executor Interface

```
// header
class SimpleExecutor : public Executor { ... }

// implementation
char* SimpleExecutor::readFuzzerInput() {
   return stdin.read();
}

bool SimpleExecutor::execute(JNINativeMethod method,
                      std::vector<std::string>
                         parameter_types,
                      JNIEnv* env,
                      char* input) {
   void* parameters[parameter_types.size()];
   for(int i=0; i<parameter_types.size(); i++) {
      parameters[i] = GetAndConvertNextArg(&input,
         parameter_types[i]);
   }

   call(parameter_types.size(), method.fnPtr, env, NULL,
      parameters);
}
```

Listing 5.1.5: Simple Fuzzing Strategy.

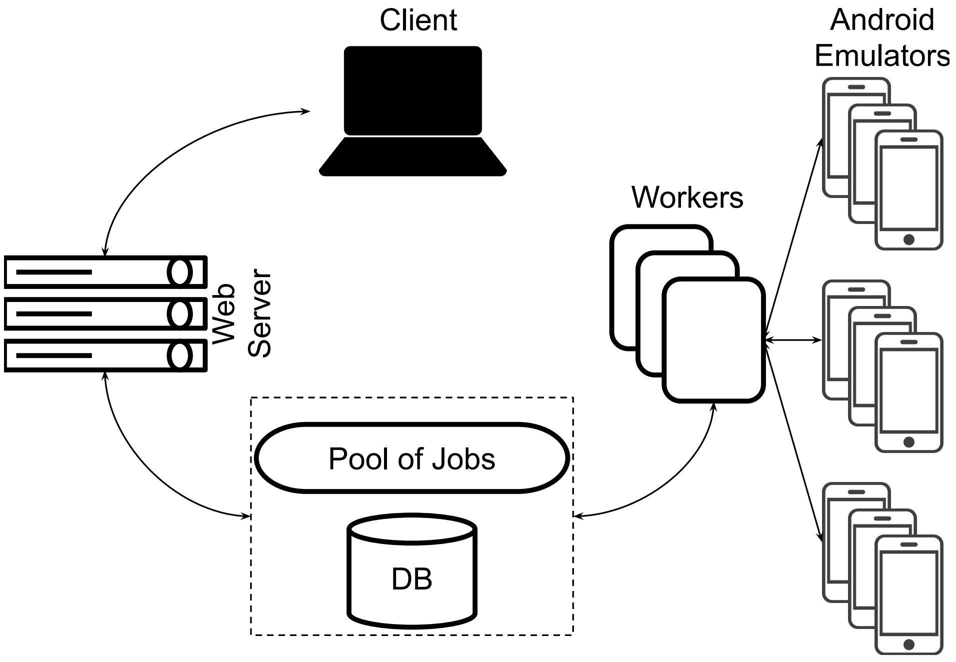


Figure 5.2: *Distributed JniFuzzer Framework.* An analyst initiate an analysis that a pool of workers concurrently perform.

fuzz Android Java native interface. We also wanted a tool that is easy to use and extend and developed a distributed version of JniFuzzer.

There are several benefits of having a distributed framework. For example, we successfully executed multiple parallel analysis across different servers. More importantly, such a framework could be deployed to help analysts and researchers to run analyses on Android applications without the burden of setting up their local environments. Furthermore, this could become a central repository of analyses to consult when needed.

In Figure 5.2 we show the main components of the JniFuzzer framework. The client instantiates a request for fuzzing to the webserver. The webserver processes and pushes it to the database, which acts as a pool of

Attribute	Description
signature	The signature of the native method to test.
apk_id	The id of the APK to extract the native method from.
timeout	Length of the analysis.
isa	The architecture for which the native library is compiled.
status	It indicates whether the job is available, in progress or completed.
node	Which node (server) is performing the analysis.
result_path	Path where to find the result.

Table 5.1: Job Description by Attribute

jobs. A pool of workers concurrently processes each available job, delivering it to a controlled pool of Android emulators. The emulators execute the analysis, whose results are then retrieved and stored by the workers and, finally, delivered to the user by the webserver.

In this section, we describe the details of the main components of Figure 5.2. We start describing the role of the database and the design of a job. We then describe the role of the client and of the webserver. Finally, we describe the functioning of the workers.

Database and Job Description. While we also use the database to store the framework users' credential, its purpose is to act as a pool of jobs. Each job contains the necessary information for the workers to execute an analysis. In Table 5.1, we provide a detailed description of the relevant sections of a job.

Client and Webserver. Analysts, who wish to use JniFuzzer, can test a native method via a web client. One can select the application to be analyzed, download it and visualize a summary of it (Figure 5.3). Moreover, one can tag an interesting APK and leave notes on it. The flow tab can be used to list all the flows that a taint analysis reported. If a method in the

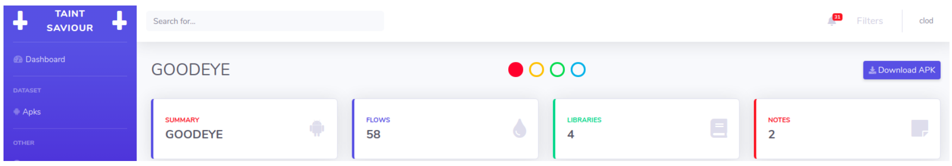
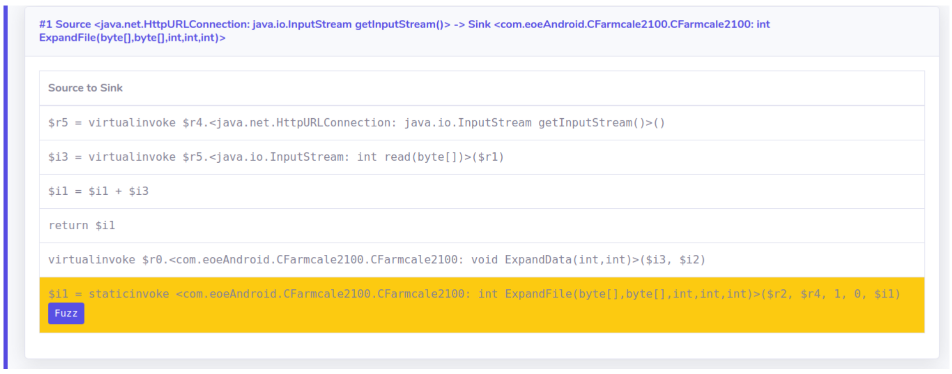


Figure 5.3: *Apk View.* An analyst can download and visualize a summary of a select application.



#1 Source <java.net.HttpURLConnection: java.io.InputStream getInputStream()> -> Sink <com.eoeAndroid.CFarmcale2100.CFarmcale2100: int ExpandFile(byte[],byte[],int,int,int)>

Source to Sink

```
$r5 = virtualinvoke $r4.<java.net.HttpURLConnection: java.io.InputStream getInputStream()>()
$i3 = virtualinvoke $r5.<java.io.InputStream: int read(byte[])>($r1)
$i1 = $i1 + $i3
return $i1
virtualinvoke $r0.<com.eoeAndroid.CFarmcale2100.CFarmcale2100: void ExpandData(int,int)>($i3, $i2)
$i1 = staticinvoke <com.eoeAndroid.CFarmcale2100.CFarmcale2100: int ExpandFile(byte[],byte[],int,int,int)>($r2, $r4, 1, 0, $i1)
```

Figure 5.4: *Flow View.* An analyst can visualize a taint analysis and select native flow to fuzz.

flow happens to be native, the UI highlights it, and the analyst is given the possibility to start a fuzzing session on it (Figure 5.4). When the fuzzing session is over, the user is notified and given the possibility to download the results.

All the requests are initialized by the client and directed to the webserver which exposes a set of REST APIs.

Workers and Emulators. The analysis ultimately happens on an Android emulator, which is controlled by a worker. We designed our framework as a distributed pool of workers. Each worker controls several emulators and lives on different nodes. Workers concurrently look up available jobs, which execute in the first available and compatible emulator for as long as specified by the timeout.

The analysis proceeds as follow. First, the worker creates the analysis environment, which consists of JniFuzzer, the native library along with the signature of the native method to test, and the fuzzer. In our implementation, we use AFL as our fuzzer[2]. Once the environment is deployed, the worker is ready to instruct AFL to fuzz JniFuzzer. Because of the lack of source code, we use AFL in dummy mode – i.e., AFL does not instrument the target function. In this settings, AFL has no information on the shape of the fuzzed method, and generates completely random inputs entirely based on the provided seeds. The worker is responsible for generating these seeds. They change according to the type of parameter of the tested method. For example, if the parameter is an integer, we generate at least three seeds consisting of zero, a positive and a negative random number, so that AFL generates input in those intervals. We recall that JniFuzzer wraps the execution of the native method under test, forwarding AFL generated inputs to it. At this point, the analysis continues for a specified timeout. When this timeout occurs, the analysis is over. The worker extracts the results from the emulator, stores them in the file system and maps them to the database. Finally, the job is marked as completed, and the analyst is notified.

5.3 Fuzzing Android JNI: Evaluation of Case Studies

In this section, we present our evaluation of JniFuzzer. We first detail our methodology (Section 5.3.1) and then we discuss how we used JniFuzzer to find potential vulnerabilities in real world applications (Section 5.3.2).

[2]We followed the steps in `https://github.com/ele7enxxh/android-afl` to have an Android working version of AFL.

5.3.1 Methodology

We obtained our data set from AndroZoo [84], selecting APKs observed in the Google Play store in 2017. We then selected applications with at least one native method – i.e., applications containing at least on native library. This resulted in a data set of 29,547.

We then selected which JNI functions to fuzz. The most interesting native methods are those reachable by external inputs from Java. Indeed, finding a bug in a native method has a higher impact if it can be triggered by a normal user. To this end, we performed a data flow analysis using FlowDroid and looking for flows to the JNI functions from user/external input sources. Note that JniFuzzer works independently from the data flow analysis, which is only used to retrieve interesting targets to fuzz. This step resulted in 4,171 APKs.

We ran JniFuzzer on different Android emulators, depending on the ISA of the library. Each emulator was running Android Marshmallow with 2 CPUs and 2GB of RAM. The emulators were deployed on two servers: one 32-core with 250GB of RAM and one 16-core with 125GB of RAM.

Our prototype does not implement the whole `jni.h` header and cannot handle native methods which uses an API that we did not mock. Currently, we only provide support for basic primitive types and partial support for strings. Unfortunately, this limits the number native methods that we can currently test. Neverthless, we were able to test 69 native methods: 68 with integer-only signatures and 1 with an integer-string signature. These native methods were found across a total of 34 applications[3] (See Table 5.2 for a breakdown).

[3]Not all the native method of each application were fuzzed, but only those one that our framework could handle.

#Native Method Fuzzed	#Apk Analysed	#Fuzzing Timeout	#Crashes
69	34	30 mins	3

Table 5.2: We report the raw numbers of our analysis. We could successfully analyse a total of 69 native methods we found in 34 different APKs. Each native method was fuzzed for 30 minutes and 3 of them crashed.

5.3.2 Analysis Results

We report the results of our analysis in Table 5.2. We successfully analysed 69 native methods from 32 different applications. We successfully fuzzed each native method for 30 minutes. In these 30 minutes, we could verify a total of 3 crashes. We then closely examined each crashed native method. First, we downloaded the analysis report. This report contains all the AFL information about the crash, as well as a copy of the native library containing the method being fuzzed. We then used a reverse engineering tool (e.g., IDA or Ghidra) to look at the decompiled source code of the library and we inspected the tested native method. In all cases we confirmed the presence of a potential bug with security implication.

In the following paragraphs, we provide a detailed overview of our findings.

Underconstrained Read. You can find the code of this native method in Listing 5.3.1. The problem of this function can be seen in Line 6. Here `input` is used as an index for `jobArr`. However, there are no checks on the boundary that can be indexed, exposing this native method to a bug which can result in a security vulnerability. In particular, by controlling `input`, an attacker can explore the program memory space looking for valid pointers. Providing `input` so that `buf` and `buf[4]` are valid pointers, `result` is set to the value of `buf[4]` at index 416. The result is finally returned, potentially containing sensitive values.

```
int ff_getRawRate(JNIEnv *env, jobj job, int input){
   int result;
   _DWORD *buf;

   doInfo("=== ff_getRawRate");
   buf = &jobArr[542 * input];
   if (buf[4])
      result = *(buf[4]+ 416);
   else
      result = 0;
   return result;
}
```

Listing 5.3.1: Arbitrary Read

```
1  int ff_setStdRate(JNIEnv *env, jobj job, int in1, int in2){
2     doInfo("=== ff_setStdRate");
3     jobArr[542 * in1 + 8] = in2;
4     return 0;
5  }
```

Listing 5.3.2: Arbitrary Write

Arbitrary Write. Consider the native method in Listing 5.3.2. Here the bug is in Line 3, where `jobArr` is accessed without any check. In this case, an attacker can use `in1` to select a memory area and write `in2` to it. This has security implications as a malicious user could take control over the program execution.

Arbitrary Read. Another bug allowing reading arbitrary memory locations is shown in Listing 5.3.3. This JNI method is part of a calendar library and it is used to get the number of days in a specified month. The days are stored in `solarcal`, which maps the month to its number of days. The bug is in Line 12, where `solarcal` is accessed with an unconstrained

```
int GetMonthDay(JNIEnv *env, jobj job, int year, int month) {
   int result;
   int leap;

   if ( year == 1582 && month == 10 )
      return 20;

   leap = getleap(in1);
   if ( month == 2 )
      result = leap + 28;
   else
      result = solarcal[month - 1];
   return result;
}
```

Listing 5.3.3: Arbitrary Read 2

index `month`. This effectively enables a memory leak, which could have security implications.

We further investigate to see if we were able to trigger the bug directly from the Android application. We manually reversed the APK and with the help of our previous flow analysis, we identified the call site for this method. We verified that there is a check preventing `month` to be greater than 12. However, developers forgot to check for negative indices.

Buffer Overflow. Listing 5.3.4 shows another interesting bug. The bug is spread across multiple lines (Line 6, Line 10 and Line 14). Again, the buffers are accessed without checking the boundaries. In particular, controlling `in2`, an attacker can read arbitrary memory locations. Moreover, note that `strcpy` is used to copy the bytes at index `in2` of another buffer into `buf`. This could lead to a buffer overflow. For instance, consider Line 14; if `YearBornStr1[in2]` is a controlled memory location, `buf` can be overflown leading to stack-based buffer overflow.

```
jstring LunarStr(JNIEnv *env, jobj job, int in1, int in2){
   char* buf[0x14];
   if ( in1 ) {
      switch ( in1 ) {
         case 1:
            strcpy(buf, YearBornStr2[in2]);
            break;
         ...
         case 33:
            strcpy(buf, KingList[in2]);
            break;
      }
   } else {
      strcpy(buf, YearBornStr1[in2]);
   }
   return env->NewStringUTF(env, buf);
}
```

Listing 5.3.4: Buffer Overflow

Discussion and Future Work. In this evaluation, we showed that JniFuzzer can successfully find potential vulnerabilities in real world applications. However, JniFuzzer cannot automatically verify that the vulnerabilities are exploitable from Java (Section 5.5).

An interesting research direction consists of filling the gap between native code and Java. In particular, it would be valuable to explore whether the native method can be executed from Java with the input received from the fuzzer.

5.4 Related Work

There have been numerous applications of fuzz testing the Android OS and Android apps. AppsPlayground [54] fuzzes Android apps via intelligent GUI exploration. IntentFuzzer [92] and DroidFuzzer [93] specifically target intents and intent-filters, respectively. Buzzer [94] targets the Binder protocol to find vulnerabilities in Android system services. While IoT-Fuzzer [52] aims to find memory corruption vulnerabilities in IoT devices, it relies on protocol information extracted from the devices' companion Android apps. Fuzzing tools that are guided intelligently rely on information from other analysis tools – e.g, IntentFuzzer relies on FlowDroid while AppsPlayground and IoTFuzzer rely on TaintDroid. While some of the tools above will detect faults emanating from native components, they are likely to ignore important paths that flow through native components due to incomplete information from the tools they are built on. We compliment these approaches by specifically testing native components, ensuring that no important libraries are missed.

5.5 Limitations and Future Work

In this section, we present the limitations of JniFuzzer and we discuss possible future direction to our line of work.

Stateful JNI Functions. In some cases, executing a JNI method in isolation is ineffective. For example, some method may return early with an error code if certain data structures are not correctly initialized. Here, JniFuzzer will fail to explore other paths. A potential solution is to extract a backward slice from the application call graph starting at the target JNI method. One can then execute the Java method contained in that slice, in an attempt to simulate a realistic execution flow. This execution should result in the correct context being present during the fuzzing of the target method.

Trigger Bugs From Application Level. While JniFuzzer can effectively find bugs in JNI methods, we are not able to exhaustively confirm whether a crash can be triggered from the Java environment. We rely on a preliminary flow analysis to highlight flows sinking to the tested method, which suggest that a bug could be triggered from external input. However, this is not sufficient as taint analysis can introduce false positives.

A future direction for this research is to extend JniFuzzer to automatically trigger the crash in the application whenever possible. To this end, one can use a combination of symbolic execution and program slicing, similar to Intellidroid [95], to confirm that the generated input can reach the JNI method.

JNI Environment Models. JNI exposes APIs via a `JNIEnv` object. Successful execution of a JNI function requires mocking these APIs as thoroughly as possible. Android implementation of JNI comes as part of the Android Runtime. Therefore, the mocks need to be as close as possible to the original implementation. Currently, our PoC implementation of JniFuzzer only supports simple interactions with `JNIEnv`, which prevents us from finding bugs involving heavy use of `JNIEnv` APIs.

To build on our work, one could extend JniFuzzer's mocked environment by automatically generating stubs from the API implementations in the Android source code.

Fuzzing Limitation. Due to lack of source code, we are forced to adopt a black-box approach which prohibits us from taking advantage of powerful gray-box fuzzers such as AFL [23]. In fact, we could not use AFL's QEMU mode as we were running our analysis on Android emulators. JniFuzzer currently integrates AFL in "dummy mode" – i.e., generating random inputs without guidance. Future extensions could mitigate this limitation implementing the same approach as afl-dyninst [96], where binary instrumentation is used to instruct AFL. This would enable gray-box fuzzing which would increase code coverage.

While a gray-box approach might be more effective at exposing dangerous bugs, a blackbox one might be more appropriate for certain use cases, such as dependency analysis. In fact, in this scenario, rather than applying a gray-box fuzzing approach, one might as well use proper static analysis dependance determination with data flow analysis. However, these techniques are heavyweight as opposed to the simplicity of a blackbox fuzzer. Moreover, we note that JNI methods are typically small and, therefore, a simple blackbox strategy might be enough to show data dependency. We further explore the idea of detecting data dependency in Chapter 6.

Another limitation is that JniFuzzer only supports a limited set of types, i.e., primitive numerical types and strings. Supporting more generic types, such as objects, is challenging, but could be complemented along the lines of Darko Marinov's tool Korat [97].

5.6 Work Outcome and Discussion

The analysis of native code is still one of the biggest challenges for the analysis of Android applications (Section 1.1.3). The goal of JniFuzzer was to provide a tool that could ease the effort of analyzing native code and expose security bug lurking in there (Section 1.2). We developed a new approach that directly targets Android JNI, applying a black-box fuzzing strategy to methodically exercise JNI functions (Section 5.1). One of the advantages of a black-box is that JniFuzzer does not require the application source code, which is usually not available to the analyst (Section 1.1.3). Another challenge was to provide an easy to use tool that is effective in practice. To this end, we developed a whole framework around JniFuzzer that provides an easy to use interface for the tool (Section 5.2). Not only this provides an easy to use interface, but it reduce the time of the analysis.

While JniFuzzer is still a proof of concept, we could assess its efficacy in our evaluation (Section 5.3). JniFuzzer found 3 possible security bugs in 3 different applications, that we could easily analyze and verify thanks to the web interface. While JniFuzzer is still a work in progress, we think that can give room to further researchers to refine the approach and extend it as we discuss in Section 5.5.

6 TaintSaviour

Existing state-of-the-art taint trackers have limited to no support for native code. The native interfaces present a blind spot for taint trackers, who are unaware of how to propagate the taint upon the invocation of these interfaces. The underlying problem is that these tools only reason about one language, Java, and the interfaces reference code written in C/C++. This problem is similar to that of dynamically loaded code, where the analysis cannot proceed because that code is not included in the app.

Recent research has focused on solutions to this challenge. Artz S. and Bodden E., developed StubDroid [98], a system based on taint analysis to automatically generate summaries for Android framework methods. Unfortunately, the underlying taint analysis does not consider the native code, which is once more left out. An approach developed independently and at the same time as ours, which targets JNI directly, is JN-SAF [99]. JN-SAF uses symbolic execution to generate summaries of native library methods inheriting all its limitations, and failing to model larger libraries.

In contrast to previous works, we propose a new dynamic approach to generate summaries for Android JNI methods. To this end, we implemented TaintSaviour, proof of concept built on JniFuzzer that uses blackbox fuzzing of methods to produce a summary. We then support our idea with a preliminary evaluation, showing that TaintSaviour can work in practice.

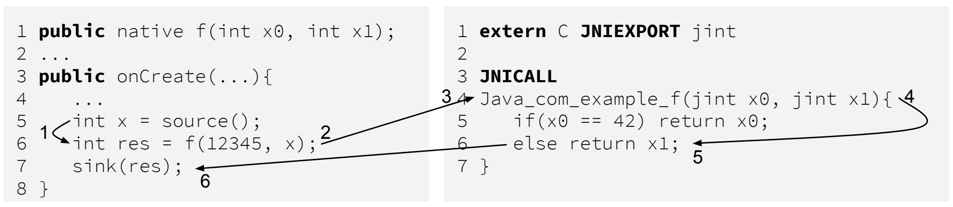
This chapter is organized as follows. In Section 6.1, we start introducing our approach. In Section 6.2, we detail TaintSaviour's implementation and, in Section 6.3, we present our preliminary evaluation of TaintSaviour. In Section 6.4, we then discuss work related to ours and, in Section 6.5, we discuss limitations of TaintSaviour and future directions for this research. Finally, we conclude with a discussion of the work outcome in Section 6.6.

6.1 Data-flow Analysis for Android JNI

In this section we introduce our approach to summary generation for Android JNI. We first introduce the problem we are trying to solve (Section 6.1.1). We then describe techniques to discover Input/Output dependencies of a program (Section 6.1.2) and we conclude detailing the core of our approach (Section 6.1.3).

6.1.1 The Need for Android JNI Summaries

Android JNIs are defined as stubs in Java and then implemented in the C/C++ counterpart. An APK with native code includes a shared library containing JNIs implementation. A data-flow analysis needs to have the capability to switch context between Java and C/C++ or it would miss relevant flows. For example, consider Figure 6.1. The variable `x` gets tainted on Line 5 of the Java snippet. It flows to the native method `f`, which returns it unmodified. This return value finally reaches the sink in Line 7. A data-flow analysis only operating on Java would miss the flow, as it has no information on how the taint should propagate in the native code. Therefore, a taint-tracker needs a model instructing it what to do with the taint.



```
public native f(int x0, int x1);
...
public onCreate(...){
   ...
   int x = source();
   int res = f(12345, x);
   sink(res);
}
```

```
extern C JNIEXPORT jint

JNICALL
Java_com_example_f(jint x0, jint x1){
   if(x0 == 42) return x0;
   else return x1;
}
```

Figure 6.1: *Flow through JNI.* A variable is tainted in the Java snippet (on the left) and propagates to native code (on the right). The same variable is then return back to Java, where it ends up in a sink.

6.1.2 Input/Output Dependency Analysis

Understanding the relationships between input and output variables of a program can be done in different ways [100]. One way relies on manual analysis. In this scenario, an analyst needs to read the program documentation or ask the program developers for insight. Unfortunately, this does not scale and any manual approach inevitably reduces the effectiveness of the analysis.

It is possible to derive Input/Output dependencies automatically. In particular, a mix of static and dynamic analysis approaches can be used.

Static Approaches. Static approaches construct a program dependency graph to determine which inputs influence a program output for each input [101, 102]. Their advantage is that they can exhaustively retrieve all the dependencies. However, they generally over-approximate the dependencies, introducing false positives.

Dynamic Approaches. White-box dynamic approaches can execute the program and maintain traces from which Input/Output relationships can be derived [103, 104]. When the source code is not available, black-box fuzzing approaches can determine Input/Output relationships by varying only a single input at a time and observing changes in the output. The advantage of dynamic analysis is that the relationships found are real. However, it is generally unfeasible to find all of them as exhaustively testing all plausible inputs is not possible.

6.1.3 A Black-box Approach

Our final goal is to generate summaries of methods that data-flow analysis can use to enhance its precision. Differently from other works [98, 99], our approach consists of executing the target method and using Input/Output dependencies to model taint propagation. If we see a method as a black-box routine, by observing how the inputs affect the outputs, we can decide whether and how to remove or propagate a taint. We assume pure functions and we introduce the following definitions:

Definition 7. We define the pure function $f : I^* \rightarrow O^*$ as a deterministic function, where I^* is the input space and O^* the output space.

Definition 8 (input)**.** We define input as the vector $I = (i_0, i_1, \dots, i_n)$ of variables that may affect a method execution. This includes anything that the method has access to (e.g., global variables, file system, etc.).

Definition 9 (output)**.** We define output of a pure function $f : I^* \rightarrow O^*$ as the vector $O = (o_0, o_1, \dots, o_m)$ of outcomes a given method produces given an input. This includes anything that the method can modify (e.g., global variables, file system, etc.).

Definition 10 (Dependency)**.** Given a pure function $f : I^* \rightarrow O^*$ and two inputs, $I = (i_0, \dots, i_j, \dots, i_n)$ and $I' = (i'_0, \dots, i'_j, \dots, i'_n)$ both $\in I^*$, such that $i_z = i'_z$ for $z \neq j$, and $i_j \neq i'_j$, then $\exists k$ such that if:

$$f(I) = (o_0, \dots, o_k, \dots, o_m) \wedge f(I') = (o'_0, \dots, o'_k, \dots, o'_m) \wedge o_k \neq o'_k$$

then we say that o_k depends on i_j.

We recognize four relevant scenarios:

1. **Taint Propagation**: the taint propagates from an input to an output.
2. **Taint Killing**: the taint stops propagating through the code.
3. **Native Code Sink**: a sink is found in native code.
4. **Native Code Source**: a source is found in native code.

Taint Propagation. Consider the following example:

```
int sum(int x, int y) {
 return x+y;
}
```

Here, the input I is composed of `x` and `y`, while the output O of the return value (r) of the method. If we execute the method whilst fuzzing the input and we observe the output, we can see there is a dependency between them. Indeed, if we fix y and fuzz x, we see that x keeps changing, suggesting r depends on it. Similarly, varying y and fixing x yields a dependency between y and r. Therefore, we can derive the following summary, where $T(x)$ means that x is tainted:

$$T(x) \Rightarrow T(r)$$
$$T(y) \Rightarrow T(r)$$

This summary replaces `sum` entirely with respect to the semantics of taint propagation. An analysis can then query the summary to know how to propagate the taint.

Taint Killing. Consider the following example:

```
int sum(int x, int y) {
  x = 42;
  return x+y;
}
```

In this case, one of the inputs, x, is overridden by a constant. Therefore, if we fix the value of the other input, y, and fuzz x, the output is never going to change. The input y and the output are dependent, hence we can derive the following summary:

$$T(x) \Rightarrow r$$
$$T(y) \Rightarrow T(r)$$

In the first rule, x is tainted implies that r is not, effectively killing the taint.

Native Code Sink. There can be situations where a sink lies in native code. Consider the following code:

```
int sum(x, y) {
  log(x);
  return x+y;
}
```

This code uses the function `log` to log the value of `x`. If `x` is tainted and `log` is a sink, then we should be able to detect a flow. We can still use our black-box approach for these situations. Indeed, if we consider the sink `log` as part of the outputs, we can see if the input affects it. We can achieve this by monitoring calls to this function, looking for any dependency between the inputs and any parameter of that function. For instance, the input consists of `x` and `y` and the output of the return value (r) and the sink `log` (sk_0). Tracing the execution of `sum`, we observe that variation of x reflects consistent variations of how `log` is invoked. We can then observe a dependency, meaning a flow is found. Therefore, the following model can be derived, where $Flow(x, m)$ means that there is a flow from x to the sink m:

$$T(x) \Rightarrow Flow(x, sk_0)$$
$$T(x) \Rightarrow T(r)$$
$$T(y) \Rightarrow T(r)$$

In this case, tainting x has two effects. A flow to `log` is detected, and the taint propagates to the returning value.

Native Code Source. In this case, we consider taints generated within the native code. For example, examine the following code:

```
int sum(x, y) {
  z = source();
  return x+y+z;
}
```

The variable z comes from a source. Therefore, we expect the return value to be tainted if z is tainted. Considering the source method as one of the inputs (sr_0) for the black-box analysis, we can see if it affects any output. In particular, we can hook `source` and override it to produce random values that we control. In the example, we can see that by fixing x and y, different results produced by `source` reflect in dependency with the output. Therefore, we can deduce the following model:

$$sr_0 \Rightarrow T(r)$$
$$T(x) \Rightarrow T(r)$$
$$T(y) \Rightarrow T(r)$$

Here, the first rule generates a taint if a dependency is found between `source` and the returning value.

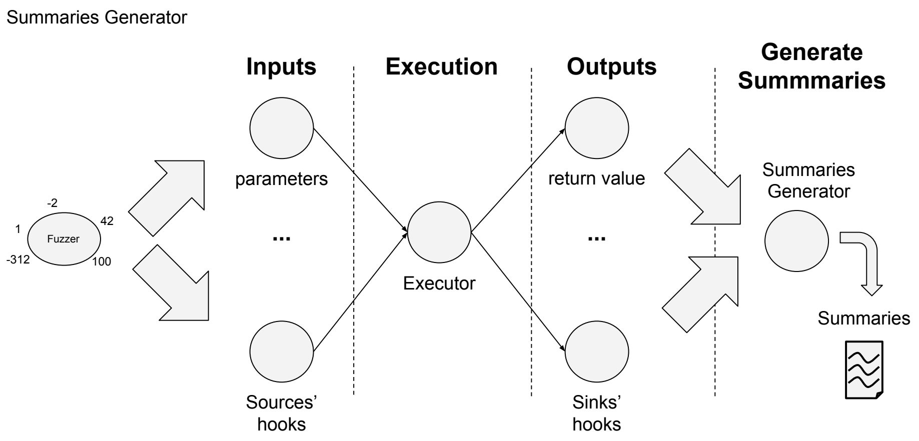


Figure 6.2: *TaintSaviour system overview.* In Phase 1, the inputs are read and converted from a fuzzer. In Phase 2, the native method is executed and the output are monitored. Finally, in Phase 3, a summary is generated.

6.2 Implementation

We built TaintSaviour as a plugin for JniFuzzer (see Chapter 5). As shown in Figure 6.2, TaintSaviour works in three different phases:

1. **Phase 1**, read the fuzzing value and convert it to fit the inputs.
2. **Phase 2**, execute the native method, monitor the outputs, and generate an Input/Output trace.
3. **Phase 3**, generate a summary based on the Input/Output trace.

In this section, we illustrate the implementation of these phases. In Section 6.2.1, we discuss how TaintSaviour reads the fuzzer generated values and uses them to fuzz the respective input (Phase 1). In Section 6.2.2, we detail how TaintSaviour executes the native method and monitors the outputs (Phase 2). In Section 6.2.3, we further detail our hooking system and how we use it to detect sources and sinks. Finally, in Section 6.2.4, we describe how TaintSaviour generates the summary.

6.2.1 Phase 1: Getting Values for the Inputs

TaintSaviour is a plugin of JniFuzzer and therefore it inherits its modularity benefits, enabling us to implement different fuzzing strategies. To fuzz numerical methods, we developed two strategies that generate values from a normal and uniform distribution. To handle types other than numerical (e.g., strings), we rely on AFL [23].

TaintSaviour reads values from a fuzzer and converts and delivers them to the chosen input. The fuzzer serializes the values as a byte stream, which TaintSaviour reads and converts to the type of the input being tested. Consider, for example, a fuzzer that generates integer numbers and an integer input. Being aware of the native method signature, TaintSaviour reads the byte stream and uses a function (e.g., `atoi`) to convert it to the proper type. Finally, we log the converted input as part of the Input/Output trace. Currently, TaintSaviour supports numerical primitive types and strings.

6.2.2 Phase 2: Execution and Output Monitoring

While TaintSaviour relies on JniFuzzer to execute the native methods, it needs to select the input to test and map it to each output. TaintSaviour assigns a key index to each input and output, and fuzzes the inputs, one by one, for a given time, while fixing the value for the other. During this process, TaintSaviour monitors the values of each output, matching them with the fuzzed input and generating a trace. A trace consists of a vector where at index zero, we store the index of the fuzzed input and at index $i + 1$, we store the values of the i_{th} input or output. For example, consider the following native method, where `puts` is a sink:

```
jint log_native(jstring str) {
 puts(str);
 return 1;
}
```

In this case, the input consists of the method parameter while the output of the return value and the sink. A possible trace looks like:

$$[\mathsf{fuzzed}_{\mathsf{index}}, \mathsf{str}, \mathsf{ret}_{\mathsf{value}}, \mathsf{puts}] \to [0, \mathsf{HelloWorld}, 1, \mathsf{HelloWorld}]$$

The result of this phase is a list of traces that TaintSaviour then uses to generate a summary.

6.2.3 The Hooks System

TaintSaviour's summaries consider sources and sinks in native code. We treat sources and sinks as part of the input and output, respectively. To summarize native methods with sinks, we monitor the sink definitions, observing if the currently fuzzed input shows a dependency with the parameters, or any other inputs of the sink. We treat sources in an orthogonal way. We modify their behavior by integrating the fuzzer's produced values into their outputs, for example, modifying their return values.

To observe and modify source and sink methods, we developed a hooking system. We use library preloading to load a library containing the hooked functions. In the following two paragraphs, we show two examples of a source and a sink hook.

Source Hooks. A source hook modifies the outputs of the source method to be random while respecting its contract. For example, consider the example in Listing 6.2.1, where we show a C-like pseudo-code for the source `get_sensitive_integer`. The first step of the hook is to get a reference to the real implementation of the source (Line 4). As we mentioned earlier,

```
1  static int (*real_get_sensitive_integer)(int) = NULL;
2
3  int get_sensitive_integer(int value) {
4   real_get_sensitive_integer = auto get_real("
       get_sensitive_integer");
5
6   if(isInputFuzzed("get_sensitive_integer")){
7    return getFuzzerValue();
8   }
9
10  return real_get_sensitive_integer(value);
11 }
```

Listing 6.2.1: Source Hook Example.

we treat sources as if they were part of the input. Therefore, we need to check if the source is the input currently being fuzzed (Line 6). If it is, we then override its return value with the value from the fuzzer. Otherwise, we call the original implementation (Line 10).

Sink Hooks. A sink is part of the output in TaintSaviour analysis. A sink hook observes how the sink parameters (or any other input) vary under TaintSaviour execution. For example, consider Listing 6.2.2, where we show a hook for the sink `puts` from `libc`. In this example, we are interested in the sink parameter `message` (the only input of the `puts` function). First, the hook obtains the index key of the currently fuzzed input (Line 6). Second, we log a trace, for the input index, with `message` as one of the outputs (Line 7). Finally, we invoke the real implementation of `puts` (Line 9).

6.2.4 Phase 3: Summary Generation

TaintSaviour uses the Input/Output traces to generate summaries for dataflow analysis. Our goal is to find dependencies between input and output

```
static int (*real_puts)(const char *message) = NULL;

int puts(const char *message) {
  real_puts = auto get_real("puts");

  int input_index = getFuzzedIndex();
  log_trace(input_index, message)

  return real_puts(message);
}
```

Listing 6.2.2: Sink Hook Example.

```
jint nativeSink(jstring value) {
  puts(value); // sink
  return 1;
}
```

Listing 6.2.3: Native method with sink.

and provide rules for taint propagation, generation and suppression. We parse the traces, look for a dependency (recall from Definition 10) and produce a JSON summary. A dependency implies that given an input and an output, there are at least two distinct values of the input resulting in two distinct values of the output. Consider, for example, the native method in Listing 6.2.3. Suppose `puts` is a sink, this method's input and output are respectively: $I = (\mathtt{param}_0)$ and $O = (\mathtt{ret}_v, \mathtt{sink}_1)$, where $\mathtt{param}_0$ is the `value` parameter of `nativeSink`, $\mathtt{ret}_v$ is the return value, the constant 1, and $\mathtt{sink}_1$ is the method `puts`. In Table 6.1, where we report part of an execution trace for this method. Since there are at least two different values for the input $\mathtt{param}_0$, leading to different values for the output $\mathtt{sink}_1$, we have a dependency between them. On the other hand, there is no dependency between $\mathtt{param}_0$ and $\mathtt{ret}_v$. Because $\mathtt{sink}_1$ is a sink, the resulting

int nativeSink(String param_0)		
Inputs	**Outputs**	
param_0	ret_v	sink_1
durhfied495jd94fj	1	durhfied495jd94fj
durhfied49+jd94fj	1	durhfied49+jd94fj
durhfied493Zd94fj	1	durhfied493Zd94fj
...	...	...
du	1	du
durh	1	durh

Table 6.1: Execution trace of `nativeSink` execution. A dependency shows between param_0 and sink_1, indicating a possible flow.

```
{
 "int nativeSink(String param_0)": [{
    "input": "param_0",
    "output": "sink_1",
    "flow": true
 }]
}
```

Listing 6.2.4: JSON Model for nativeSink

model is:

$$\mathsf{T}(\mathsf{param}_0) \Rightarrow \mathsf{Flow}(\mathsf{param}_0, \mathsf{sink}_1)$$

This summary corresponds to the JSON in Listing 6.2.4. Each JSON entry is a model for a specific signature, which is used as a key to access the model. The signature entry consists of a list of Input/Output dependencies. If the boolean `flow` is true, it means that a flow is found, which is the case for `nativeSink`.

6.3 Testing and Preliminary Evaluation

In this section, we show preliminary results that demonstrate that TaintSaviour's approach is viable in practice. We start by showing and discussing how TaintSaviour can be used to model the math library from `libc` (Section 6.3.1). We then provide a preliminary evaluation of how our hook system works to model sources and sinks in native code (Section 6.3.2). We then integrate TaintSaviour with FlowDroid and discuss how the former can improve the results of the latter (Section 6.3.3). We finally show a case study of how we use TaintSaviour to model a JNI method of an app from the Google Play Store (Section 6.3.4).

6.3.1 Models for Mathematics Functions

In this section, we show how TaintSaviour can be used to generate models for the standard math library `math.h`.

We ask the following research questions:

1. Does our approach work in principle? We conduct a preliminary evaluation and consider stateless Input/Output functions. We verify that our system generates meaningful summaries in this scenario.
2. How do the results vary depending on the distribution of the input? We use different fuzzing strategies and compare the results.

Methodology. To test functions in the Math library in the context of JNI, we developed an Android application, wrapping math functions with native methods. We also introduce a new function, **`const`**, that given an integer always returns the same constant value. We report the signatures in Table 6.2. Apart from **`const`**, all functions in Table 6.2 have Input/Output dependency. Therefore, we expect TaintSaviour to generate summaries

Name	Return Type	Parameter Types	Name	Return Type	Parameter Types
acos	double	double	asin	double	double
atan	double	double	atan2	double	double, double
cos	double	double	cosh	double	double
sin	double	double	sinh	double	double
tanh	double	double	exp	double	double
log	double	double	log10	double	double
sqrt	double	double	fmod	double	double, double
floor	double	double	const	double	double

Table 6.2: Math library function signatures.

that propagate a taint for all the function with a dependency and that kill a taint for `const`.

We fuzz each function for 60 seconds using three input generation strategies: normal distribution, uniform distribution, and AFL [23]. We performed our analysis on a pool of `X86_64` Android emulators, running Android Marshmallow (android-23 API) with 1,024MBytes of RAM.

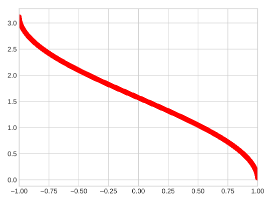
Input from a Normal Distribution. We generated inputs following a normal distribution with mean 0 and standard deviation 2. We chose these parameters to increase the probability of producing decimal numbers, both positive and negative, with values closer to 0 more likely to be generated. TaintSaviour could find all the expected dependencies and determined `const`'s return value to be input-independent.

Input from a Uniform Distribution. We generated inputs following a uniform distribution between -1,000 and 1,000. With this experiment, our goal was to evaluate whether TaintSaviour can detect the expected dependencies in a larger dataset. Once more, TaintSaviour generated the expected summaries.

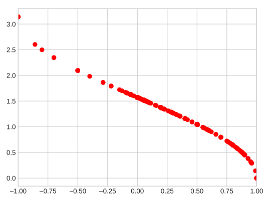
AFL-generated Input. For this experiment, we used AFL to generate the inputs. Our goal was to evaluate whether we can reuse state-of-the-art fuzzing strategies. TaintSaviour met the expectations in this scenario as well.

Discussion. In this evaluation, we used the `math.h` library, extended with **`const`**, as ground truth, showing that our approach is, in principle, effective. We expected to find a dependency for every function but **`const`**. We enumerated all inputs and outputs, respectively parameters and return values. As expected, the only summary without dependencies (i.e., the taint does not propagate) is the one for **`const`**. Every other summary, instead, showed a dependency – i.e., at least two different inputs are generating two different and consistent outputs. Precisely, if one parameter is tainted, then the return value is tainted too. This result showed that TaintSaviour could generate proper summaries, proving our approach is worth exploring further.

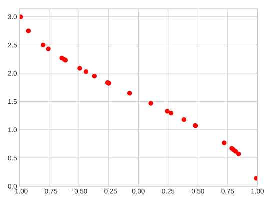
Although all the input generation strategies proved successful at finding Input/Output dependencies, we noticed a performance discrepancy. For instance, two of the functions we modeled are `acos` (Figure 6.3) and `cos` (Figure 6.4). First, consider `acos`; the domain of this function is defined in the interval of $(-1, 1)$. Amongst the three strategies, the normal input distribution is, not surprisingly, the best fit (Figure 6.3a). Indeed, the generated values concentrate around the distribution means which, in our case, was zero. The uniform distribution proved to be the worst choice (Figure 6.3c). The values spread on a larger interval and therefore, they are less representative. AFL comes in the middle (Figure 6.3b); the inputs gather around the zero, but the more we get closer to the boundaries, the more they scatter. This result happens because of the seed choice for AFL. AFL starts generating values around those seeds. In our case, we had zero, negative, and positive floating-point values (falling out of



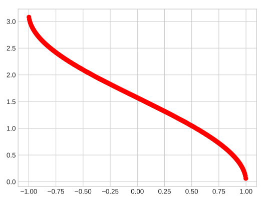
(a) `acos` normal distribution.



(b) `acos` AFL.

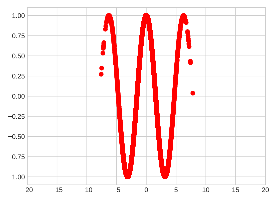


(c) `acos` uniform distribution.

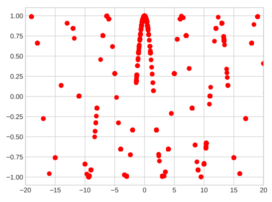


(d) `acos` function plot.

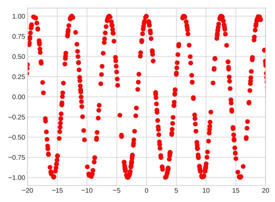
Figure 6.3: *Fuzzing Strategies for* `acos`. Fit of the `acos` function with input generated with normal, uniform and ALF strategies.



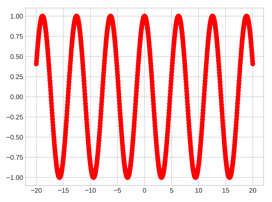
(a) `cos` normal distribution.



(b) `cos` AFL.



(c) `cos` uniform distribution.



(d) `cos` function plot.

Figure 6.4: *Fuzzing Strategies for* `cos`. Fit of the `cos` function with input generated with normal, uniform and ALF strategies.

the boundaries). Second, consider `cos`; this function is defined in $\mathbb{R}$. The normal distribution does fit the function in a specific interval but it fails to explore more (Figure 6.5a). By contrast, the uniform distribution proved to be the best choice, as it could explore a wider interval (Figure 6.5c). AFL comes, again, in the middle (Figure 6.5b).

Which strategy to choose depends on the specific case. If nothing at all is known about the function to test, or it deals with non-numerical inputs, then AFL is the best option. On the other hand, if the input is numerical and there is knowledge of the boundaries of the function, a uniform or normal strategy can be preferred. If the function domain is defined within a finite interval, then a normal distribution strategy should be used, while if the domain is relatively large (e.g., $(-\infty, +\infty)$), a uniform distribution is the best option.

6.3.2 Native Code Sources and Sinks

In this section, we show how TaintSaviour can generate models where sources and sinks are in native code. To this end, we ask the following research questions:

1. Can we use our approach to propagate taints generated in native code?
2. Can we use our approach to model taints reaching native code sinks?

Methodology. We developed an Android application and implemented two native methods, `nativeSource` and `nativeSink`, introducing a source and a sink, in native code, respectively. In Listing 6.3.1 and Listing 6.3.2, we show their implementation; in this example, we selected `puts` as a sink and `rand` as a source. We used TaintSaviour to model those methods and verified the summaries were meaningful.

```
void log_message(char *msg) {
  puts(msg);
}

extern C JNIEXPORT jint
JNICALL
Java_com_example_nativeSink(JNIEnv *env,
                            jobject thiz,
                            jstring str) {
  char *msg = env->GetStringUTFChars(str, false);
  log_message(msg);
  return 1;
}
```

Listing 6.3.1: Native Code Sink Method

```

int source() {
  return rand()
}

extern C JNIEXPORT jint
JNICALL
Java_com_example_nativeSource(JNIEnv *env,
                              jobject thiz,
                              jint value) {
  printf("received %d", value);
  return source();
}
```

Listing 6.3.2: Native Code Source Method

We fuzz the two methods for 60 seconds each, using AFL [23] to generate inputs. We performed our analysis on an `X86_64` Android emulator, running Android Marshmallow (android-23 API) with 1,024MBytes of RAM.

Native Code Sink. For this experiment, the input and output were respectively:

$$I = \{\mathsf{param}_0\}$$
$$O = \{\mathsf{ret}_v, \mathsf{sink}_1\}$$

where, param_0 is the parameter `str` of `nativeSink`, ret_v is the return value, and sink_1 is `puts` (Listing 6.3.1). The results show a dependency between param_0 and sink_1, meaning that if param_0 is tainted then there is a sink to `puts` in native code. However, there is no taint propagating from the return value, as its value does not depend on any inputs. The resulting summary is as follows:

$$\mathsf{T}(\mathsf{param}_0) \Rightarrow \mathsf{Flow}(\mathsf{param}_0, \mathsf{sink}_1)$$

Native Code Source. In this experiment, we wanted to verify whether TaintSaviour could summarize native methods generating a taint. The input and output were as follow:

$$I = \{\mathsf{param}_0, \mathsf{source}_0\}$$
$$O = \{\mathsf{ret}_v\}$$

Here param_0 is the parameter `value` of `nativeSource`, ret_v is the returning value, and source_0 is `source` – i.e., the defined source (Listing 6.3.2). We found a dependency between source_0 and ret_v, meaning that the defined source affects the method's return value. The return value and $param_0$ do not depend on each other, which TaintSaviour correctly infers by not propagating the taint.

Discussion. The results confirmed that our approach is effective for modeling situations where a native method generates a taint, or where it contains a sink. As we previously mentioned, the underlying fuzzing strategy has an impact on the accuracy of the approach. In particular, we are only able to model dependencies reachable with our inputs in the given amount of time.

The time variable has an important role when fuzzing. Fuzzing the same function for 10 minutes can give different results that fuzzing the same function for, say, an hour. In fact, the longer we fuzz a function, the more likely we are to find a dependency, if any exists. Clearly, time is not the only variable, and the diversity of the input generated by the fuzzer have an impact as well. As we discussed earlier (Section 6.3.1), the generated input has an effect on how much output can be explored. However, we are not interested in modeling the exact shape of the function we are testing. Instead, our goal is to find input output dependencies, which is an easier problem to solve. To find a dependency, it is enough to observe one variation of output given different inputs and there is no need to find all the instances where this variation occurs.

For this experiment, and in the time give, the AFL strategy was enough and we could model everything. We showed that TaintSaviour works and successfully summarizes microbenchmarks. However, as we discussed, there might be cases where the fuzzer fails at exposing dependency. For example, if a dependency is never triggered, TaintSavior would cause a subsequent taint analysis to potentially have a false negative, as it would fail to propagate the taint. Moreover, we considered stateless and deterministic functions. This assumption is realistic in the case we are able to enumerate all possible input and output variables. In the specific, the state of a function could be either modeled as an input or an output, and so can the source of non determinism. However, this is currently a limitation of our implementation as we discuss in Section 6.5.

6.3.3 FlowDroid and TaintSaviour

In this section, we evaluate how TaintSaviour models can integrate with state-of-art taint trackers. We chose FlowDroid [3], as it is open-source and widely used. For this evaluation, we explore the following:

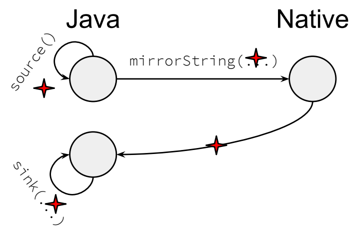
- Our models enable native code taint propagation.
- TaintSaviour removes taints when required, avoiding false positives.
- Our models identify sources in native code.
- Our models identify sinks in native code.

Methodology. We implemented an Android application to test all those use cases. Consider Listing 6.3.3; `nativeSink` and `nativeSource` are the same native methods from Listing 6.3.1 and Listing 6.3.1 . We introduced two more native methods, `mirrorString` and `killTaint`. The former has an Input/Output dependency, while the latter does not. Moreover, the resulting Activity has a source (`javaSource`) and a sink (`sink`) method that we use for our testing purposes. We have four different situations to test:

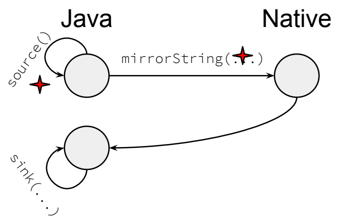
1. We invoke `mirrorString` with a tainted value and we send the execution result to the sink stub.
2. We invoke `killTaint` with a tainted value and we send the execution result to the sink stub.

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  NativeMethods nm = new NativeMethods();

  // Test taint propagation
  String sensitive = javaSource();
  String mirror = nm.mirrorString(sensitive);
  sink(mirror);

  // Test taint is suppressed
  String sensitive1 = javaSource();
  String killed = nm.mirrorKillString(sensitive);
  sink(killed);

  // Test we have a source from native code
  Integer sensitive2 = nm.sourceInNativeCodeTest(42);
  sink(sensitive2.toString());

  // Test we have a sink in native code
  nm.sinkInNativeCodeTest(javaSource());
}
```

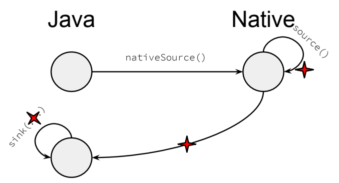
Listing 6.3.3: Native Code Source Method



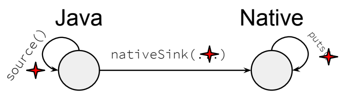
(a) *Flow_1* Native taint propagation.



(b) *Kill* Native taint killed.



(c) *Flow_2* Native source.



(d) *Flow_3* Native sink.

Figure 6.5: *JNI Taint Propagation Cases.* We identified four main scenarios: (i) the taint prograted via native code, (ii) the taint is killed in native code, (iii) the taint is generated in native code, and (iv) there is a sink in native code.

3. We invoke `nativeSource` and send its result to the sink stub.
4. We invoke `nativeSink` with values from our source stub.

The resulting application contains three distinct flows and one interrupted flow as shown in Figure 6.5. We used TaintSaviour to model each native method involved and report them in Table 6.3. We then modified FlowDroid to support JNI models and ran three flow-analysis with the following configurations:

1. Default FlowDroid configuration.
2. FlowDroid extended with an over-approximating model for native methods – i.e., if a native method is invoked with tainted values,

Native Method	Inputs	Outputs	Model
`mirrorString`	$param_0$	ret_v	$T(param_0) \Rightarrow T(ret_v)$
`killTaint`	$param_0$	ret_v	$T(param_0) \Rightarrow ret_v$
`nativeSource`	$param_0$, $source_0$	ret_v	$T(source_0) \vee source_0 \Rightarrow T(ret_v)$
`nativeSink`	$param_0$	ret_v, $sink_0$	$T(param_0) \Rightarrow Flow(param_0, sink_0)$

Table 6.3: *TaintSaviour Models for Native Methods*. With $param_i$ we intend the i_{th} parameter, while ret_v is the return value of the method. $source_0$ is the method `source` in Listing 6.3.2 and $sink_0$ is $puts$ from `libc` in Listing 6.3.1

then always taint the return value (Conservative).

3. FlowDroid extended with TaintSaviour models support.

In its default configuration, FlowDroid uses StubDroid [98] summaries (see Section 6.4) and a set of taint wrappers. FlowDroid uses the *EasyTaintWrapper* when there is no summary. This wrapper contains a list of methods for which their return values and base objects are tainted if they are invoked with tainted values. We developed the conservative taint wrapper as an extension of the EasyTaintWrapper.

Results. We ran the three experiments and compared the results. We report the results in Table 6.4. To analyze the results, consider Figure 6.5. FlowDroid alone did not find any leaks. FlowDroid with the conservative taint wrapper found one flow out of three and introduced a false positive due to the inability of detecting when a taint is killed in native code. Finally, FlowDroid with TaintSaviour's summaries successfully found all the expected leaks.

Discussion. The result of this experiment has confirmed that our approach is a viable solution to enhance the data-flow analysis of Android native applications. With TaintSaviour's summaries, FlowDroid found all the native flows, without introducing any false positives.

FlowDroid Config.	**TP/FP**	**FN**	**Description**
Default	0/0	3	By default, FlowDroid cannot find any leaks involving native code.
Conservative	1/1	2	1 out of 3 leaks are found and 1 false positive is introduced.
TaintSaviour	3/0	0	All and only the expected leaks are found.

Table 6.4: *FlowDroid Different Strategies Runs.* Results of running FlowDroid with three different strategies. In the microbenchmark, we expect to find a total of 3 flows.

Not surprisingly, FlowDroid, with its default configuration could not detect any flow. FlowDroid does its analysis on the Android application Dalvik code – i.e., it considers only Java located in the APK. Therefore, it needs taint wrapppers to propagate taints where the code is not available at the Dalvik level, which is the case for native methods.

As expected, the conservative wrapper found the taint propagation flow. However, it has no information regarding the native method behavior, leading to the false-positive in our evaluation. The lack of information about the native method behavior also prevents this approach from considering sources and sinks in native code. A workaround would be to add all native methods as sources and sinks but this would drastically increase the number of false positives in the results.

In contrast, our approach proved more effective. The Input/Output dependencies behind TaintSaviour's models provide enough information about the method behavior. This information is then reused to instruct the data-flow analysis. In particular, TaintSaviour's models inform the taint tracker of what happens to the inputs and outputs of the native method, resulting in a more precise analysis.

6.3.4 Case Study On App from Google Play Store

In this section, we show how TaintSaviour works on a real application we found on Google Play Store. The goal is to show that our approach can scale beyond microbenchmarks. In this evaluation, we explore the following:

- TaintSaviour enables taint propagation in real-world apps.
- We compare the result of taint analysis with and without TaintSaviour summaries.

Methodology. For this experiment, we reused the same dataset we collected for JniFuzzer (see Section 5.3). It consists of 29,547 apps containing at least one native method from the Google Play Store. We performed a taint analysis and selected apps showing a flow to a JNI method. This resulted in 4,171 apps.

We narrowed down this set further, looking for an interesting application to show case. In a first step, we ran a context-sensitive taint analysis using Flowdroid extended with the conservative model for native methods – i.e., if a native method is invoked with tainted values, then always taint the return value; see Section 6.3.3. This step allowed us to identify applications with a potential flow going through native code. Notice that, a flow reported with this conservative taint wrapper is an overapproximation and therefore could be a false positive. We then imported this result into JniFuzzer framework. The framework allowed us to visualize the flow and browse through the dataset.

We finally found a diving application, *MultiDeco v4.0.4*. From JniFuzzer framework, we saw that a native method [1] was part of a potential flow involving the device id. We found this application to be interesting and we

[1] `<com.*.*.Settings: java.lang.String ic(java.lang.String)>`

```
{
  "<com.*.*.Settings: java.lang.String ic(java.lang.
     String)>": [{
     "input": "param_0",
     "output": "strcpy",
     "flow": true
  },{
    "input": "param_0",
    "output": "ret_value",
    "flow": false
  }]
}
```

Listing 6.3.4: JSON Model for `ic`

verified that our PoC could work without any problem (see Section 6.5). We then summarized that native method and looked for sinks to `strcpy`. To this end, we ran TaintSaviour on a ARM android emulator running Android Marshmallow with 2CPUs and 2GB of RAM. First, we verified that our summaries were valid by manually reversing the native library bytecode, confirming the quality of our model. We then used FlowDroid [3] for a taint analysis, with and without our model, and compared the results.

Resulting Model. We ran TaintSaviour on the selected native method, retrieving the summary in Listing 6.3.4. The summary yields 2 dependencies. The first one is between `ic`'s parameter and `strcpy`, suggesting an information flow to `strcpy`. The second dependency taints the return values, as it depends on `ic`'s parameter.

We reversed the app's native library to validate the summary. In Listing 6.3.5, we show a simplified version of the retrieved code. This code uses the parameter `imei` to retrieve an installation code. At Line 5, it is converted to a buffer, which is then copied to a global `IMEI` variable

```
1  jstring ..._ic(JNIEnv *env, jobject obj, jstring imei){
2    char *my_buf;
3    int len = env->GetStringUTFLength(imei);
4    ...
5    my_buf = env->GetStringUTFChars(imei);
6
7    strcpy(&IMEI, my_buf);
8    const char *installcode = mic(&IMEI);
9
10   if(result){
11     return env->NewStringUTF(installcode);
12   }
13
14   return result;
15 }
```

Listing 6.3.5: Reversed code of `ic`

(Line 7). Based on `IMEI`, an installation code is then calculated (Line 8) and returned to Java as a `jstring`.

We could verify the two dependencies reported in the summary for `ic`. The return value is calculated based on the method parameter and, therefore, they are dependent. Similarly, `imei` is directly used in `strcpy` (Line 7), generating a sensitive information flow.

Taint Analysis. For this experiment, we ran two taint analyses: one using FlowDroid's default configuration and the other with our summary. We report the result in Table 6.5. With TaintSaviour's summary, FlowDroid found two more sensitive information flows. One was from the source `getDeviceId` to the sink `strcpy`. The other one was from the source `getDeviceId` to the sink `startActivityForResults`. We inspected the app's Dalvik and verified that this extra flow is due to a taint propagating via `ic`'s return value.

This experiment has shown that the TaintSavior approach is practical

FlowDroid Config.	Result	Description
Default	2 distinct flows	FlowDroid found 2 flows.
TaintSaviour	4 distinct flows	We found the same leak as the default configuration plus an extra one from `getDeviceId` to `startActivityForResults` and a flow from `getDeviceId` to `strcpy`.

Table 6.5: Result of using FlowDroid with and without TaintSaviour summary of `ic`.

and can work on real apps. However, our implementation is still a proof of concept.

6.4 Related Work

In this section we discuss related work to TaintSaviour. First, we explore different approaches for generating method summaries (Section 6.4.1) and finally, we present related work on Android native components (Section 6.4.2)

6.4.1 Method Summary Approaches

The IFDS framework provides a model and a uniform and polynomial solution for data-flow problems [105]. The authors designed a dynamic-programming algorithm, which reuses low-level method summaries to improve efficiency.

Naeem and Lhotak [106] presented a way to generate summaries, which however, are only meant to summarize aliases. Zuhu et al. [107] use a technique to automatically infer libraries' specifications based on the analysis of the client program. They infer the smallest set of must-not-flow requirements on library functions that are sufficient to ensure that the client program is free from leaks. However, as they state, they require an oracle to verify the generated specification, as the library code is not considered at all.

Rountev et. al. [108] presented a way to summarize methods of a library. First, they perform a data-flow analysis of the library. Second, they remove redundant information from the flow-analysis results.

One recent work that fully automates summary generation is StubDroid [98]. StubDroid uses FlowDroid for preliminary data-flow analysis on the library bytecode, using the results to generate the summaries. The data-flow analysis starts from each public method of the library, considering different data sources (e.g., parameters, static fields, this object, etc.). Accesses to these access paths determine the conditions for taints propagation. Finally, StubDroid serializes the summaries as XML files, one per class, enabling a client to load summaries on demand – i.e., when a library method is invoked. Unfortunately, one of the limitations of StubDroid is the lack of support for native code call. To this end, the authors provided manual summaries. This solution is, of course, not sustainable if there are many native invocations in the library, let alone if the application uses JNI. TaintSaviour is specifically designed to work on native code and therefore it could be used as a complementary of StubDroid. Furthermore, StubDroid's design focuses on Object-Oriented languages (i.e., Java for Android). In contrast, our Input/Output approach provides an abstraction layer above the programming language.

6.4.2 Android Native Components

The primary focus of prior work has been to sandbox native code to limit the risk of malicious code hidden there [109, 110, 111]. Alfonso et al. [112] performed a large-scale measurement across Google Play to estimate the risk posed by native code in Android apps and used this information to automatically generate native code sand-boxing policy. None of the major Android static analysis tools support native code invocation: FlowDroid [3], IccTA [1], Amandroid [4], or CHEX [5]; nor does TaintDroid [113], a dynamic taint tracker.

Closest to our work is JN-SAF [99], which uses static analysis and symbolic execution to generate a model of the JNI framework. JN-SAF is composed of two parts: JavaDroid, built on Amandroid and NativeDroid, built on angr [114], a symbolic execution engine. NativeDroid generates JNI method summaries using symbolic execution of native libraries. JN-SAF inherits the limitations of symbolic execution: path and state explosion during analysis of larger binaries. Additionally, as a static framework, JN-SAF cannot handle libraries that use obfuscation techniques such as string encryption and dynamic code loading, which do not hinder our dynamic, black-box approach. We see the summaries generated by JniFuzzer as being complimentary to the NativeDroid component of JN-SAF.

6.5 Limitations and Future Work

Stateful Methods. The black-box approach described in Section 6.1.3 works under the assumption that methods are side-effect free and that we can exhaustively enumerate all the inputs and outputs of a method. However, without any knowledge of the method internals, we cannot always know the input, or the output shape. Consider the code in Listing 6.5.1 as an example. The method `getAndUpdate` returns the current value of the class variable `state` and updates its value with the one of the parameter `x`. The actual return value of `getAndUpdate` depends on the current execution state and vary accordingly.

TaintSaviour is not aware of the internal state and would fail to sum-

```
class Stateful {
  int state = 0;
  public int getAndUpdate(int x) {
    int tmp = state;
    this.state = x;
    return tmp;
  }
}
```

Listing 6.5.1: Stateful function dependency

marize `getAndUpdate` reporting no dependencies. When fuzzing the target method, TaintSaviour restarts the target method resetting its internal state. Therefore, in the example the only ever explored value for the output would be zero.

Similarly, TaintSaviour cannot say anything about the variable `state`. In fact, it has a direct dependency from the input `x`, which cannot be detected. Therefore a taint propagation through this global variable cannot be properly summarized.

A possible solution to this problem would be to consider global variables as inputs and outputs. In the example, considering `state` as one of the input to fuzz would show a dependency with the return value. Similarly, when `state` is considered an output, we would observe a dependency with the parameter `x`, resulting in `state` to be tainted.

Fuzzing and Technical Limitations. TaintSaviour's analysis uses fuzzing to discover dependencies. Fuzzing is known to be keen to false negatives i.e., it can miss dependencies even when they do exist. This problem goes back to the one of exploring all execution paths of a program (Section 2.4). If a dependency is hidden in a path the fuzzer cannot reach then that dependency is missed.

Another limitation of TaintSaviour is the inability to fuzz inputs other than numerical primitive types and strings. Java Native Interfaces support more complex types, including a C representation of Java Objects, pointers and arrays. How to generate this sort of inputs is challenging and itself an open problem.

We implemented TaintSaviour as a proof of concept and, as such, there are technical limitations. First, TaintSaviour is limited by the JNI environment mocks supported by JniFuzzer. A fully functional implementation would require to implement the entire `jni.h` header, providing meaningful mocks for the exported API. This task requires a noticeable engineering effort, which is out of the scope of this work.

6.6 Work Outcome and Discussion

Taint analysis of Android applications suffers from the presence of native code (Section 1.1.3). In fact, native code is for the most part ignored introducing a considerable gab in the analysis. With TaintSaviour, our goal was to fill this gab and provide a solution to this problem.

With TaintSaviour we introduced a new approach, based on black-box fuzzing, which aims at creating summaries of Android JNIs (Chapter 6). TaintSaviour summaries enable an analysis to identify flows through native code, as well as flows generating and ending there. We implemented TaintSaviour as a plugin for JniFuzzer framework, which helped us overcoming the technical challenges present when analyzing native code.

While TaintSaviour is still a proof of concept, we successfully carried out a preliminary evaluation (Section 6.3). We could show that TaintSavior is practical and that it effectively helps in exposing flows from, to and through native code. To this end, we could found a case study where the device identifier was propagating through JNI. TaintSavior summary enabled a taint analysis to detect this flow, which would otherwise go missing.

While the idea of creating summaries for methods is not new to taint analysis, TaintSaviour is the first approach that uses a dynamic input/output dependency analysis to identify flows propagation. We believe that our research has the potential to scale up and open new avenue of research for the analysis of Android native code (Section 6.5).

7 Conclusion

Android applications are a complex compound of different languages. Their support for web technology and native code poses a serious challenge to their analysis. In fact, most analysis ignore this hybrid nature, effectively introducing gaps in their results. In this thesis, we presented novel techniques to assess the security of Android applications in the light of their hybrid nature.

We began exploring Android Webviews and their strong interaction with the web. In particular, we studied how JavaScript interfaces are used and how they can become a threat to the final user if misused. This problem was not new to literature and existing research showed how JavaScript interface can be dangerous if improperly used [69, 39, 71, 13]. While these study acknowledged the risks involved in using JavaScript interfaces, they were not exploring what is the actual damage that an attack could cause if successful. We believed that this was a noticeable gap and, for this reason, we developed BabelView (Chapter 3). With BabelView, we enhance current static analysis techniques to consider JavaScript interfaces. Considering the hybrid web component in our analysis enabled us to understand what JavaScript interfaces were actually doing and, therefore, we could assess their security in a more granular way. In fact, not all these interfaces are dangerous and labeling an application as dangerous just because it uses JavaScript interfaces is a lax over-approximation.

BabelView is not the only work that pointed out this gap in the state of the art. More work has been carried out on Webviews at the same time BabelView was being developed and afterwards [35, 74, 34, 36, 38]. Because our work and all the work that followed, it is now clearer the importance of having analysis that can tackle the web component of Android applications. With BabelView, we showed the state of Webview security, finding 10,808 potential vulnerabilities in 4,997 (Chapter 4).

After Webviews, we switch our focus on another hybrid component of Android applications: native code. Native code is currently one of the most challenging aspects to consider when performing an analysis. This resulted in tools that marginally consider or totally ignore native code, which is often listed as a limitation of the analysis.

Performing an analysis that considers native codes comes with different challenges. First, there is the hybrid component. Android applications are mainly written in Java and analyses are principally tuned for it. It not trivial to implement a tool that can model the interaction among Java and native code. Second, to analyze native code one needs to enter the realm of binary analysis, which is notoriously complex (source code is a luxury that in most cases is not available). Moreover, the fast changing Android echo-system pose yet another challenge, resulting in tools that are now outdated and unusable because the lack of maintenance and support.

Our wok on native code aims at filling this gap. We wanted to provide an easy to use framework that can be extended to support native code analysis. We also wanted to provide a new approach that could integrate with existing state-of-art tools to enable native code analysis. To this end we first developed JniFuzzer framewrok (Chapter 5). JniFuzzer uses a black-box fuzzing strategy that directly targets Android native code (JNI). One advantage of JniFuzzer is that it not tight to a specific Android version. Moreover, we implemented it in an extensible way, enabling future researcher to developed their analysis within the framework. We successfully used JniFuzzer on real world applications and could find some potential security issues in their native libraries (Section 5.3).

One interesting research question opened by JniFuzzer is whether a bug found in native code could be triggered from the application level (i.e., Java). As we discussed in Section 5.5, the presence of a security bug in native code does not necessarily means that the application is vulnerable. It would be interesting to explore techniques to propagate inputs from a source to the native method. If then there are ways for the input to be the same as the one produced by the fuzzer, we could conclude that the application is indeed vulnerable.

Secondly, we focused on how we could extend existing taint-analysis approaches to consider native code. To this end, we extended JniFuzzer and developed a novel approach, TaintSaviour (Chapter 6), which adopt a black-box fuzzing approach to create summaries for Andorid JNI. These summaries can then be reused by existing taint analysis to include native code. We tested and preliminary evaluated TaintSaviour and we could show that this approach is practical and works on real applications (Section 6.3).

TaintSaviour is not the only approach that aims at creating summaries of methods. For example, one noticeable work in the field is StubDroid [98]. StubDroid as well enhance taint analysis with models of methods that would be unexplored otherwise. However, its implementation cannot deal with native code. We see TaintSaviour as a complimentary of StubDroid. Wiring both the approaches to taint analysis, would bring about the advantages of both, i.e., analysis of Android framework and native code.

We were not the only one interested in filling the gaps for Android native code analysis. While we were developing TaintSaviour, JN-SAF was published [99]. The final goal of this research is to enable taint propagation across JNI functions. TaintSaviour has a similar goal. However, the approach we took is orthogonal to JN-SAF one. While they use heavy handed techniques, such as symbolic execution, to model native code, TaintSaviour uses a black-box fuzzing approach, which is far less computationally expensive. Unfortunately, we were not able to carry out a comparison among the two approaches. The lack of time and resources prevented us from having a fair confrontation, which we left for future work.

One of the biggest outcome of our research, and more generally of the research in our field, is to raise the awareness of security in the broader field of compute science and engineering. We have already seen the good impact of it. For example, security is now consider one of the big components in the life cycle of applications. Continuous integration (CI) chain more often include static analysis techniques as a mean of discovery security bugs before they are released to the public. Tool such as BabelView or JniFuzzer could be taken further and deployed in such context. For example, an Android web developers might benefit from finding out that their Webviews use dangerous JavaScript interfaces. In fact, BabelView could deployed as a plugin for Android integrated development environments. Also, its report could be tuned to block certain patterns to even go in the production branch without a further review. Similarly, JniFuzzer could be executed as a step in the application CI, potentially preventing bugs to arise later in production.

Acronyms

AFL	American Fuzzy Lop.
APK	Android Pakcage.
ART	Android RunTime.
GUI	Graphical User Interface.
HAL	Hardware Abstraction Layer.
HTTP	HyperText Transfer Protocol.
HTTPS	HyperText Transfer Protocol Secure.
ICC	Inter Component Communication.
IFDS	Interprocedural Finite Distributive Subset problem.
IoT	Internet of Things.
ISA	Instruction Set Architecture.
JNI	Java Native Interface.
JVM	Java Virtual Machine.
LCM	Least Common Multiple.

MITM	Man in The Middle.
NDK	Native Development Kit.
OS	Operating System.
PoC	Proof of Concept.
SDK	Software Development Kit.
SMT	Satisfiability Modulo Theories.
SOP	Same Origin Policy.
SSL	Secure Sockets Layer.
TLS	Transport Layer Security.
UI	User Interface.
XML	Extensible Markup Language.
XSS	Cross Site Scripting.

Bibliography

[1] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *Int. Conf. Softw. Eng. ICSE*. {IEEE} Computer Society, 2015, pp. 280–291.

[2] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," Tech. Rep. June, 2014. [Online]. Available: http://www.cs.cmu.edu/~wklieber/papers/soap2014-didfail.pdf

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *ACM SIGPLAN Conf. Program. Lang. Des. Implementation, PLDI*, vol. 49, no. 6. ACM, 2014, pp. 259–269.

[4] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*. New York, New York, USA: ACM Press, 2014, pp. 1329–1341. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660267.2660357

[5] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting

Android apps for component hijacking vulnerabilities," in *{ACM} Conf. Comput. Commun. Secur.* ACM, 2012, pp. 229–240.

[6] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-Flow Analysis of Android Applications in DroidSafe," in *22nd Annu. Netw. Distrib. Syst. Secur. Symp. NDSS*, 2015. [Online]. Available: http://dx.doi.org/10.14722/ndss.2015.23089

[7] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A Survey of Mobile Malware in the Wild," in *SPSM'11, Proc. 1st ACM Work. Secur. Priv. Smartphones Mob. Devices, Co-located with CCS*, 2011. [Online]. Available: www.cs.berkeley.edu/~daw/malware.html

[8] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis *," in *Proc. 22nd {ACM} SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014. [Online]. Available: http://dx.doi.org/10.1145/2635868.2635869

[9] J. Qiu, S. Nepal, W. Luo, L. Pan, Y. Tai, J. Zhang, and Y. Xiang, "Data-Driven Android Malware Intelligence: A Survey," in *Mach. Learn. Cyber Secur. - Second Int. Conf. ML4CS*, vol. 11806 LNCS. Springer Verlag, 2019, pp. 183–202.

[10] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, jan 2017.

[11] G. Suarez-Tangil and G. Stringhini, "Eight Years of Rider Measurement in the Android Malware Ecosystem," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, mar 2020.

[12] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are these Ads Safe: Detecting Hidden Attacks through the

Mobile App-Web Interfaces," 2016. [Online]. Available: http://dx.doi.org/10.14722/ndss.2016.23234

[13] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Twenty-Seventh Annu. Comput. Secur. Appl. Conf. ACSAC*, 2011, pp. 343–352.

[14] V. Moonsamy, M. Alazab, and L. Batten, "Towards an understanding of the impact of advertising on data leaks," Tech. Rep. 3, 2012.

[15] S. Son, G. Daehyeok, K. Kaist, and V. Shmatikov, "What Mobile Ads Know About Mobile Users," 2016. [Online]. Available: http://dx.doi.org/10.14722/ndss.2016.23407

[16] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proc. Second ACM Work. Secur. Priv. smartphones Mob. devices*. New York, New York, USA: ACM Press, 2012, pp. 93–104. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2381934.2381950

[17] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. 2013 ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. OOPSLA*, vol. 48, no. 10. New York, New York, USA: ACM Press, 2013, pp. 623–639. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2509136.2509552

[18] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. 2013 ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. OOPSLA*, vol. 48, no. 10. New York, New York, USA: ACM Press, 2013, pp. 641–660.

[Online]. Available: http://dl.acm.org/citation.cfm?doid=2509136.2509549

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, sep 2015. [Online]. Available: https://ieeexplore.ieee.org/document/6786194/

[20] W. Choi, K. Sen, G. Necula, and W. Wang, "DetReduce: Minimizing Android GUI test suites for regression testing," in *Proc. - Int. Conf. Softw. Eng.*, vol. 2018-Janua, 2018. [Online]. Available: https://doi.org/10.1145/3180155.3180173

[21] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "LNCS 7313 - Correlation Tracking for Points-To Analysis of JavaScript," Tech. Rep., 2012. [Online]. Available: http://jquery.com

[22] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *2013 9th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. ESEC/FSE 2013 - Proc.*, 2013, pp. 499–509.

[23] M. Zalewski, "Technical whitepaper for afl-fuzz," \url{http://lcamtuf.coredump.cx/afl/technical_details.txt}, 2014.

[24] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the impact of code injection attacks in mobile webviews," in *RAID 2018 Res. Attacks, Intrusions, Defenses*, vol. 11050 LNCS. Springer Verlag, 2018, pp. 25–46.

[25] "App Manifest Overview | Android Developers." [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro (Accessed 2019-11-29).

[26] Android, "Building Web Apps in WebView," 2016. [Online]. Available: http://developer.android.com/guide/webapps/webview.html (Accessed 2019-09-03).

[27] Google, "WebView | Android Developers," 2012. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView.html#WebView(android.content.Context) (Accessed 2019-05-17).

[28] Google, "View | Android Developers." [Online]. Available: https://developer.android.com/reference/android/view/View.html (Accessed 2019-09-03).

[29] Google, "Custom View Components | Android Developers." [Online]. Available: https://developer.android.com/guide/topics/ui/custom-components (Accessed 2019-05-17).

[30] MWR Infosecurity Labs, "WebView addJavascriptInterface Remote Code Execution," pp. 1–6, 2013. [Online]. Available: https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/%5Cnhttps://archive.is/KxtXb (Accessed 2019-05-20).

[31] A. Barth, "The Web Origin Concept." [Online]. Available: https://tools.ietf.org/html/rfc6454 (Accessed 2020-06-06).

[32] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android API vulnerabilities: Case study on the JavaScript-to-Java interface," in *Secur. Protoc. XXIII - 23rd Int. Work.*, vol. 9379. Springer, 2015, pp. 126–138.

[33] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection

and mitigation," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*. ACM, 2014, pp. 66–77.

[34] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications," in *Proc. - IEEE Symp. Secur. Priv.*, vol. 2018-May, 2018, pp. 742–755.

[35] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and Scalably Vetting JavaScript Bridge in Android Hybrid Apps," in *Res. Attacks, Intrusions, Defenses - 20th Int. Symp. RAID*, vol. 10453 LNCS. Springer, 2017, pp. 143–166.

[36] G. Yang, J. Huang, and G. Gu, "Automated Generation of Event-Oriented Exploits in Android Hybrid Apps," in *Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[37] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View to a Kill: WebView Exploitation," in *USENIX Work. Large-Scale Exploit. Emergent Threat.*, 2013.

[38] T. Li, X. Wang, M. Zha, K. Chen, X. F. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing thewalking dead: Understanding cross-app remote infections on mobilewebviews," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*, 2017, pp. 829–844.

[39] B. Hassanshahi, Y. Jia, R. H. Yap, P. Saxena, and Z. Liang, "Web-to-application injection attacks on android: Characterization and detection," in *Comput. Secur. 20th Eur. Symp. Res. Comput. Secur. ESORICS*, vol. 9327. Springer, 2015, pp. 577–598.

[40] "Android Enterprise Security White Paper," Tech. Rep., 2018.

[41] A. Possemato and Y. Fratantonio, "Towards HTTPS Everywhere on Android: We Are Not There Yet Towards HTTPS Everywhere on Android: We Are Not There Yet," in *Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.*, 2020, pp. 343–360. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/possemato

[42] H. Guihot and H. Guihot, "Getting Started With the NDK," pp. 33–71, 2012. [Online]. Available: https://developer.android.com/ndk/guides (Accessed 2019-07-29).

[43] Oracle, "JNI APIs and Developer Guides." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/ (Accessed 2019-07-25).

[44] Google, "runtime/jni - platform/art - Git at Google." [Online]. Available: https://android.googlesource.com/platform/art/+/refs/heads/master/runtime/jni/ (Accessed 2019-07-30).

[45] Oracle, "JNI Functions." [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#wp16656 (Accessed 2019-07-31).

[46] Oracle, "JNI Modified UTF-8." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html#modified_utf_8_strings (Accessed 2019-07-31).

[47] Oracle, "JNI | Exceptions." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html#java_exceptions (Accessed 2019-08-08).

[48] Y. Smaragdakis and G. Balatsouras, "Pointer Analysis," *Found. Trends® Program. Lang.*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: http://www.nowpublishers.com/article/Details/PGL-014

[49] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," Tech. Rep., 2018. [Online]. Available: http://arxiv.org/abs/1812.00140

[50] P. Godefroid, "Software Model Checking Improving Security of a Billion Computers," in *Model Checking Software, 16th Int. SPIN Work.*, 2009, pp. 1–1. [Online]. Available: https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

[51] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, jul 1976. [Online]. Available: http://portal.acm.org/citation.cfm?doid=360248.360252

[52] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *25th Annu. Netw. Distrib. Syst. Secur. Symp. NDSS*. The Internet Society, 2018.

[53] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *Proc. - 2014 IEEE 15th Int. Symp. High-Assurance Syst. Eng. HASE 2014*. IEEE, jan 2014, pp. 176–183. [Online]. Available: http://ieeexplore.ieee.org/document/6754603/

[54] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *CODASPY 2013 - Proc. 3rd ACM Conf. Data Appl. Secur. Priv.* New York, New

York, USA: ACM Press, 2013, pp. 209–220. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2435349.2435379

[55] Google, "UI/Application Exerciser Monkey | Android Developers," 2019. [Online]. Available: https://developer.android.com/studio/test/monkey.html%0Ahttps://developer.android.com/studio/test/monkey (Accessed 2019-09-03).

[56] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proc. Conf. Cent. Adv. Stud. Collab. Res.*, 1999, p. 13.

[57] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," Tech. Rep. [Online]. Available: https://svn.sable.mcgill.ca/wiki/

[58] A. Bartel, J. Klein, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP 2012*. New York, New York, USA: ACM Press, 2012, pp. 27–38. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2259051.2259056

[59] S. S. Einarsson, J. Dam, and N. Brics, "A Survivor's Guide to Java Program Analysis with Soot´Arni," Tech. Rep.

[60] "androguard/androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)." [Online]. Available: https://github.com/androguard/androguard (Accessed 2020-07-09).

[61] “dex2jar: Tools to work with android .dex and java .class files.” [Online]. Available: https://github.com/pxb1988/dex2jar (Accessed 2020-07-10).

[62] “Jasmin Home Page.” [Online]. Available: http://jasmin.sourceforge.net/ (Accessed 2020-07-10).

[63] “Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps.” [Online]. Available: https://ibotpeaches.github.io/Apktool/ (Accessed 2020-07-10).

[64] “JesusFreke/smali: smali/baksmali.” [Online]. Available: https://github.com/JesusFreke/smali (Accessed 2020-07-10).

[65] F. Pauck, E. Bodden, and H. Wehrheim, “Do android taint analysis tools keep their promises?” in *ESEC/FSE 2018 - Proc. 2018 26th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, New York, USA: ACM Press, 2018, pp. 331–341. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3236029

[66] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, “Static analysis of android apps: A systematic literature review,” *Inf. Softw. Technol.*, vol. 88, pp. 67–95, aug 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917302987

[67] “DidFail.” [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=508078 (Accessed 2020-07-09).

[68] L. Qiu, Y. Wang, and J. Rubin, “Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe,” 2018. [Online]. Available: https://doi.org/10.1145/3213846.3213873

[69] A. Doupé, P. Mutchler, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security CTFs for Education View project A Large-Scale Study of Mobile Web App Security," in *Proc. IEEE Symp. Secur. Priv. Work. (SPW), Mob. Secur. Technol.* IEEE, 2015. [Online]. Available: https://www.researchgate.net/publication/278724743

[70] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*, 2012, pp. 50–61. [Online]. Available: http://android-ssl.org/s/3

[71] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Inf. Secur. Appl. - 14th Int. Work. WISA*, vol. 8267 LNCS. Springer, 2014, pp. 138–159.

[72] A. B. Bhavani, "Cross-site Scripting Attacks on Android WebView," *CoRR*, vol. abs/1304.7, 2013. [Online]. Available: http://arxiv.org/abs/1304.7451

[73] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps: A report on the state of Apache Cordova nation," in *Eng. Secur. Softw. Syst. - 8th Int. Symp. ESSoS 2016*, vol. 9639. Springer, 2016, pp. 72–88.

[74] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for android hybrid applications," in *ASE 2016 - Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.* ACM, 2016, pp. 250–261.

[75] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks," in *Annu. Netw. Distrib. Syst. Secur. Symp.* The Internet Society, 2014.

[76] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on Web in Android, iOS, and Windows Phone," in *Found. Pract. Secur. - 5th Int. Symp. FPS*, vol. 7743 LNCS. Springer, 2013, pp. 227–243.

[77] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks," in *Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014.

[78] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on Android," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*, vol. 24-28-Octo, 2016, pp. 104–115. [Online]. Available: https://dl.acm.org/citation.cfm?doid=2976749.2978322

[79] M. Shehab and A. Aljarrah, "Reducing attack surface on cordova-based hybrid mobile apps," in *MobileDeLi 2014 - Proc. 2nd Int. Work. Mob. Dev. Lifecycle, Part SPLASH 2014*. ACM, 2014, pp. 1–8.

[80] X. Jin, L. Wang, T. Luo, and W. Du, "Fine-grained access control for HTML5-based mobile applications in android," in *Inf. Secur. 16th Int. Conf. ISC*, vol. 7807, 2015, pp. 309–318.

[81] P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar, "Hybrid-Guard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications," in *Proc. - 2017 IEEE Symp. Secur. Priv. Work. SPW 2017*, vol. 2017-Decem, 2017, pp. 147–156.

[82] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proc. 2006 EuroSys Conf.* ACM, 2006, pp. 73–85.

[83] J. Kinder and H. Veith, “Precise static analysis of untrusted driver binaries,” in *Proc. 10th Int. Conf. Form. Methods Comput. Des. FMCAD*, 2010, pp. 43–50.

[84] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “AndroZoo: Collecting millions of Android apps for the research community,” in *Proc. - 13th Work. Conf. Min. Softw. Repos. MSR 2016*. ACM, 2016, pp. 468–471.

[85] P. Mutchler, Y. Safaei, A. Doupe, and J. Mitchell, “Target Fragmentation in Android Apps,” in *Proc. - 2016 IEEE Symp. Secur. Priv. Work. SPW 2016*. IEEE, 2016, pp. 204–213.

[86] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, “Measuring the declared SDK versions and their consistency with API calls in android apps,” in *Wirel. Algorithms, Syst. Appl. - 12th Int. Conf. WASA*, vol. 10251 LNCS, 2017, pp. 678–690.

[87] D. R. Thomas, A. R. Beresford, and A. Rice, “Security metrics for the android ecosystem,” in *SPSM 2015 - Proc. 5th Annu. ACM CCS Work. Secur. Priv. Smartphones Mob. Devices, co-located with CCS 2015*. ACM, 2015, pp. 87–98.

[88] E. Athanasopoulos, V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, “NaClDroid: Native Code Isolation for Android Applications,” in *Comput. Secur. - ESORICS 2016 - 21st Eur. Symp. Res. Comput. Secur.*, ser. Lecture Notes in Computer Science, vol. 9878. Springer, 2016, pp. 422–439.

[89] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, “The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching,” in *{IEEE} Symp. Secur. Priv.* {IEEE} Computer Society, 2015, pp. 692–708.

[90] Google, “JNI tips | Android NDK | Android Developers,” 2020. [Online]. Available: https://developer.android.com/training/articles/perf-jni (Accessed 2020-09-26).

[91] A. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in *Proc. - Work. Conf. Reverse Eng. WCRE*, vol. 2003-Janua. IEEE Computer Society, 2003, pp. 260–269.

[92] R. Sasnauskas and J. Regehr, “Intent fuzzer: Crafting intents of death,” in *WODA+PERTEA 2014 Jt. 12th Int. Work. Dyn. Anal. Work. Softw. Syst. Perform. Testing, Debugging, Anal. - Proc.* Association for Computing Machinery, Inc, jul 2014, pp. 1–5.

[93] H. Ye, S. Cheng, L. Zhang, and F. Jiang, “DroidFuzzer: Fuzzing the Android apps with intent-filter tag,” in *11th Int. Conf. Adv. Mob. Comput. &Multimedia, MoMM*, 2013, pp. 68–74.

[94] C. Cao, N. Gao, P. Liu, and J. Xiang, “Towards analyzing the input validation vulnerabilities associated with android system services,” in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, vol. 7-11-Decem. ACM, 2015, pp. 361–370.

[95] M. Y. Wong and D. Lie, “IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware,” in *Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[96] “moflow/afl-dyninst at master · Cisco-Talos/moflow.” [Online]. Available: https://github.com/Cisco-Talos/moflow/tree/master/afl-dyninst (Accessed 2019-11-25).

[97] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, “Korat: A tool for generating structurally complex test inputs,” in *29th Inter-*

national Conference on Software Engineering (ICSE'07), May 2007, pp. 771–774.

[98] S. Arzt and E. Bodden, "StubDroid: Automatic inference of precise data-flow summaries for the android framework," in *Proc. - Int. Conf. Softw. Eng. ICSE*, vol. 14-22-May. ACM, 2016, pp. 725–735. [Online]. Available: http://dx.doi.org/10.1145/2884781.2884816

[99] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*. New York, New York, USA: ACM Press, 2018, pp. 1137–1150. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3243734.3243835

[100] P. J. Schroeder and B. Korel, "Black-box test reduction using input-output analysis," in *Proc. ACM SIGSOFT 2000 Int. Symp. Softw. Test. Anal.*, 2000, pp. 173–177.

[101] B. Korel, "The program dependence graph in static program testing," *Inf. Process. Lett.*, vol. 24, no. 2, pp. 103–108, jan 1987. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0020019087901025

[102] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *Int. Symp. Program.*, vol. 167 LNCS, 1984, pp. 125–132. [Online]. Available: https://www.cs.utexas.edu/~pingali/CS395T/2009fa/papers/ferrante87.pdf

[103] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, vol. 20-22-June, no. 6. New York, New York, USA: ACM Press, 1990,

pp. 246–256. [Online]. Available: http://portal.acm.org/citation.cfm?doid=93542.93576

[104] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," Tech. Rep. 7, 1988. [Online]. Available: http://cist.buct.edu.cn/staff/zheng/COMP544-PA/papers/10.1.1.50.4405.pdf

[105] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Conf. Rec. Annu. ACM Symp. Princ. Program. Lang. POPL*. ACM, 1995, pp. 49–61.

[106] N. A. Naeem and O. Lhoták, "Faster alias set analysis using summaries," in *Compil. Constr. - 20th Int. Conf. CC*, vol. 6601 LNCS, 2011, pp. 82–103.

[107] H. Zhu, T. Dillig, and I. Dillig, "Automated inference of library specifications for source-sink property verification," in *Program. Lang. Syst. - 11th Asian Symp. APLAS*, vol. 8301 LNCS, 2013, pp. 290–306.

[108] A. Rountev, M. Sharp, and G. Xu, "IDE dataflow analysis in the presence of large object-oriented libraries," in *ompiler Constr. 17th Int. Conf. CC*, vol. 4959 LNCS, 2008, pp. 53–68.

[109] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-Privilege Compartments," in *5th USENIX Symp. Networked Syst. Des. Implementation, NSDI*. {USENIX} Association, 2008, pp. 309–322.

[110] M. Sun and G. Tan, "NativeGuard: protecting android applications from third-party native libraries," in *7th ACM Conf. Secur. Priv. Wirel. Mob. Networks, WiSec'14*. ACM, 2014, pp. 165–176.

[111] J. Siefers, G. Tan, and G. Morrisett, "Robusta: taming the native beast of the JVM," in *{ACM} Conf. Comput. Commun. Secur. CCS*. ACM, 2010, pp. 201–211.

[112] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, "Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy," in *23rd Annu. Netw. Distrib. Syst. Secur. Symp. NDSS*. San Diego, California, USA: The Internet Society, 2016.

[113] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *9th USENIX Symp. Oper. Syst. Des. Implementation, OSDI*. {USENIX} Association, 2010, pp. 393–407.

[114] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *{IEEE} Symp. Secur. Priv.* {IEEE} Computer Society, 2016, pp. 138–157.