Informatics

Bridging Realms: Analyzing App-to-Web Interactions in Android IABs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Beer, BSc
Matrikelnummer 11807877

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dott.ric. Marco Squarcina
Mitwirkung: Dr.-Ing. Sebastian Roth, MSc
Dipl.-Ing.in Dr.in techn. Martina Lindorfer

Wien, 8. Mai 2024

Philipp Beer Marco Squarcina

Informatics

Bridging Realms: Analyzing App-to-Web Interactions in Android IABs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Beer, BSc
Registration Number 11807877

to the Faculty of Informatics

at the TU Wien

Advisor: Dott.ric. Marco Squarcina
Assistance: Dr.-Ing. Sebastian Roth, MSc
Dipl.-Ing.in Dr.in techn. Martina Lindorfer

Vienna, 8th May, 2024

Philipp Beer

Marco Squarcina

Erklärung zur Verfassung der Arbeit

Philipp Beer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Mai 2024

Philipp Beer

Acknowledgements

I would like to express my sincere gratitude to all those who have supported me in completing this thesis. First and foremost, thank you to my supervisors, Marco, Sebastian, and Martina, for their guidance and support throughout this thesis. Your insights have been crucial to my work.

Special thanks go to David, who not only pointed me in the right direction during challenging times but also provided consistent support throughout the thesis.

My deepest appreciation is for my family and friends, whose endless encouragement and support have been my stronghold. I am particularly grateful to my parents, Florian and Krimhild, who allowed me to pursue my academic goals.

Thank you all for your support and encouragement.

Kurzfassung

In-App-Browser (IABs) sind häufig verwendete Komponenten in mobilen Anwendungen, die es App-Entwicklern ermöglichen, Webinhalte in nativen Anwendungen anzuzeigen. Neben der einfachen Darstellung von Webinhalten bieten solche Komponenten der Anwendung Funktionen wie das Einfügen von JavaScript-Code und den Zugriff auf die Cookies der Website. Während diese Funktionen für Entwickler nützlich sind, ermöglichen sie es potenziell unerwünschten Anwendungen (PUAs), bösartige Aktivitäten auf gutartigen Websites durchzuführen, wie z. B. Session-Hijacking durch JavaScript-Injection. In dieser Arbeit wird ein neuartiger Ansatz zur Analyse von App-to-Web-Interaktionen in Android WebView, der primäre integrierten IAB-Komponente in Android, vorgestellt. Wir verwenden eine Kombination aus statischen und dynamischen Analysetechniken, um zunächst den Bauplan einer Anwendung zu erstellen und dann die App mithilfe dessen zu Codeabschnitten zu navigieren, bei denen IABs gestartet werden. Unsere kontrollierte Umgebung ermöglicht es uns, die Interaktionen zwischen der Anwendung und dem Webinhalt aufzuzeichnen und so Falsch-Positive zu minimieren. Wir implementieren unseren Ansatz in Form eines Prototyps namens IABInspect und wenden ihn auf 1.000 beliebte Android-Anwendungen an. Insgesamt konnten wir 508 IAB-Aufrufe in 196 Anwendungen dynamisch auslösen und in 50 Anwendungen eine Injektion von JavaScript-Code finden. Unsere Ergebnisse zeigen, dass die Verwendung von WebViews in Android-Anwendungen allgegenwärtig ist und dass das Einfügen von JavaScript-Code eine gängige Praxis ist, was den Bedarf an weiterer Forschung in diesem Bereich unterstreicht.

Abstract

In-app browsers (IABs) are heavily used components in mobile applications that allow app developers to display web content in native applications. Apart from simply rendering web content, such components provide the application with capabilities like the injection of JavaScript code and access to the website's cookies. While these features are useful for developers, they also allow potentially unwanted applications (PUAs) to perform malicious activities on benign websites, such as session hijacking using JavaScript injection. This thesis presents a novel approach to analyzing app-to-web interactions in Android WebView, the main built-in IAB component in Android. We use a combination of static and dynamic analysis techniques to first build a blueprint of an application and then dynamically drive the execution of the application to calls where IABs are launched. Our controlled environment allows us to record the interactions between the app and the web content, effectively minimizing false positives. We implement our approach as a prototype called IABInspect and apply it to 1,000 popular Android applications. In total, we are able to dynamically trigger 508 IAB launch calls in 196 applications and find an injection of JavaScript code in 50 applications. Our results show that the use of WebViews is ubiquitous in Android applications and that the injection of JavaScript code is a common practice, underscoring the need for further research in this area.

Contents

CHAPTER 1

Introduction

The rise of smartphones has significantly shifted the way users access web content. Users no longer exclusively view web content on desktop devices but increasingly on their smartphones as well. While in the first quarter of 2015, only 31.16% of the worldwide website traffic was generated through mobile devices, this has risen up to 54.7% in the third quarter of 2023 [1]. On desktop devices, the browser, e.g., Google Chrome or Mozilla Firefox, is usually the primary means to access web content. While the browser is still a possible way to access web content on mobile devices, users can also use non-browser applications that offer so-called *In App Browsers* (IABs) to view web content. Both Android and iOS, the two most popular mobile operating systems that hold a combined global market share of 99.3% as of 2023 [2], allow such practices and offer developers different components (*embedded browsers*) that allow the rendering of web content within their applications. These components are heavily used in mobile applications. Android WebView, a component used to render web content within Android applications, is used by 85% of all free applications listed on the Google Play Store as of June 2014 [3].

While such embedded browsers enable web content rendering within applications, some of them also offer the possibility to interact with the web content displayed in them, such as Android's WebView and iOS's WKWebView. Other components, such as Android Custom Tabs and iOS SFSafariViewController, are more limited in their capabilities but provide features supported by a fully-fledged browser, such as Safe Search.

Interactions of IABs and applications can be categorized into two types: *Web-To-App Interactions*, where web content can interact with the application, and *App-To-Web Interactions*, where the application modifies or retrieves information from the web content. With web-to-app interactions, applications can, for example, declare that web content can call specific functions from the application. This can be used by a website to read a user's contacts or instruct the device to send an SMS message. In the case of app-to-web interactions, applications can add cookies to a website loaded in an IAB to automatically log users in and thus contribute to a seamless experience for the users.

Similarly, they can read cookies from the website in an IAB used for authentication and use them to authenticate the user in the application. Moreover, applications can inject JavaScript code into the web content to customize it or add additional functionality. For example, an application that provides an IAB to browse a third-party website can use JavaScript to change the website's text font.

While these interactions provide significant benefits for developers, they also pose severe threats to the users' security and privacy. Extensive research on web-to-app interactions [4, 5, 6, 7, 8, 9, 10, 11, 12] has revealed various vulnerabilities in this type of interaction. For instance, research has shown how malicious websites loaded in IABs in vulnerable applications can access sensitive user information like call logs and contacts. However, app-to-web interactions have not been as thoroughly investigated despite their potential for misuse. For example, a *Potentially Unwanted Application* (PUA), i.e., an application that seems benign and useful but may have functionality that the user does not desire, could abuse these interactions to steal the cookie of a user on a website loaded in an IAB and conduct session hijacking. By allowing applications to inject JavaScript code into the web content, applications can manipulate the web content, read the user's username, password, or other sensitive information displayed on the website, and potentially monitor the user's actions. An investigation by Felix Krause [13] in August 2022 that received worldwide media attention highlighted this issue. By conducting a manual analysis of a handful of social network applications for iOS, among which were popular applications like Facebook, Instagram, and Twitter, he found that these applications inject JavaScript code into the website that can be used to monitor and intercept user actions. This discovery underscores the need for a systematic examination of app-to-web interactions. The limited body of previous research in this area [4, 13, 14] is characterized by high rates of false positives due to only relying on static analysis. Moreover, other studies lack breadth, focusing on only a few applications.

To fill this gap, in this thesis, we present a novel approach to detect app-to-web interactions in Android IABs and implement our approach in a prototype tool called *IABInspect*. Our approach combines static and dynamic analysis to detect app-to-web interactions in Android applications. We use static analysis to identify launch calls of IABs and find paths from an entry point of the application to the IAB. We then use dynamic analysis to drive the execution along these paths and detect app-to-web interactions by monitoring the interactions in our controlled environment. IABInspect targets Android applications due to Android's popularity and the availability of tools to analyze applications. It focuses on Android WebView as the underlying embedded browser.

In specific, we make the following contributions:

- we provide a summary of the state-of-the-art research on IAB security (Chapter 3),
- we present a novel approach based on a combination of static and dynamic analysis to detect app-to-web interactions in Android applications and implement a prototype tool called *IABInspect* tool (Chapter 4),

- we evaluate the effectiveness of IABInspect through a detailed manual analysis of 10 applications, comparing these manual findings with the automated results generated by our tool to ensure precision (Chapter 5), and
- we apply IABInspect to a dataset of 1,000 popular Android applications, successfully triggering 508 IAB launch calls across 196 applications. Our analysis reveals 137 unique JavaScript code snippets being injected into web content in 50 applications (Chapter 6).

To facilitate reproducibility, we open-source the implementation of IABInspect on `https://purl.org/bridging-realms`.

CHAPTER 2

Background

This chapter introduces basic concepts that this thesis relies upon. We first discuss the basics of the Android operating system and then provide a brief overview of In-App Browsers (IABs). Following this, we present fundamental app analysis principles, including static and dynamic analysis approaches.

2.1 Android Operating System

Android, developed by Google, is the most popular operating system for mobile computing, holding a worldwide market share of 70.11% and powering the vast majority of smartphones, tablets, and other mobile devices [2]. Its popularity and open-source nature, which contrasts with iOS's closed-source approach, make Android an attractive platform for researchers.

2.1.1 Basics of Android Applications

Android applications are primarily developed in Kotlin, as it is officially the preferred language for Android application development since 2019 [15]. However, applications can also be developed in Java. Moreover, the Native Development Kit (NDK) enables the integration of native code, such as C or C++, which can then be called from either Java or Kotlin code.

Unlike traditional Java and Kotlin applications that compile into Java-compatible bytecode (`.class` files) and are executed by the Java Virtual Machine (JVM), Android applications run on the Android Runtime (ART). Since ART cannot directly execute Java bytecode, it is instead compiled into Dalvik bytecode (`.dex` files), which is then executed by the Dalvik virtual machine. Native code, such as C, is compiled into shared libraries (`.so`). These are bundled with Dalvik bytecode and other resources, such as images and XML layout files, into an Android Package (APK). This APK file, which the developer

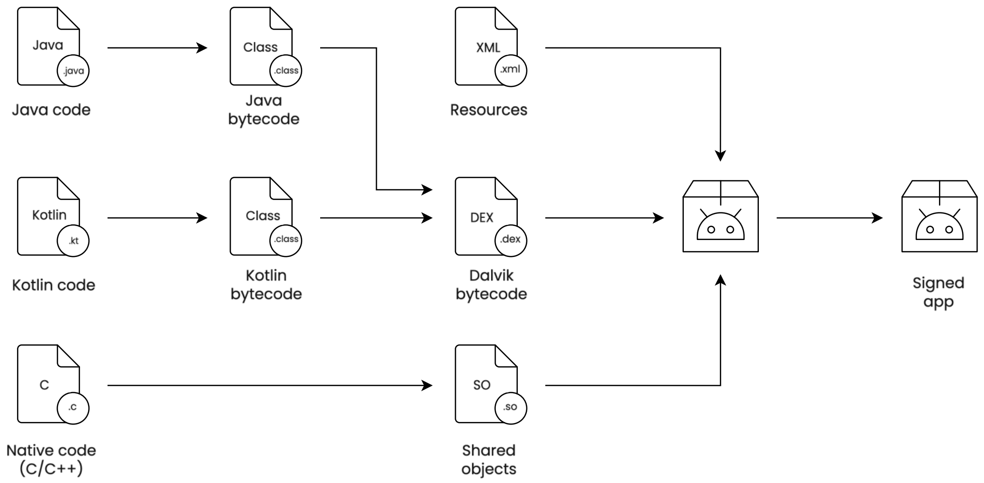


Figure 2.1: Overview of the Android application generation process.

must digitally sign, serves as the distribution format for Android applications [16]. An overview of this process is shown in Figure 2.1.

Each application installed on an Android device operates within its own sandbox, meaning it runs in a separate process and is assigned a unique user ID. This architecture ensures that applications are isolated from one another, securing them against unauthorized data access.

2.1.2 Android Application Components

Unlike traditional Java applications, which typically start the execution at a single `main` method, Android applications are highly event-driven and can have multiple entry points. Application developers implement specific functions of the Android framework that are then called by the system at specific points in time, e.g., when the user receives a text message or a button is clicked. Therefore, an Android application can be considered a "plugin into the Android framework" [17]. Each Android application consists of four different types of loosely coupled components, namely *activities*, *services*, *broadcast receivers*, and *content providers* [18].

Activity

An activity can be considered a single screen that presents a user interface and serves a specific purpose [19]. For instance, a social networking app might include an activity for displaying the user's news feed (`UserFeedActivity`), another one for composing posts (`CreatePostActivity`), and yet another one for viewing a user profile (`UserProfileActivity`). Activities can launch other activities within the same or a different app via intents, which we further discuss in Section 2.1.4. Typically, one activity is the *main activity*, which is the first screen presented when the app is launched.

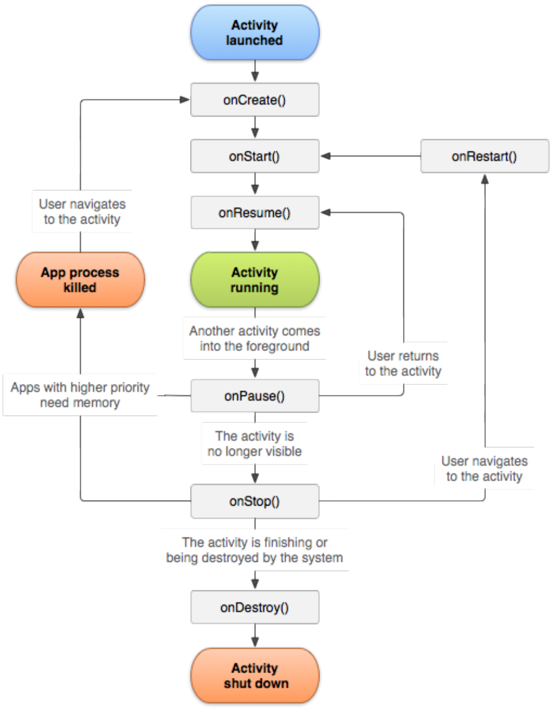

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Lifecycle of an Android activity [20].

Activities are subclasses of the `Activity` class and must override specific lifecycle methods to manage their state within the app. Each method handles one stage of an activity, from creation to destruction, including starting, resuming, pausing, and stopping an activity. The lifecycle of an activity is depicted in Figure 2.2.

Service

Unlike an activity, a service does not provide a user interface but is used to perform long-running operations in the background, e.g., music playback. Services are categorized into three types: *foreground services*, *background services*, and *bound services*. While a foreground service is used to run a task that is noticeable to the user and requires a notification to display while it runs, background services are used for tasks that are not noticeable to users. Due to system optimizations in recent Android versions, their use is restricted to ensure they do not consume excessive system resources. These two types of services are also commonly called *started services*. Bound services, on the other hand, are used when an application component wants to communicate with the service. The component then binds to the service and can interact with it through the `IBinder` interface [21].

Like activities, services have a lifecycle that includes several states, such as `onCreate`, `onStartCommand`, and `onBind`. The latter is limited to bound services.

Broadcast Receiver

Broadcast receivers in Android are components that respond to system-wide broadcast announcements or events. These events can range from system events, like a change in battery level or connectivity status, to custom user-defined events. Even when the application is not actively running, broadcast receivers can respond to incoming broadcast messages [18].

Content Provider

Lastly, a content provider in Android serves as the standard interface for managing an application's data storage and provides a data retrieval system through URIs. For instance, a content provider might manage access to a set of application settings. Other applications can request data from this provider using a specific URI, which the content provider uses to determine what data to return and whether the requester has the necessary permissions to access it [18].

2.1.3 Manifest File

Every Android application includes a manifest file (`AndroidManifest.xml`) that informs the Android system about its components, i.e., its activities, services, broadcast receivers, and content providers, alongside their properties. It also specifies the application's permissions, such as the ability to access the internet or the camera.

Additionally, the manifest file declares the hardware and software requirements necessary for the application, such as a microphone or camera requirement. It also plays a role in declaring so-called intent filters, which are used to define possible inter-component communication, i.e., specifying if and how different components can be started by others, as we describe in Section 2.1.4 in more detail [22].

A snippet of an Android manifest is given in Listing 2.1. In this example, the application declares two activities: `MainActivity` and `SecondActivity`. The intent filter for `MainActivity` specifies that it is the application's main activity, intended to be launched when the application is started from the device's launcher. The `exported` attribute controls whether the activity is accessible to other applications. Because `MainActivity` is exported, the intent filter has an effect, and the activity can be started externally, e.g., by other apps. In contrast, `SecondActivity` is not exported, meaning it can only be initiated internally by the application itself.

```
<application
    [...]>
    <activity
        android:name=".MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity
        android:name=".SecondActivity"
        android:exported="false">
    </activity>
</application>
```

Listing 2.1: Snippet of an Android manifest file.

2.1.4 Inter-Component Communication

In Android applications, components often need to communicate with and launch each other. Android facilitates this communication through a mechanism called *intents*. Intents can either be *explicit* or *implicit*.

Explicit Intent. An explicit intent directly specifies the component that should be started. For example, when the `UserFeedActivity` of a social network application, which displays the user's feed, wants to launch the `CreatePostActivity` to allow the user to create a new post, it would use an explicit intent to do so.

Implicit Intent. An implicit intent, on the other hand, does not specify the target component but rather a general action that should be performed. For example, suppose the `CreatePostActivity` wants to enable users to add a picture to their post. In that case, it might launch the camera app using an implicit intent with the action `android.media.action.IMAGE_CAPTURE`. The Android system then searches for suitable applications that contain an activity that can handle this type of action, based on the activity's intent filters. If there is such an application, the corresponding activity is launched. If multiple activities or applications can handle the intent, the user is prompted to select one. Note that doing so with an explicit intent would not trivially work, as the package name of the installed camera app may not be known.

Intents can also carry additional information in the form of *data* and *extras*. Data is a URI that specifies the actual data to operate on; e.g., in a web browser, the data would likely be the URL to open. Extras are key-value pairs that can pass additional information to the component [23].

An activity can launch the `SecondActivity` activity using an explicit intent, as we show in Listing 2.2. This intent also contains the extra `key` with the string value `value`.

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("key", "value")
startActivity(intent)
```

Listing 2.2: Creating an explicit intent to launch the `SecondActivity` activity.

2.2 In-App Browsers

Websites on mobile devices are traditionally accessed through standalone web browsers such as Google Chrome, Mozilla Firefox, or Microsoft Edge. However, they can also be viewed through in-app browsers (IABs). An IAB is a component embedded within a mobile application used to display and interact with web content. IABs are implemented on top of embedded browsers, which are responsible for rendering the web content and offering interaction capabilities between the app and the web, with both iOS and Android offering such components. Embedding IABs into mobile applications is a common practice, with popular apps such as Facebook, Instagram, TikTok, and X utilizing it.

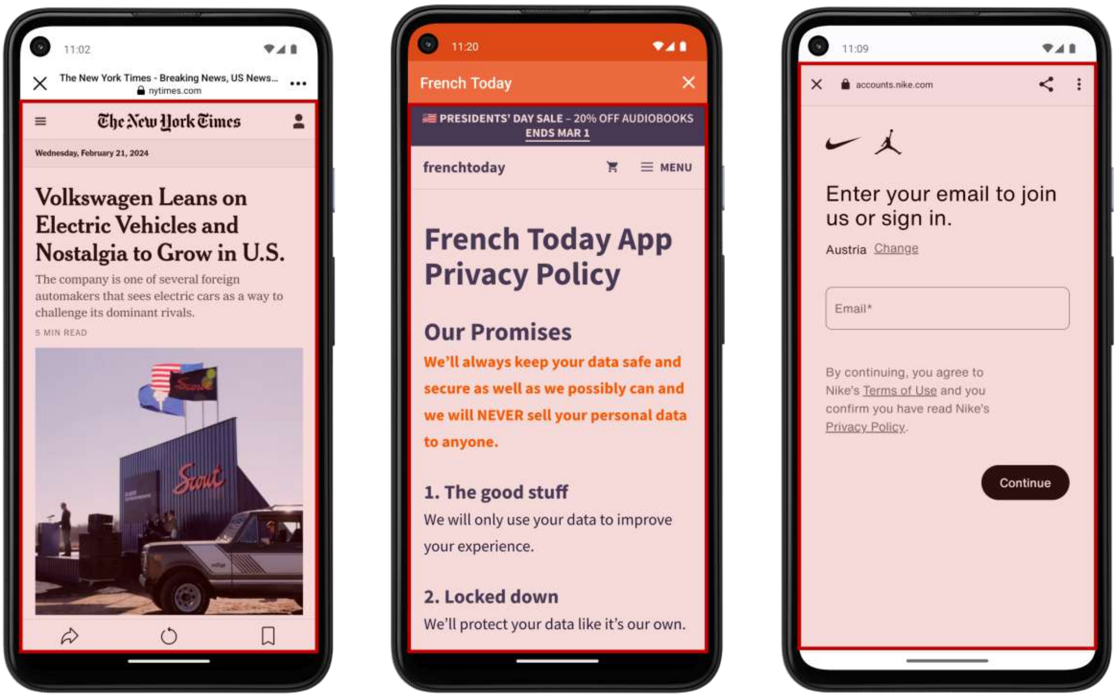
2.2.1 Android Embedded Browsers

On Android, various embedded browsers are available, ranging from system-provided to third-party options. The system-provided embedded browser, Android WebView, is widely used across applications to render web content within the application. In addition to WebView, developers can utilize the Custom Tab protocol, which all major browsers on Android support. Furthermore, developers can integrate third-party embedded browsers directly into the application's package. A visual comparison of a selected subset of embedded browsers on Android and their usage in real-world applications is shown in Figure 2.3.

Android WebView

Android WebView is a built-in system-level component that renders web content within Android applications. Unlike a full-fledged browser, WebView does not include any browser UI elements, such as a URL bar, navigation bar, or forward and backward buttons. Android WebView is based on the Chromium project but does not support all features available in Chrome, such as synchronizing browsing data or Web APIs such as the Push API [24] used to receive push notifications. Moreover, it does not share any data, such as cookies, caches, or service workers, with the Chrome browser [25].

Although WebView is a system-level component, it can be updated and installed as an APK file. The Android system provides the API to interact with the WebView, but the APK provides the actual implementation. This allows Google to update the WebView implementation independently of the Android system. For security reasons, on release builds of Android, the APK providing the concrete WebView implementation must be signed by Google [26].


11:02
The New York Times - Breaking News, US News...
nytimes.com
The New York Times
Wednesday, February 21, 2024
5 MIN READ
11:20
French Today
🇺🇸 PRESIDENTS' DAY SALE – 20% OFF AUDIOBOOKS ENDS MAR 1
frenchtoday
MENU
11:09
accounts.nike.com
Email*
Continue

Volkswagen Leans on Electric Vehicles and Nostalgia to Grow in U.S.

The company is one of several foreign automakers that sees electric cars as a way to challenge its dominant rivals.

French Today App Privacy Policy

Our Promises

We'll always keep your data safe and secure as well as we possibly can and we will NEVER sell your personal data to anyone.

1. The good stuff

We will only use your data to improve your experience.

2. Locked down

We'll protect your data like it's our own.

Enter your email to join us or sign in.

Austria Change

By continuing, you agree to Nike's Terms of Use and you confirm you have read Nike's Privacy Policy.

(a) Facebook provides a browser backed by Android WebView.

(b) French Today uses an Android WebView to display their privacy policy.

(c) Nike Training uses a Chrome Custom Tab for user authentication.

Figure 2.3: Usage of IABs in real-world applications on Android. The embedded browser is highlighted in red.

Applications can open a website in a Webview by calling its `loadUrl` method. Listing 2.3 shows how to open `https://www.example.com` in a WebView. Other methods also exist, such as `loadData` and `loadDataWithBaseURL` that allow loading data from a string, and `postUrl` that allows loading a website with a `POST` request.

```
val wv: WebView = findViewById(R.id.webview);
wv.loadUrl("https://www.example.com");
```

Listing 2.3: Launching `https://www.example.com` in an Android WebView.

Apart from simply displaying the web content, WebView also offers some interaction between the app and the web content, both in terms of app-to-web interactions and web-to-app interactions.

App-to-web interactions. App-to-web interactions allow the application to engage with the web content. For example, the app can inject JavaScript code into the website being displayed by either calling the `loadUrl` method and passing the JavaScript code as a string preceded by `javascript:` or by using the `evaluateJavascript` method of the

WebView [27] class. This JavaScript code is then executed in the context of the website. The WebView class also provides other APIs that can be used for app-to-web interactions, such as the getCookie and setCookie methods of the CookieManager [28] class, which allows the application to access and manipulate cookies stored for a website. Applications can also save the whole page as a web archive using the saveWebArchive method of the WebView class. A web archive is an MHTML file that contains all resources of a webpage, such as images, CSS, and JavaScript files, in a single file. WebViews also provide additional APIs for such interactions, and the interested reader is referred to the official documentation for a comprehensive list [27].

Web-to-app interactions. In web-to-app interaction, the web content can interact with the application and call native Java or Kotlin functions defined in the application. This is achieved by exposing these functions to the web content through the addJavascriptInterface method of the WebView class and specifying the target class and the interface's name. All functions annotated with the @JavascriptInterface annotation in the target class are then exposed to the web content. For example, if a function sendMessage should be exposed to the web content that sends a message to a given number, the code shown in Listing 2.4 would be used. The JavaScript code running on the website can then call this function by executing window.JSBridge.call(1234567, "Hello World").

```
webView.addJavascriptInterface(new JSBridge(), "JSBridge");

class JSBridge {
    @JavascriptInterface
    fun sendMessage(number: Int, message: String) {
        // Send the message to the given number
    }
}
```

Listing 2.4: Exposing the sendMessage function to the web content.

WebView Hooks. In addition to app-to-web interaction and app-to-web interaction APIs, WebViews provide hooks that allow applications to intercept and respond to events occurring within the WebView. For example, the WebViewClient [29] class provides a method called shouldOverrideUrlLoading that is called whenever a new URL is about to be loaded in the WebView. The application can override this method to intercept the URL and perform custom actions, such as opening the URL in the system browser instead of the WebView. The WebChromeClient [30] class supports similar hooks, such as the onConsoleMessage method, which is called whenever a JavaScript console message is logged in the WebView.

Android Custom Tabs

An Android Custom Tab is, strictly speaking, not a single component offered by Android but a protocol implemented by browser vendors. Unlike Android WebViews, Custom Tabs are provided by the browsers, e.g., Google Chrome or Mozilla Firefox. When the embedding application opens a Custom Tab, an activity of the underlying browser is launched. This activity includes a browser UI, including a navigation and URL bar [31].

Since the Custom Tab is an activity of the underlying browser, the component and the browser are tightly coupled, and their state, such as cookies, is shared. Therefore, Custom Tabs are especially useful in Single Sign On (SSO) flows, where the user is already logged into the browser [32].

For security reasons, Custom Tabs do not support the same level of interaction between the app and the web content as WebViews do. For example, the app cannot inject JavaScript code into the website or access the website's cookies. Custom Tabs also do not allow the web content to call functions defined in the app. Nevertheless, Custom Tabs offer so-called callbacks and engagement signals that the app can use to track the user's interaction with the web content [31]. Because of Custom Tab's lack of app-to-web interaction, we do not consider them for the rest of this thesis.

Third-Party Embedded Browsers

Android also allows the inclusion of third-party libraries that provide embedded browsers. Unfortunately, there is no complete list of such libraries available, and our research indicates that there are very few such libraries available, among which are Mozilla GeckoView [33] and the discontinued Crosswalk Project [34], which we both do not consider in this thesis.

2.2.2 iOS Embedded Browsers

In iOS applications, web content can only be rendered by the system-level components `WKWebView` or `SFSafariViewController` [35]. Although this thesis focuses on Android, we provide a brief overview of these iOS components for completeness.

WKWebView

`WKWebView` is the counterpart to Android's WebView. Like Android WebView, `WKWebView` displays web content without any browser UI elements and supports a comparable level of interaction between the app and the web content. Apps can inject JavaScript into the webpage using functions such as `evaluateJavaScript`, `callAsyncJavaScript`, and `addUserScript`. Additionally, functions defined in the app can be exposed to the web content through the `WKUserContentController` class [36].

SFSafariViewController

`SFSafariViewController`, equivalent to Android Custom Tabs, is provided by Safari. Unlike Custom Tabs, `SFSafariViewController` does not share state with the underlying browser. This component restricts interactions between the app and the web content, limiting its functionality compared to `WKWebView`. However, for authentication purposes, developers can utilize the `ASWebAuthenticationSession` class to open an authentication website within an `SFSafariViewController`, allowing the app to receive an authentication token. When `ASWebAuthenticationSession` is used, the component can be configured to share state with the underlying browser in a controlled manner [37].

2.3 Android App Analysis

Program analysis techniques are used in many fields, such as for assessing the security of applications, finding bugs, verifying program correctness, code optimization, and supporting program development [38]. The field of program analysis can broadly be categorized into two main types: static analysis and dynamic analysis.

2.3.1 Static App Analysis

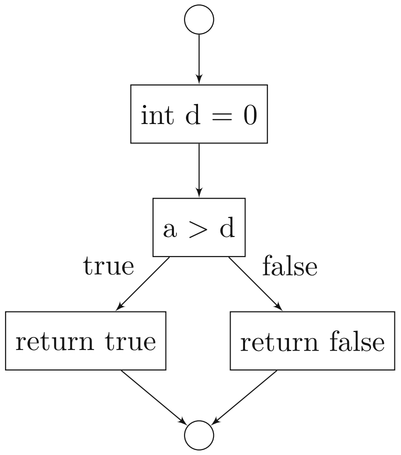
Static analysis involves examining a program's behavior by inspecting its source code or bytecode without executing the program. This type of analysis often utilizes abstract representations such as call graphs and control flow graphs to deduce the program's behavior. Generally, static app analysis is divided into two categories: *intra-procedural* and *inter-procedural* analysis. Intra-procedural analysis examines the behavior of a single method in isolation, typically using a *control flow graph* (CFG) for representation. On the other hand, inter-procedural analysis crosses method boundaries to analyze the program as a whole, often employing a *call graph* to connect methods [39].

Control Flow Graph

A control flow graph (CFG) is a directed graph that models the flow of control within a method. Each node in the CFG represents a basic block, i.e., a sequence of instructions executed sequentially without branching and interruption. Edges between the nodes represent possible paths that execution might follow, making CFGs particularly useful for understanding the branching mechanics within methods, such as those introduced by conditional statements.

Figure 2.4 illustrates a CFG for a simple Java method named `isPositive`, which checks if a parameter is greater than zero. The graph captures all potential execution paths, including those conditioned by the if-else statement. Thus, a CFG is a useful representation of conditional statements and their impact on a program.

```
public void isPositive(int a) {
    int d = 0;
    if (a > d)
        return true;
    return false;
}
```



(a) Java method `isPositive`.

(b) Control flow graph of `isPositive`.

Figure 2.4: Java method and its control flow graph.

Call Graph

Object-oriented programming languages like Java and Kotlin are built around classes as the general concept. These classes contain methods that can call other methods and fields that represent the object. Such programs usually have a single entry point, i.e., a `main` method, from which the program's execution starts. Analyzing a program's *main* method, listing all methods it calls, and then iteratively analyzing these methods yields the construction of a *call graph* and is a form of inter-procedural analysis. A call graph consists of nodes representing methods and directed edges representing method calls. Figure 2.5 shows an example of a call graph of a simple Java program.

A call graph can come in different precision levels that also impact the running time of the analysis. Call graphs can be, for example, *flow-sensitive*, *path-sensitive*, or *context-sensitive*. We will not detail these different precision levels but refer the interested reader to Li et al. [39] for a detailed discussion.

2.3.2 Dynamic App Analysis

Dynamic analysis examines or modifies an application's behavior by actually executing the application. This method offers a key advantage over static analysis: it captures the app's actual behavior, making it particularly effective for tasks like malware analysis where understanding real-world behavior is critical. However, dynamic analysis is less scalable than static analysis due to the necessity of running the app, which can be resource-intensive and time-consuming.

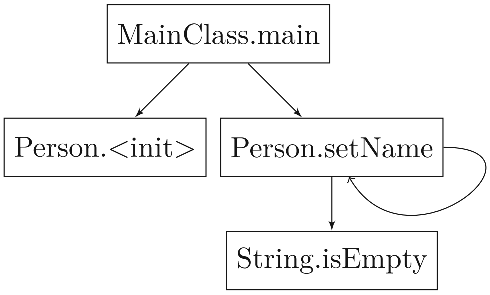
The monitoring and manipulation can be external to the device, such as intercepting network traffic using tools such as Wireshark [40], or internal, such as intercepting and modifying API calls and function parameters. For the latter, common approaches include modifying the app's execution environment. This can be achieved by modifying the

```
public class MainClass {
    public static void main(String
    [] args) {
        Person person = new Person
    ();
        person.setName("John Doe")
    ;
    }
}

public class Person {
    public String name;

    public void setName(String
    name) {
        if (!name.isEmpty()) {
            this.name = name;
        } else {
            setName("unknown")
        }
    }
}
```

(a) Java program.



(b) Call graph of the program.

Figure 2.5: Java program and its call graph.

underlying system or employing runtime instrumentation. In runtime instrumentation, an app is modified dynamically during its execution. This can be done, for example, by hooking functions and overwriting them with user-defined code, such as with the popular dynamic instrumentation tool Frida [41].

CHAPTER 3

Related Work

The related work for this thesis can be categorized into two areas: the security of IABs on mobile devices and targeted execution on Android. We provide a brief overview of these topics in the following sections.

3.1 Security of IABs

3.1.1 Web-To-App Interaction

Previous research on the security of In-App Browsers (IABs) has explored various vulnerabilities and the associated risks of app-to-web and web-to-app interactions in Android WebViews. Luo et al. [4] analyzed the dual threats posed by malicious web pages interacting with benign apps and malicious apps exploiting benign web pages. Specifically, they highlighted the dangers of allowing JavaScript within WebViews to call Java or Kotlin functions, potentially leading to unauthorized access to sensitive user data such as location information. They also discussed the risks associated with the injection of malicious JavaScript code into WebViews and the possibilities for event sniffing through the use of WebView hooks. Additionally, they conducted a small-scale evaluation of 200 popular Android applications, revealing that WebViews were utilized in 113 of these applications.

Neugschwandtner et al. [5] further focused on the security implications of Java code invocation from JavaScript in Android WebViews, such as data exfiltration under a man-in-the-middle attack scenario. Their analysis of over 250,000 Android applications revealed that 30% had JavaScript interfaces allowing JavaScript to Java communication, with 10% vulnerable to potential exploitation.

Chin et al. [6], Rizzo et al. [7], and El-Zawawy et al. [11] provided further insights into app-to-web interactions, proposing tools for the automatic detection of attacks leveraging these vulnerabilities. Their methodologies relied on static analysis to identify exploitable interfaces between web content and native app code.

Additionally, Zhang et al. [8] examined the security within “super-apps” like WeChat and TikTok, which host multiple sub-apps within a single application using WebViews. They discovered access control vulnerabilities across all 47 analyzed super-apps, enabling unprivileged sub-apps to exploit privileged APIs.

3.1.2 App-To-Web Interaction

Expanding beyond the Android ecosystem, Krause [13] analyzed the usage of IABs in a small set of popular applications (Instagram, Facebook, TikTok, Facebook Messenger, Amazon, Snapchat, and Robinhood) on iOS. The methodology of the analysis consists of manually interacting with the applications and loading a custom web page (`inappbrowser.com`) in their IAB, which records JavaScript injections. The author found that out of the seven analyzed applications, 5 inject JavaScript code, while 4 modify the web content. The injected JavaScript code encompassed code that could be used to monitor user inputs such as keyboard input and user taps. Unlike our work, Krause relied on manual analysis of a handful of applications, which is time-consuming and does not scale. Also, he focused on the iOS ecosystem. Furthermore, the analysis is limited to detecting JavaScript injection and did not consider other types of interactions between the app and the web content.

Most similar to our work, Zhang et al. [14] focused on the usage of web resource manipulation APIs of Android WebView, i.e., APIs that can be used to modify the web content displayed in a WebView, such as `evaluateJavascript`. To do so, they conducted a large-scale analysis of 80,694 Android applications. Even though they targeted a similar problem as ours, their methodology differs. They relied on purely static analysis of the application’s code to detect how the APIs are used by building an inter-procedural control flow graph, which they traverse to check whether specific APIs are used. To retrieve the parameter values of the API calls, they create a backward slice of all instructions necessary to compute the parameter values, thus being able to reconstruct the value of the parameter. Compared to our approach, this has some severe limitations. To begin with, static analysis is known to be imprecise and prone to false positives, i.e., static analysis flags a code as reachable even though it is not. Our conservative targeted execution approach, however, is designed to avoid false positives. Furthermore, only statically analyzing an application’s code does not capture the actual behavior of the application, i.e., how the application interacts with the web content at runtime. JavaScript code could be dynamically fetched from a remote server before it is injected into the web content, or the JavaScript code could be dynamically generated at runtime. A methodology purely based on static analysis would not capture these cases, unlike our approach. Lastly, Zhang et al.’s analysis dates back to 2018, and the Android ecosystem has evolved since then. Therefore, it is important to question the relevance of their findings in the current Android landscape. Since they did not open-source and publish their tool, we cannot verify their findings or compare their results with ours.

3.1.3 Non-Interaction-Related Security Risks

Yang et al. [9] explored security issues using iframes and popups within Android WebViews. Their research uncovered three novel attack vectors that exploit the lack of proper sandboxing and isolation in WebViews. These vulnerabilities could allow a malicious website loaded in an iframe to secretly redirect the main frame to malicious pages, secretly access Java functions from JavaScript code, and overlay legitimate content with phishing content. They also introduced DCV-Hunter, a static analysis tool that scans Android applications for these vulnerabilities, and found high-profile applications like Facebook and Instagram vulnerable.

Tiwari et al. [42] explored the potential for browser fingerprinting in applications that use IABs. Their findings indicate that WebView fingerprints were more unique than those of standalone browsers like Chrome. They also noted that the fingerprints of IABs often contain device-specific and user-specific information, allowing them to fingerprint a user uniquely.

Furthermore, Zhang et al. [43] focused on the usability of IAB interfaces and discovered that several IABs fail to provide users with enough information about the website that is opened, including information about unsafe operations during browsing, e.g., when the website is served over an unencrypted connection.

3.2 Targeted and Forced Execution on Android

Research on driving the execution of an application to a specific location on Android can broadly be divided into two categories: forced execution and targeted execution.

Forced execution on Android has been mainly used to detect malware. Tang et al. [44] proposed Dual-Force, a forced execution tool that exposes malicious behavior in WebView malware. WebView malware is malware that, among other malware techniques, also uses the WebView component to conceal malicious behavior. For example, WebView malware might not invoke a function using Java but instead use the JavaScript bridge to call a malicious function via the web content, thus concealing the malicious behavior. To detect malicious behavior in such malware, Dual-Force uses forced execution to explore the application and the web content it loads. Similarly to our approach, the tool consists of a static and a dynamic analysis phase. The static analysis phase is used to build the call and control flow graphs and determine where Java and JavaScript code interact using a WebView. However, unlike our approach, they employed a forced execution approach for the dynamic part. That means they forced specific branch conditions and, therefore, forced the execution of specific code paths by instrumentation. While this allowed them to reach a high code coverage, it also has the downside of possibly reaching code that is not reachable under normal circumstances, introducing false positives. Furthermore, the tool is not open-source, and the exact internal workings are unknown to us.

Wang et al. [45] targeted a similar problem with Droid-AntiRM, a forced execution tool that aims to uncover malicious behavior in Android applications by circumventing anti-analysis techniques, i.e., techniques to avoid detection by malware analysis tools. Such techniques comprise conditional switches that, for instance, check whether the device that currently executes the app is an emulator. Following a hybrid analysis approach, after statically constructing a call graph and control flow graph and determining potential anti-analysis condition statements, they instrument the application and then dynamically force those condition statements to be taken.

Ares, proposed by Bello and Pistoia [46], takes a very similar approach to tackle the same problem. They first use dataflow analysis to find evasion points, i.e., points where anti-analysis techniques take place. For each evasion point, the tool instruments the Java bytecode and flips the condition of the evasion point. The APK is also instrumented with log calls after each evasion point to log whether a specific branch was taken. A new APK is generated for every combination of forced conditions. Each APK is installed on an actual device. Monkey, a tool that simulates random user inputs, is used to interact with the application. Furthermore, they used a modified Android that logs when sensitive functions are called, such as when an SMS is sent or code is dynamically loaded. Like other tools on forced execution, Ares also has the downside of possibly reaching code that is not reachable under normal circumstances, introducing false positives.

Wong and Li [47] presented IntelliDroid, a tool that drives the execution of an application to a target location. Similar to our tool, it employs a hybrid analysis methodology consisting of static and dynamic analysis. Unlike previous tools, IntelliDroid is not based on forced execution. Instead, IntelliDroid uses constraint solving to generate inputs that satisfy conditions that are required to reach a target location, e.g., IntelliDroid simulates an incoming SMS of a specific format to satisfy a specific branch. To do so, the static part first generates a call graph and finds the paths to the target location. The constraints that need to be satisfied for each path are then extracted. At runtime, IntelliDroid executes the application and uses a constraint solver to solve the constraints on the branch conditions. The tool then generates the inputs that satisfy the constraints and injects them into the application. While the tool is open-sourced, it, unfortunately, targets an old version of Android, and a significant amount of engineering effort would be required to adapt the tool to the current Android ecosystem.

Wong and Li [48] extended this work with a tool called Tiro and added support for UI interactions. Their approach targets the problem of de-obfuscation of Android applications by driving the execution of the application to a location where possible obfuscation happens, e.g., a call to a native function. Unfortunately, only a small portion of the tool was open-sourced.

CHAPTER 4

IABInspect

In this chapter, we present IABInspect, our prototype tool to automatically analyze app-to-web interactions within IABs in Android applications. IABInspect employs a hybrid analysis approach, combining both static and dynamic analysis techniques that reduce the high false-positive rate of purely static analysis by dynamically triggering the execution of IABs and recording the interactions between the application and the web. This chapter begins with an overview of our approach, followed by a detailed discussion of the tool's components and operational steps.

4.1 Overview of the Approach

As discussed in Chapter 3, previous work only used static analysis to examine app-to-web interactions in IABs. Although static analysis gives a high-level overview of the app-to-web interactions, its precision is limited, often leading to a significant rate of false positives, i.e., identifying code that, although present, is never executed. This is especially true for IABs since they are frequently embedded in third-party libraries. Often, only a fraction of the included library is used. Additionally, static analysis cannot accurately trace dynamic interactions, such as JavaScript code executions that occur at runtime and are fetched from remote servers.

While effective at capturing runtime interactions, purely dynamic analysis also has drawbacks. It can be laborious, especially when involving manual UI exploration, and does not scale well and is time-consuming when using automated exploration.

To address these challenges, we develop IABInspect, which consists of both static and dynamic analysis phases. The tool first uses static analysis to construct a blueprint of the application, identifying potential paths from the application launch to an IAB launch call. The application is then instrumented with logging statements that facilitate monitoring during the dynamic analysis phase.

The dynamic component of IABInspect executes the instrumented application, steering it along the predefined paths to trigger IAB launches. This phase not only verifies the actual use of the code paths but also captures the live interactions between the app and the web content. An essential feature of IABInspect is its custom Android WebView implementation, which we instruct Android to use. It can record the app-to-web interactions whenever the dynamic executor triggers the launching of an IAB. This custom WebView is furthermore modified to always load a controlled website that is able to capture injected JavaScript code. In the rest of this thises, we refer to this website as the "hooked website". Figure 4.1 illustrates the high-level workflow of IABInspect.

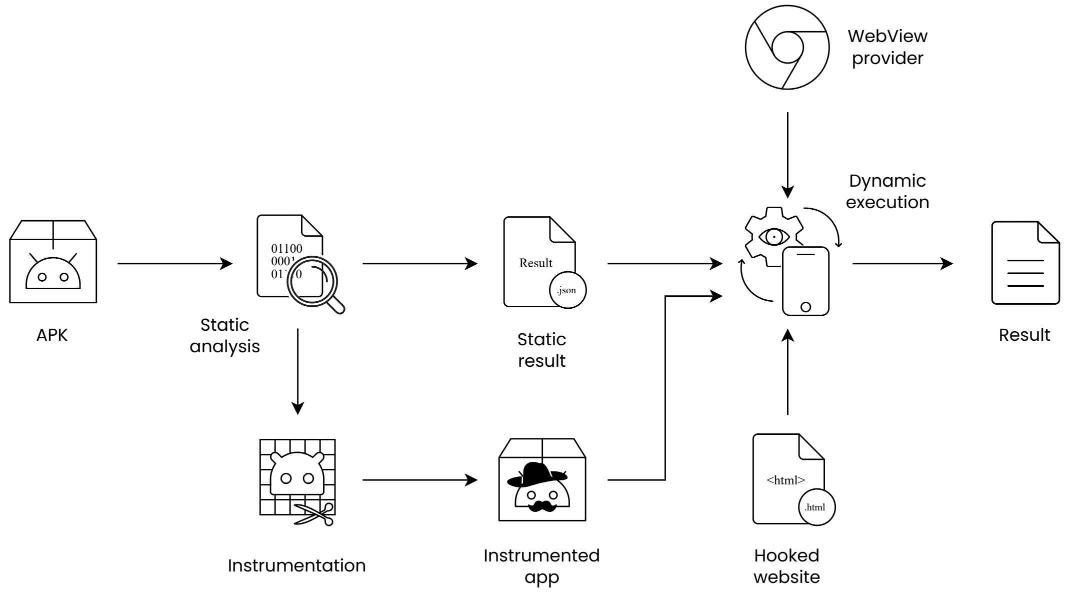


Figure 4.1: Workflow and components of IABInspect.

4.2 Threat Model

As our threat model, we consider an *App Attacker using an IAB*, where a *Potentially Unwanted Application* (PUA) loads a benign website in an IAB. The attacker's objectives can vary, aiming either to compromise the confidentiality of user data, such as by stealing cookies or passwords from a user on a third-party website or to target the website's integrity, such as by executing unauthorized actions in it. A practical example of such a PUA is an application requiring users to perform a login action at a third-party service in an IAB, opening the door for the application to steal the user's credentials.

There are two primary ways in which a PUA can become malicious or privacy-invasive:

Malicious or privacy-invasive application. In this scenario, the application itself is malicious or privacy-invasive and operated directly by the attacker, masquerading as a legitimate and useful application to convince users to install it. This model aligns with the threat model proposed by Luo et al. [4].

Benign application with malicious or privacy-invasive library. Here, the application itself is benign but unknowingly embeds a malicious or privacy-invasive library that performs unwanted and potentially harmful actions on the IAB [49, 50].

4.3 Static Analysis

The static analyzer of IABInspect is responsible for building the blueprint of the application and computing paths from an application's entry point to an IAB launch call, i.e., a code segment that launches the IAB. The dynamic executor then uses these to steer the application toward the IAB launch calls.

Initially, the analyzer preprocesses the application, followed by an analysis of the application's manifest file to understand the components of the application. It then generates a call graph, which is further enriched by analyzing the application's inter-component communication (ICC). The call graph is extended with ICC edges and custom entry methods designed for this analysis. Finally, the analyzer computes the paths leading from an entry method to the IAB launch calls using the constructed and enhanced call graph. A schematic overview of the static analyzer is illustrated in Figure 4.2.

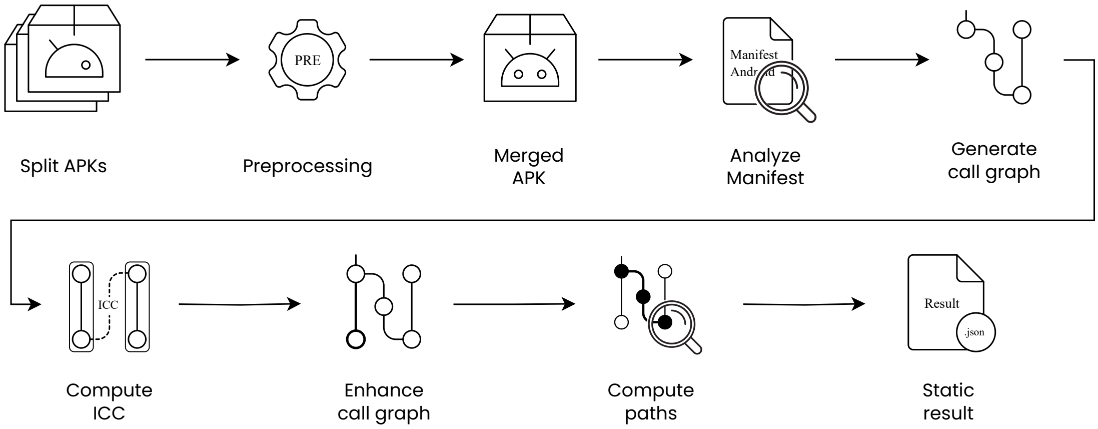


Figure 4.2: Workflow of the static analyzer of IABInspect.

4.3.1 Preprocessing

Typically, Android applications are distributed as APK files, which are compressed archives containing all the app's code and necessary resources. However, Google now requires that developers publish their apps as Android App Bundles (AAB) on the Google Play Store. This format allows the Play Store to handle the APK generation and signing process, optimizing resource delivery based on the user's device [51]. As a result, when a user downloads an app from the Play Store, they only receive the code and resources necessary for their specific device, such as resource files for different screen sizes, languages, or processor architectures. These resources are not served as a whole but delivered as split APKs, tailored to meet the particular needs of the user's device [52]. For example, the application `abmm.heckyl.com` may be divided into the following split APK files:

- `abmm.heckyl.com.apk` is the base APK,
- `abmm.heckyl.com.split.config.de.apk` contains the resource files for the German language, and
- `abmm.heckyl.com.split.config.xxhdpi.apk` contains the resource files for the xxhdpi screen size.

Given our need to analyze the complete application, including all code stored in split APKs, and considering that the application is only fully functional when all split APKs are present, we must merge these split APKs into a single APK. To do so, we use the open-source tool APKEditor [53], leaving us with a single APK file.

4.3.2 Manifest Analysis

The next step is the analysis of the application's manifest file. The manifest file contains a list of the application's components and intent filters, which we later feed into the dynamic executor. Additionally, the manifest file also declares which components are exported, i.e., can be started from outside of the app, which we require in a later step to compute the call paths accurately.

For parsing the manifest file, we rely on Soot, a state-of-the-art framework designed for analyzing Java-based applications [54].

4.3.3 Call Graph Generation

We then generate a call graph of the application. This process begins by converting the application's Dalvik bytecode into the Jimple intermediary representation (IR) using the Soot framework. Given that both Java and Kotlin applications compile down to Dalvik bytecode, Soot can analyze applications written in either language. Although generating a call graph may initially seem straightforward, the complexity of Android applications presents significant challenges.

As previously discussed in Section 2.1, Android applications do not have a single `main` method that is called at startup. Instead, these applications can have multiple entry points, such as various activities through which users can enter the application. Furthermore, Android apps are highly event-driven. Components like activities implement lifecycle methods that the Android framework invokes at specific times, such as when an activity starts or stops. Other callbacks are called in response to user interactions, e.g., when a button is clicked.

Accurately modeling these idiosyncrasies of Android applications is crucial for creating a precise call graph that can guide the dynamic executor to the IAB launch calls. Fortunately, FlowDroid [17], a state-of-the-art static taint analysis tool for Android applications, can construct a callgraph that models these Android-specific behaviors. While we do not use FlowDroid's taint analysis capabilities, we use its capabilities to determine the application's entry methods and callbacks.

After parsing the application's manifest file to determine the app's components, i.e., activities, services, providers, and broadcast receivers defined in the application, FlowDroid constructs a dummy main method that serves as an artificial single entry point into the application. It also constructs dummy methods for each component of the application. These component-specific dummy methods contain edges to the actual lifecycle methods of the component, such as `onCreate` or `onStart`, and to callback methods in the component like `onClick`. Flowdroid adds an edge from the single dummy main method to the dummy methods of all components. FlowDroid considers both exported and non-exported components to be reachable from the dummy main method.

In each iteration, it generates a call graph that includes all methods reachable from the dummy main method and searches for callback methods in the newly reachable code. For example, an application might register a click callback by overriding the `onClick` method or by defining it in the layout files. FlowDroid also parses these layout files, integrating any discovered callbacks into the component-specific dummy methods before reconstructing the call graph. This iterative process continues until no further changes are detected in the call graph, resulting in the preliminary call graph on which we base our static analysis.

4.3.4 ICC Computation

Android components may communicate with each other using intents. For example, an activity `HomeActivity` might start another activity `NewsFeedActivity` by sending an intent. This type of communication is known as *Inter-Component Communication* (ICC). In our dataset that we propose in Chapter 6, applications are composed of, on average, 43.6 activities. Incorporating ICC into our analysis is thus crucial for comprehensively understanding an application's behavior.

Determining ICCs in Android applications is challenging due to the dynamic nature of intents. Unlike intent filters, which are statically defined in the manifest file, intents are typically created at runtime, and their target components may even be influenced by user input. Although FlowDroid supports ICC analysis through its integration with ICCTA, it cannot handle modern applications [55]. Instead, we rely on ICCBot [56].

ICCBot is a tool that uses static analysis to determine ICCs in Android applications. Apart from tracking the source and target components of ICCs, it can also determine the intent extras and intent data that a component expects. While we use the former to enhance our call graph, as we discuss in the next section, we save the latter and feed it to the dynamic executor.

4.3.5 Call Graph Enhancement

We enhance the callgraph constructed by FlowDroid in two ways. First, we add ICC edges to the callgraph to be able to reason about the interactions between different components of the application. Second, we add our own entry methods to the callgraph. A graphical representation of the enhancements is shown in Figure 4.3 based on an application containing two activities, `ExportedActivity` and `NotExportedActivity`, as well as a broadcast receiver, a service, and a content provider.

ICC Edges. As output, ICCBot provides a list of ICC calls from a source statement to a target component. For each of these ICC calls, we add extend the call graph by adding and edge. For example, ICCBot may determine that a call to `startActivity` in function `onCreate` of `ExportedActivity` starts the `NotExportedActivity`. In this case, we add an edge from the `startActivity` statement in `ExportedActivity.onCreate` to the dummy method generated by FlowDroid of the `NotExportedActivity`, thus encoding the ICC in the callgraph.

Custom Entry Methods. FlowDroid's `dummyMainMethod` is a good starting point to model the entry points of an Android application. However, it also contains edges to activities that are not exported, i.e., activities that are not accessible from other applications and cannot be directly started from the outside, as well as content providers, broadcast receivers, and services, which IABInspect currently does not support. We add custom entry methods to the call graph to distinguish between exported activities that can be opened from the outside and non-exported activities that we need for the call path computation. In specific, we add the `EntryClass` class that contains the `resolvedEntry` and `partiallyResolvedEntry` methods to the call graph. The `resolvedEntry` method contains edges to the dummy methods of all exported activities, while the `partiallyResolvedEntry` method contains edges to the dummy methods of non-exported activities.

4.3.6 Call Path Computation

After constructing the call graph that includes callbacks and ICC edges, we proceed by iterating through all Jimple statements of the application and identifying invocations of methods that can be used to launch IABs. Each detected launch call is assigned a unique ID. This ID is used in the dynamic analysis phase to identify the IAB launch call.

A complete list of methods we consider to be used to launch IABs is provided in Table 4.1. Note that we consider all relevant methods that can be used to launch WebViews as per the WebView documentation [27]. While our current analysis is limited to these methods,

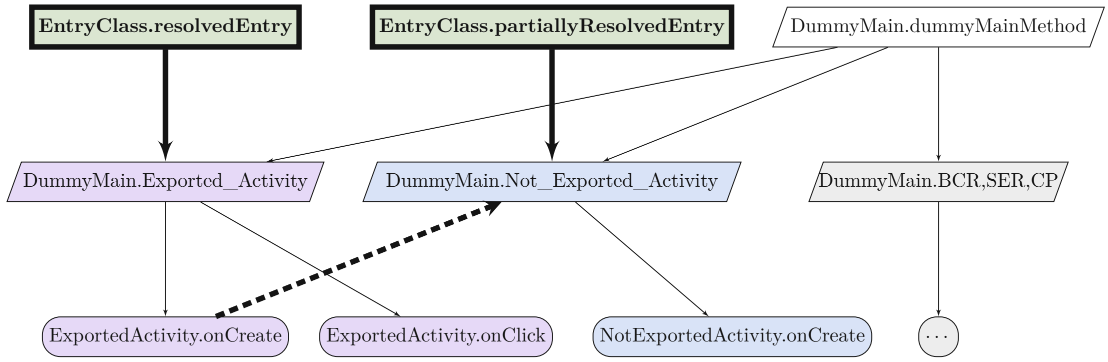

EntryClass.resolvedEntry
EntryClass.partiallyResolvedEntry
DummyMain.dummyMainMethod
DummyMain.Exported_Activity
DummyMain.Not_Exported_Activity
DummyMain.BCR,SER,CP
ExportedActivity.onCreate
ExportedActivity.onClick
NotExportedActivity.onCreate
...

Figure 4.3: Callgraph enhancements with custom entry methods and ICC edges.
The colors indicate method affiliation: 🟢 for newly added methods, 🟣 for classes related to `ExportedActivity`, 🔵 for classes related to `NotExportedActivity`, ⚪ for other classes.
Shapes represent the method source: □ for newly added methods, ▱ for methods generated by FlowDroid, ○ for application methods.
Thick lines represent newly added edges.

IAB Method	**Description**
`WebView.loadUrl(String)`	Loads the given URL in the WebView.
`WebView.loadUrl(String, Map)`	Loads the given URL in the WebView with additional HTTP request headers.
`WebView.loadData(String, String, String)`	Loads the given data (e.g., HTML content) in the WebView.
`WebView.loadDataWithBaseURL(String, String, String, String, String)`	Loads the given data in the WebView and allows to specify a base URL.
`WebView.postUrl(String, byte[])`	Loads the given URL in the WebView using a `POST` request.

Table 4.1: Considered target functions used to launch IABs.

IABInspect is designed with flexibility and can be extended to include additional methods in the future.

Once we have identified the IAB launch calls, we approach the pathfinding as a graph search problem, backtracing the call paths from the target method to an entry point. We define two types of entry points that lead to two different types of paths.

Exported activities leading to a resolved path. In case we find a path to the `resolvedEntry` method from an IAB launch call, we refer to this path as a *resolved path*. These paths originate from an actual entry point of the application, i.e., an exported activity, and may involve ICC edges. Therefore, IABs launched by following a resolved paths are guarenteed to be reachable in a real-world scenario.

Non-exported activities leading to a partially resolved path. In contrast, we may find paths that start from the `partiallyResolvedEntry` method. This is the case if the ICC analysis has not been able to resolve and find all ICCs of the application. We call such paths *partially resolved paths*. Partially resolved paths serve as shortcuts to reach the target IAB launch call, initiating from a point within the application that, while not directly accessible from the outside, represents an entry to a self-contained segment of the app. Since an application can declare activities in its manifest file that are never used in practice, partially resolved paths are not guaranteed to be reachable in real-world scenarios.

Starting from an IAB launch call, we then backtrace the call path in a breath-first search manner. Using a breadth-first search has the advantage that it allows us to find the shortest path to an entry method. IABInspect is configured to record up to 100 fully resolved paths and 100 partially resolved paths to each target call site.

4.4 Instrumentation

In the instrumentation phase, we augment the application with information necessary for the dynamic executor to accurately navigate to the IAB launch calls. The instrumentation process fulfills two primary functions: (1) enabling the identification of IAB launch calls and (2) enabling tracking of the application's execution. Similar to the static analysis phase, the instrumentation process relies on Soot.

4.4.1 Enabling the Identification of IAB Launch Calls

During the application's execution, the dynamic executor must be able to monitor the app's current state by tracking which functions have been called. As we will discuss in Section 4.5.3, the dynamic executor utilizes Frida [41], a dynamic instrumentation tool that allows method invocations to be intercepted. This would allow the dynamic executor to intercept calls to the IAB launch functions, such as `WebView.loadUrl`, and be notified whenever these functions are called. By dynamically instrumenting the function to call the `java.lang.Thread.currentThread().getStackTrace()` method, it is possible to determine the source method from which the IAB launch call was made. However, this approach has limitations in specific scenarios. For example, if one method contains multiple IAB launch calls, we can only determine the method from which the IAB launch call was made, but not the exact call site. Even though the `StackTraceElement` returned by the `currentThread` call contains a line number, it does not correspond to the line number in the source code.

To address this limitation, we instrument the application to insert a `android.util.Log` call immediately before each IAB launch call. Each call is assigned a unique identifier, which is included in the log statement and can be accessed by the dynamic executor.

Additionally, to ensure that information about the IAB launch call is passed to both the custom WebView provider and the hooked website, we statically instrument the application to append the identifier as a URL fragment[1] to the URL passed in the IAB call. If the URL already contains a fragment, the identifier is appended to the existing fragment. This approach allows the dynamic executor to identify the IAB launch call and the corresponding call site, even if the call is made from a method that contains multiple IAB launch calls.

4.4.2 Enabling Function Tracking

While Frida's hooking functionality theoretically enables tracking function calls by hooking into application functions and logging their execution, this approach is not practical for real-world-sized applications. Frida can only hook a limited number of functions before the application crashes. In our experiments, Frida could hook only about 150 functions before the application crashed, a limitation that is insufficient for handling applications containing thousands of functions.

To address this limitation, we instrument the application by adding a log statement at the beginning of each method on one of the identified paths from an entry method to an IAB launch call. Each log statement includes the method's signature, allowing the dynamic executor to track the execution without the stability issues associated with excessive hooking.

4.4.3 Changes to Soot

Soot throws an exception when attempting to build Android applications that specify a `minSdk` version lower than 22 due to the lack of multi-dex support for these older API versions. Since Android API versions 22 and above include built-in support for multi-dex, applications targeting these versions can run on devices that support them without issues. To allow Soot to build applications with a lower `minSdk` version, we introduce a new flag, `ignore_dex_overflow`. This flag allows Soot to ignore the multi-dex overflow error and successfully build applications that would otherwise fail due to this limitation.

4.5 Dynamic Analysis

Following the static analysis, which constructs a blueprint of the application and identifies paths from an entry method to the IAB launch calls, the instrumented application is executed to record the actual app-to-web interactions during the dynamic analysis phase. This phase involves three main components, as illustrated in Figure 4.4: the *Custom WebView Provider* used to record the IAB API calls of the application, the *Hooked Website*, a controlled site used to record JavaScript injections performed by the application into the web content which is forcibly loaded into the IAB, and the *Dynamic Executor*, which is responsible for monitoring the execution of the application and steering it towards the

[1] A URL fragment is the part of a URL that follows the # symbol [57]. For example, in the URL `example.com/path#fragment`, `fragment` represents the URL fragment.

IAB launch calls by following the paths computed by the static analyzer. In this section, we will discuss each component in detail.

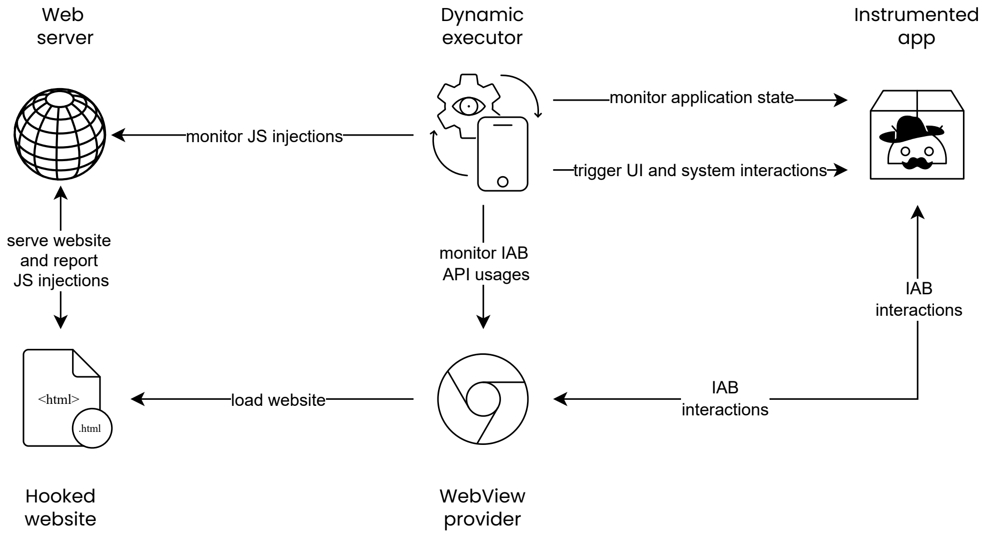


Figure 4.4: Overview of the dynamic analysis phase.

4.5.1 Custom WebView Provider

The `WebView` class in Android, which represents the View in which web content is rendered and provides the APIs to interact with the web, is not directly linked to the underlying browser. Instead, the actual browser functionality is provided by an external application, known as the *WebView Provider*, which is loaded into the app's process upon the initialization of a WebView. The Android framework handles the loading of the WebView provider. If multiple WebView providers are installed on a device, users can select their preferred WebView provider through the developer settings [58]. We use this to our advantage and implement a custom WebView provider that records the API calls made by the application and replaces the requested website with our controlled website used to monitor the injected JavaScript code.

We base our custom WebView provider on Chromium 125.0.6374 and open-source our implementation.

Monitoring and Recording API calls

To monitor and record the API calls made by the application, we insert `Log.i` statements at the beginning of each function in the WebView provider that matches an API call. The log statements contain the function's name, the parameters passed to the function, and a unique identifier. Upon initialization, this identifier is created for each WebView instance and is used to determine if two API calls belong to the same WebView instance. This is necessary since one WebView provider may simultaneously be responsible for multiple

WebViews, e.g., when multiple WebViews are displayed on a single screen. In specific, we monitor every API call to the `WebView` [27], `CookieManager` [28], `WebSettings` [59], and `WebViewDatabase` [60] classes.

The parameters passed to the function may be very large, e.g., when the `loadData` function is called with a large HTML string. This string may be too large to be logged in a single log statement since the maximum length of a logcat log statement is 4KB. Our evaluations showed that splitting the string into multiple log statements and immediately logging many statements one after the other may lead to logcat missing some statements in some cases. We use the following approach to prevent this: If the string is too large to fit into a single log statement, we create a temporary file and write the string to the file. Instead of logging the whole string, we log the API usage and the path to the temporary file. As we will discuss later, the dynamic executor can then pull the file from the device and thus retrieve the whole string.

Loading the Hooked Website

For our hooked website to record the injected JavaScript code, we must ensure it is loaded into the WebView. To this end, we add a new flag to Chromium that can be enabled in the WebView DevTools, a graphical interface on the device that allows developers to set various flags and settings in the WebView provider. Enabling this flag causes the WebView to force-load the hooked website, reflects this in the WebView hooks, and ignores possible SSL exceptions that this may cause.

Overwriting the URL. The WebView provider overwrites the URL of the requested website passed in the `loadUrl` and `postUrl` calls with the URL of the hooked website. Calls to `loadData` and `loadDataWithBaseURL` are internally redirected to `loadUrl` with the hooked website as a parameter. Furthermore, we add the unique identifier of the WebView instance as a URL fragment to the website's URL. This allows the hooked website to track which WebView instance loaded the website, as we will describe in more detail in Section 4.5.2.

Handling hooks. Because applications embedding WebViews can be notified about events on the website, e.g., through the `WebViewClient` and `WebChromeClient` classes, and these notifications contain website-specific information, such as the website's URL, we overwrite these methods to return the URL of the originally requested website, which obfuscates the fact that the hooked website is loaded.

Handling SSL exceptions. Although the hooked website is served via HTTPS and the certificate from the root CA that signed the website's certificate can be installed into Android's system certificate storage, some applications employ certificate pinning, which can block websites not signed by a designated certificate. To circumvent this, we modify the `onReceivedSslError` method in the `WebViewClient` class to always invoke the `proceed` method of the `SslErrorHandler`, effectively allowing the WebView to load the hooked website despite SSL certificate errors.

WebView Provider Restrictions

For security reasons, Android restricts the applications that can be used as WebView providers. When attempting to use an application as a WebView provider, three checks are typically performed [61]:

- **Signature check at installation:** Since Android uses Application Signing [62], all applications installed on the device must be signed. In case an application is updated, the signature of the updated application must match the signature of the installed application.
- **Package name check when setting the WebView provider:** Since Android 10 (API level 29), the package name of the application that is set as the WebView provider is restricted to `com.google.android.webview` (preinstalled) and `com.google.android.webview.[beta|dev|canary]` [26].
- **Signature check when setting the WebView provider:** When setting an application as the WebView provider, the application's signature is checked against a list of predefined signatures located in the `res/xml/config_webview_packages.xml` file of the `framework-res.apk`, which is part of the Android framework.

Therefore, the installation and setting of a WebView provider differs from a regular application installation. Google waives some restrictions for some Android build types, and the correct setup depends on the build type used.

AOSP Builds. In Android Open Source Project (AOSP) builds, builders have full control and can modify the system to accept any WebView provider. Unfortunately, AOSP does not contain the Google Mobile Services (GMS), which applications may rely upon. This renders applications relying on these services, such as those displaying Google Maps, unusable.

User Builds. All three checks are enforced on user builds of Android, i.e., builds that end users use. Interestingly, the `framework-res.apk` does not specify a signature for the `com.google.android.webview` package name. Nevertheless, it is impossible to simply update it, as the install-time signature check is enforced, and an application with this package name comes pre-installed. To circumvent this, we temporarily disable the verification of application signatures on Android using the CorePatch module [63] of the Xposed framework, build the custom WebView Provider with the package name `com.google.android.webview`, and install it on the device. After the installation, we re-enable the signature verification.

Userdebug Builds. Google waives some of the restrictions for `userdebug` builds of Android, i.e., builds used for development purposes, for which Google also provides ready-to-use emulator images running Android with Google Mobile Services (GMS). While this build type still only allows the usage of WebView providers with the package names `com.google.android.webview` and `com.google.android.webview.[beta|dev|canary]`, the signature check at the time of setting the WebView provider is disabled. The WebView provider can, therefore, be built with any of the above-mentioned package names and installed on the device without any further modifications.

4.5.2 Hooked Website

A primary objective of this thesis is to examine the JavaScript code injected by applications into web content. Since manual analysis of the injected JavaScript code is cumbersome, we have established a controlled monitoring environment, i.e., a website that we control and that monitors the actions performed by injected JavaScript code. This website is always loaded by our custom WebView provider whenever a website is requested.

Serving the Website

The website is served by a web server running on the Android device. We implement the custom web server as a separate application and rely on ktor [64], a framework for building web servers in Kotlin. The application consists of an activity that can be used to start and stop the server and a foreground service that keeps the server running in the background, i.e., the web server runs even when the application is closed. Compared to running an external web server, this has the advantage that we do not have network-related overhead and can stick to a simple setup.

Recording Actions of the Injected JavaScript Code

Upon loading the website, we hook JavaScript functions that are interesting to us, such as `document.cookie`. We do this as follows: For functions, we overwrite the original function with a new function that reports the function name along with the parameters passed to it before calling the original function. For properties, we use `Object.defineProperty` to overwrite the property's getter and setter. This interception records any access or modifications to the property, recording how the JavaScript manipulates and accesses the web content. In cases where the function or property is defined on the object's prototype, we modify the prototype directly.

Additionally, we set up a `MutationObserver` within the `window.onload` function to monitor any changes to the DOM[2], enabling observation of how the document's structure changes.

We also implement a Content-Security-Policy (CSP) [66]. Although primarily a security feature designed to mitigate XSS attacks by restricting resources the browser can load,

[2] The Document Object Model (DOM) represents the structure and content of a website [65].

we utilize it to monitor the JavaScript's activities further. Our CSP prevents form submissions, blocks all navigations on the page, and restricts URLs that can be accessed via functions such as `XMLHttpRequest` or `fetch` to only allow the website's origin. By monitoring violations of this CSP, we can detect injected JavaScript code that attempts to submit form data, navigate to other websites, or send data to external servers.

Furthermore, we register a Trusted Types policy [67], another security measure designed to mitigate XSS attacks that works by ensuring that all data passed into sensitive functions, such as `Element.innerText` or `Element.innerHTML`, is properly sanitized. For example, a direct assignment, such as `element.innerHTML = "<p>Paragraph</p>"`, would be denied under this policy. However, the same content passed through a Trusted Types policy `policy` as in `element.innerHTML = policy.createHTML("<p>Paragraph</p>")`, would be allowed. Monitoring and logging violations of this policy allow us to identify such modifications.

In order to test whether the application actively seeks out login forms and attempts to extract values, the website includes a fake login form with username and password fields.

We give a complete list of the monitoring capabilities of the hooked website in Table 4.2.

Reporting Actions

Whenever an action of interest occurs within the website, it initiates a `POST` request to the web server hosting it. This request includes detailed information about the action performed and the identifier of the WebView instance that loaded the website. This identifier, previously passed to the website as a URL fragment, helps in tracking which WebView instance is responsible for the action. Upon receiving a `POST` request, the server logs the event using `Log.i`, informing the dynamic executor about the actions.

4.5.3 Dynamic Executor

The dynamic executor is the key component of the dynamic analysis phase, responsible for driving the execution of the application towards the IAB launch calls. It also monitors the interactions between the application and the web. The executor is implemented as a Python script that employs the Android Debug Bridge (ADB) and the dynamic analysis tool Frida for application interaction.

High-Level Overview

The dynamic executor begins by targeting one IAB launch call at a time. For each IAB launch call, it follows the paths computed by the static analyzer, which outlines the exact sequence of methods the application must execute to reach the call, along with necessary interactions at each step, such as whether a button click is necessary or a specific activity has to be launched. Therefore, it continuously monitors the functions executed by the application and performs the required interactions. If the executor encounters an IAB launch call, it queries and records the app-to-web interactions captured by the custom

Monitored Element	**Description**
DOM Manipulation Monitoring	
Hooking of `Element.innerHTML`, TT	Monitors changes to the inner HTML content of elements.
Hooking of `Element.outerHTML`, TT	Monitors changes to the outer HTML content of elements.
Hooking of `HTMLFormElement.action`	Monitors changes to the action URL of form elements.
Hooking of `HTMLIFrameElement.srcdoc`	Monitors changes to the source document of iframe elements.
Hooking of `HTMLIFrameElement.src`	Monitors changes to the source URL of iframe elements.
Hooking of `HTMLScriptElement.text`	Monitors changes to the text content of script elements.
Hooking of `Element.insertAdjacentHTML`, TT	Monitors insertion of HTML content adjacent to elements.
Hooking of `document.write(ln)`, TT	Monitors calls to write new content into the document.
Hooking of `Range.createContextualFragment`	Monitors creation of new DOM subtrees from strings.
Hooking of `Element.setAttribute`, TT	Monitors changes to attributes of HTML elements.
Hooking of `HTMLInputElement.value`	Monitors changes to the value of input elements.
Hooking of `HTMLAnchorElement.href`	Monitors changes to the hyperlink reference of anchor elements.
Registration of `MutationObserver`	Monitors changes to the DOM.
Data Exfiltration Monitoring	
Hooking of `document.cookie`	Monitors access to and modifications of cookies.
Hooking of `window.postMessage`	Monitors messages sent between window contexts.
Hooking of `XMLHttpRequest`, CSP	Monitors AJAX requests made by the browser.
Hooking of `window.fetch`, CSP	Monitors fetch API requests made by the browser.
CSP	Monitors insertions of `a` tags.
CSP	Monitors the creation of WebSockets.
CSP	Monitors usage of EventSource.
CSP	Monitors calls to Navigator.sendBeacon.
Dynamic JavaScript Execution Monitoring	
Hooking of `document.cookie`	Monitors access to and modifications of cookies.
Hooking of `window.postMessage`	Monitors messages sent between browser contexts.
Hooking of `XMLHttpRequest`	Monitors HTTP requests made by the browser.
Hooking of `window.fetch`	Monitors fetch API requests (e.g. HTTP requests) made by the browser.
TT	Monitors the usage of `eval`.
Event Monitoring	
Hooking of `EventTarget.addEventListener`	Monitors event listeners added to DOM elements or other event targets.
Hooking of `document.addEventListener`	Monitors event listeners added to the entire document.
Hooking of `window.addEventListener`	Monitors event listeners added to the window object.
Hooking of `document.onunload`	Monitors unload events on the document.
Hooking of `window.onunload`	Monitors unload events on the window.
Hooking of `HTMLFormElement.onsubmit`	Monitors form submission events.
Hooking of `HTMLInputElement.onkey[down|up|press]`	Monitors key[down|up|press] events on input elements.
Hooking of `document.onkey[down|up|press]`	Monitors key[down|up|press] events on the document.
Hooking of `window.onkey[down|up|press]`	Monitors key[down|up|press] events on the window.
CSP	Monitors form submission and navigation.

Table 4.2: Monitoring capabilities of the hooked website.

WebView provider and the JavaScript injections reported by the web server serving the hooked website.

Following a Path

The static analyzer often generates a large set of potential paths, some of which may include subpaths of others. Before following a path, we check whether the path is new, i.e., whether we have already followed a path with the same beginning and got stuck before the paths diverge. For instance, the analyzer may identify six paths to an IAB launch call, all requiring the launch of the same activity, but the paths divert after the activity launch. The application immediately crashes because the activity requires a specific intent extra to be set that the analyzer does not know about. In this case, there is no benefit in following all six paths since they all require the same interaction that fails. We, therefore, skip such paths.

Furthermore, we prioritize resolved paths, i.e., those that start at an exported activity. We only follow partially resolved paths, i.e., those starting at a non-exported activity, if following all resolved paths did not lead to the IAB launch call. We do this because we can guarantee that reaching an IAB launch call via a resolved path does not lead to a false positive, whereas this guarantee does not hold for partially resolved paths.

For time reasons and to avoid getting stuck on a path, we set a timeout for each path to 4 minutes. Furthermore, we restrict our exploration to 20 resolved and 20 partially resolved paths for each IAB launch call.

Progress and Application Monitoring

We continuously monitor the execution of the application while we follow a path. In specific, we monitor three aspects: (1) the functions of the application that are executed, (2) IAB launches, and (3) IAB interactions. Since all of these information are logged, we can use the `logcat` [68] command line tool of the Android Debug Bridge (ADB), which is part of the Android SDK, to read these them from the device.

Monitoring Function Calls. In order to determine whether the execution of the application is on track with the path that we are following and update the path progress accordingly, we monitor the functions executed by the application. In the instrumentation phase, we instrumented the application by adding logging statements to the beginning of the functions that are on our path, which the dynamic executor can now read from the device's log.

Monitoring IAB Launches. The instrumented application contains a log statement that is logged when an IAB launch call is triggered. This log statement contains a unique identifier that allows us to determine which IAB launch call has been triggered and that we can read.

Monitoring IAB Interactions. We also query the IAB interactions that have been performed. These come from two sources: the web server that serves the hooked website and the custom WebView provider. While the web server logs the actions of the JavaScript code injected into the hooked website, the custom WebView provider logs the app-to-web interaction API usage. In case the log statement suggests that the log line was too long, meaning that the custom WebView provider created a temporary file on the device to store the log line, we pull this file from the device using ADB and reconstruct the log line.

Interacting with the Application

When we encounter a step along the path that requires an interaction, such as clicking a button, we execute this interaction. In IABInspect, such interactions are modeled as *Actions*. One interaction might lead to multiple possible actions. For example, several buttons may activate the same callback method, but only one may lead to the desired IAB launch call. Every possible button click is therefore modeled as an action, i.e., an action is a specific instance of an interaction we can perform.

Whenever we determine the possible actions of a required interaction, we add these to the *Action Queue*. The action queue can be considered the set of "to-do" actions. Whenever the execution is stuck on a path, the dynamic executor takes the first action from the action queue and performs it, hoping that this action will lead to progress in the path. Conceptually, this is similar to a breadth-first approach, where we explore all possible actions that can be performed at a given application state before moving on to the next state.

One challenge arises when an action fails to advance the path, requiring recovery from this detour. Simple strategies like pressing the back button may work if a new activity is initiated but fail in other scenarios, such as when a button click starts a video. In such cases, pressing back may lead to unexpected behaviors. To address this, we adopt a more comprehensive strategy. Along with determining possible actions for a required interaction, we also record the sequence of events that led to the current application state, referred to as the *Transition Path*. An action is, therefore, always associated with a specific application state defined by the transition path. If an action is about to be performed, we verify that the application is in the correct state. If not, we revert the application to the correct state using the transition path and then perform the action.

A transition path comprises a sequence of *transitions*, each involving a source screen, a target screen, and an action. A *screen* is defined by the currently displayed activity and its clickable elements. We use the `uiautomator2` library to retrieve the XML layout of the current screen and identify its clickable elements. Since we noticed issues with this approach when WebViews are involved, we exclude clickable elements that are part of a WebView. To determine the displayed activity, we use the `adb shell dumpsys activity activities` command.

To revert an application to a specific state, we restart the application and perform each action in the transition path. For UI interactions, we may have to wait until a specific screen is displayed. Therefore, we continuously cross-check the current screen with the screens in the transition path, waiting for the correct screen to be displayed and then performing the action.

We distinguish between two types of possible interactions: *system interactions* and *UI interactions*, which we will discuss in the following sections.

System Interactions

A *system interaction* is an interaction that requires the application to interact with the Android system. Currently, we only support launching activities as system interactions. When a system interaction is required, we first check whether the target component to launch requires any intent extras. Fortunately, the static analysis phase provides this information. From the static analysis, we know the intent extras' names and their data types. The activity to launch may require intent extras to be set to be started, or it may implement conditional logic based on the intent extras. For example, an activity may take a URL as a string extra and only launch a WebView if the URL is not empty or starts with `https://`. Therefore, we need to supply values for the intent extras. First, since we noticed that applications heavily rely on the use of the `url` intent extra, we supply this intent extra in every launch. We do this to increase the chances of triggering an IAB, even when the static analysis did not detect the use of such an intent extra. Second, we use a simplified fuzzing strategy.

Fuzzing of intent extras. We use a simplified fuzzing strategy and add common values for each data type as intent extras. Specifically, we create all possible combinations of intent extras based on the data types and values depicted in Table 4.3. For instance, if a target activity `TargetActivity` takes a boolean extra `open` and a string extra `web`, we add four actions to the action queue:

- `(TargetActivity, {open: true, web: "https://44website66.com"})`
- `(TargetActivity, {open: true, web: "44String66"})`
- `(TargetActivity, {open: false, web: "https://44website66.com"})`
- `(TargetActivity, {open: false, web: "44String66"})`

Performing system interactions. Although intents can be sent and activities launched using ADB with the `am` command, which also supports sending intent extras such as strings, integers, or booleans, we opt not to use this method. ADB is limited to launching only exported activities. Instead, to also launch non-exported activities, we utilize Frida to send the intent. Specifically, we dynamically instrument the application to call the `startActivity` method from the currently opened activity and pass in any required intent extras.

Extra data type	Values
boolean	true, false
String	“https://44website66.com”, “44String66”
int	0, 1
float	1.0
long	1
short	1
char	’a’
byte	1

Table 4.3: Intent Extras

UI Interactions

Android applications use event handlers to react to user inputs. To react to button clicks, application developers have two options: they can implement a predefined interface that reacts to button clicks, e.g., `View.OnClickListener` and then set the button’s click listener to an instance of the implemented interface, or they can define the click handler in the XML layout file. In the latter case, the click handler is defined as an attribute of the button element, e.g., `android:onClick="onButtonClick"`.

Whenever a step in the path requires a UI interaction, the static analyzer provides the callback method that needs to be triggered. A naive approach would be directly calling the respective callback method dynamically, e.g., using Frida. This has some drawbacks, however. These callback methods take the `View` that was clicked as an argument. Since we do not know which `View` needs to be clicked, and the callback method may use the `View` to perform some action, we cannot simply call the callback method directly. Instead, we need to find the specific `View` that needs to be clicked. Statically determining the specific `View` that calls a specific callback is challenging and leads to inaccurate results. Instead, we dynamically determine the specific `View` that needs to be clicked.

To do so, we first use `uiautomator2` to get all of the clickable `Views` on the current screen. We then use Frida to overwrite all click listeners in the application with a custom click listener. We then click all the clickable `Views` on the screen using `uiautomator2`, which gives us a mapping between `Views` and click listeners. After this process, we remove the overwritten methods and create an action for each clickable `View` that triggers the required callback method, which we add to the action queue.

The callback method is not directly called for applications that use the AndroidX library and those that specify the callback method in XML. Instead, the `onClick` method of the `androidx.appcompat.app.AppCompatViewInflater$DeclaredOnClickListener` is called. This listener then dynamically searches for the respective method to call. In this case, we cannot simply overwrite this intermediate callback. Instead, we need to instrument it and dynamically retrieve the target method call using Frida.

IABInspect currently only supports clicks on Views. Other UI interactions, such as swipes or taps, are not supported.

CHAPTER 5

Evaluation and Limitations

In this chapter, we evaluate our prototype tool, IABInspect, focusing on its ability to accurately trigger IABs that are reachable in real-world scenarios while avoiding launching those that are not. A reliable evaluation of IABInspect's overall false negatives requires a ground truth of all reachable IABs within an application, which is not available for our dataset. Establishing such a ground truth without access to the application's source code is challenging and not only laborious but also prone to errors, especially considering that applications may include various forms of obfuscation to hide code or hidden triggers activated only under specific conditions.

Given these challenges and considering IABInspect's primary objective to minimize false positives, we do not attempt a full evaluation of the tool's overall false negatives. Instead, we focus on assessing its precision, which is defined as the ratio of correctly triggered IAB launch calls to the total number of IAB launch calls triggered. Nevertheless, by tracking why the dynamic analysis phase failed to trigger a reachable IAB call for which a path could be found, we can pinpoint and understand the current limitations of the dynamic analysis phase in triggering IABs.

Even though we do not conduct a detailed evaluation of the tool's overall false negatives, we also discuss potential limitations of the static analysis phase that may lead to false negatives in IABInspect later in this section.

5.1 Evaluation

From our dataset of applications, which we run IABInspect on in Chapter 6, we randomly select 10 applications that were successfully processed. We restrict our selection to applications that meet two criteria: each application must contain at least one IAB launch call that was identified by the static analyzer but not triggered during the dynamic analysis phase and at least one IAB launch call that was successfully triggered. This dual selection criterion allows us to comprehensively evaluate both the static and dynamic analysis phases simultaneously.

5.1.1 Static Analysis Phase

For each application in our sample, we manually analyze the IAB launch calls for which the static analyzer identified a path. This is done to determine whether these launch calls are indeed reachable in a real-world scenario, i.e., are true positives, or whether the static analyzer falsely identified them as reachable, i.e., they are false positives. We define a call as reachable if it can be triggered on an actual device, specifically a Pixel 4a running Android 13, which represents the testing environment used throughout this thesis. If we encounter a false positive, we analyze the reason for this.

To verify the reachability of each IAB launch call, we employ a combination of manual reverse engineering and code inspection, supplemented by manual UI fuzzing. The latter technique involves interacting with the application's user interface to trigger the IAB launch call. If we cannot conclusively determine a call's reachability through code inspection, e.g., due to heavy obfuscation of the code, and we cannot trigger it through manual fuzzing, we consider it as not reachable. We explicitly highlight such cases in our evaluation.

Among the 10 selected applications, the static analyzer found paths to 45 IAB launch calls. Out of these, 35 IAB launch calls were correctly identified as reachable. In contrast, 10 were inaccurately identified as reachable, resulting in a precision[1] of 77.8%. The results of our manual analysis of the static analysis phase are presented in Figure 5.1.

Reasons for False Positives

Various factors led the static analyzer to identify paths to IAB launch calls that were not actually reachable in our sample. We identified the following reasons.

Device and version-specific functionality. Two IAB launch calls were behind checks for device-specific functionality or a specific version. In one case, the IAB launch call could only be triggered on devices with a specific version of the WebView installed, while in the other case, the IAB launch call was only reachable on devices with a specific screen size, both of which were not met in our testing environment.

Conditional logic in code. Applications can include conditional checks that block the execution of IAB launch calls unless specific variable values are met. We identified four such IAB launch calls in our sample, for which the conditions to trigger these calls were never satisfied. This, among others, occurred in scenarios where the IAB launch call was located within a library and depended on the usage of a specific library functionality. However, the functionality was never used in the app.

[1] *Precision* is defined as $precision = \frac{TP}{TP+FP}$, where TP denotes the true positives, and FP the false positives.

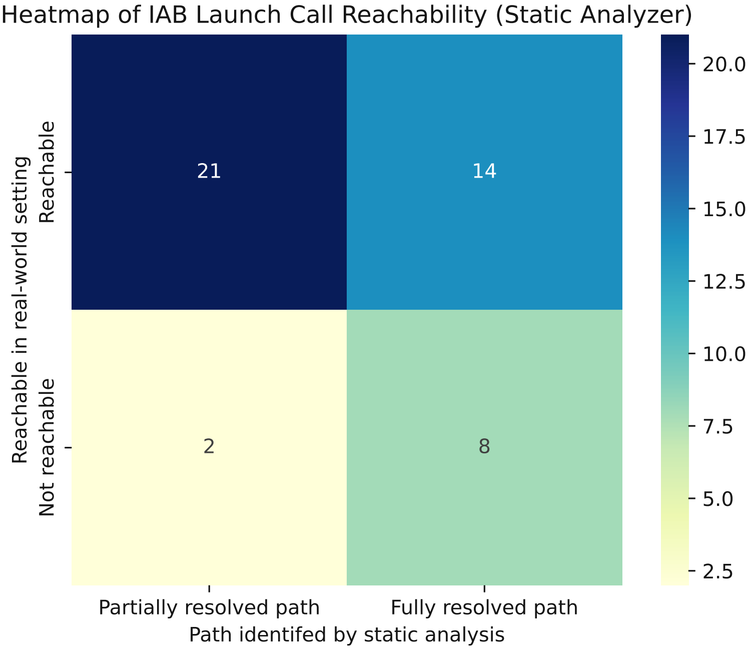


Figure 5.1: Heatmap of IAB launch call reachability in the dynamic analysis phase. The x-axis categories represent the path type identified in the static analysis and the y-axis represents reachability in a real-world scenario. The cell numbers indicate the count of IAB calls in each category.

Class inheritance. In one case, the method containing the IAB launch call was overwritten by a subclass, making the original method in the superclass obsolete. The static analyzer did not accurately model this.

Unknown but not manually triggered. In two cases, we could not determine the exact reason why the IAB launch call was not reachable due to heavy obfuscation, but could not trigger the IAB launch call by manual fuzzing.

5.1.2 Dynamic Analysis Phase

For every IAB launch call triggered by the dynamic analysis phase and where a path was found in the static analysis phase, we check whether we previously flagged the IAB launch call as reachable in a real-world scenario. If we find a discrepancy between our manual analysis and the results of the dynamic analysis phase, we analyze the reasons for this. This evaluation helps us assess the precision and effectiveness, including limitations of the dynamic analysis phase in triggering IAB launch calls.

Out of 20 IAB launch calls triggered by the dynamic executor, only one was not reachable

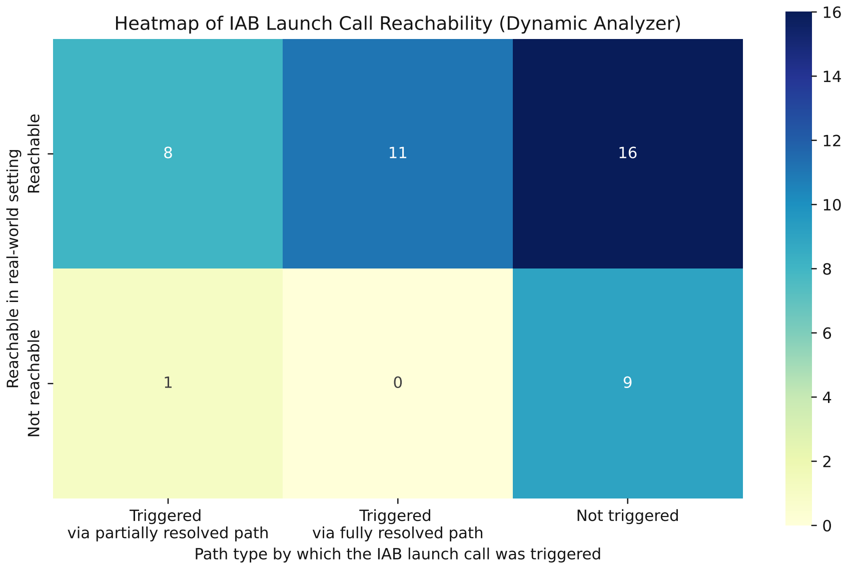


Figure 5.2: Heatmap of IAB launch call reachability in the dynamic analysis phase. The x-axis categories represent the path types identified in the static analysis and the y-axis represents reachability in a real-world scenario. The cell numbers indicate the count of IAB calls in each category.

in a real-world scenario, resulting in a false positive. Conversely, 10 IAB launch calls were correctly identified as not reachable, i.e., the IAB launch calls were indeed not reachable in a real-world scenario.

The dynamic analysis phase achieved a precision of 95% and a recall[2] of 54% (based on the results of the static analysis phase), therefore significantly reducing the number of false positives compared to the static analysis phase. The results of this evaluation are depicted in Figure 5.2.

Reasons for False Positives

False positives in the dynamic analysis phase can only occur when following a partially resolved path, i.e., whenver we launch an activity that is not exported. In real-world scenarios, these activities might not be used at all or only in conjunction with specific values for intent extras. The dynamic executor, however, launches each activity for which a partially resolved path can be found and supplies arbitrary intent extras, which can trigger code paths that would not normally be executed under standard application usage, leading to false positives.

[2] *Recall* is defined as $recall = \frac{TP}{TP+FN}$, where TP denotes the true positives, and FN denotes the false negatives.

In our sample, we encountered one such false positive. The dynamic executor followed a partially resolved path and launched an activity listed in the application's manifest. The purpose of the activity was to be used by developers to test certain functionalities, but it was never called from the application code.

Reasons for False Negatives

The dynamic executor was not able to trigger 4 IAB launch calls that were flagged as reachable and for which the static analyzer could find a path. This discrepancy helped us identify several limitations in the dynamic analysis phase.

System interaction not modeled. The dynamic analyzer is limited to triggering activities only. Reaching some IAB launch calls requires more complex interactions, such as pressing the back button (`onBackPressed`) or any hardware key (`onKeyDown`). Two IAB launch calls could not be reached because the dynamic executor lacked the capability to simulate these interactions.

App crash due to instrumentation. Instrumentation issues with the underlying tool, Soot, can lead to unstable APKs. The application may crash before reaching the IAB launch call in such cases. One IAB launch call was not triggered due to this issue.

UI interaction not modeled. Currently, the dynamic executor only supports triggering UI interactions through button clicks (`onClick`) and clicks on list items `onItemClick`. Other UI callbacks like `onItemSelected`, which may be necessary for triggering specific IAB launches, are not supported. This limitation prevented three IAB launch calls from being triggered.

Incomplete identification of intent extras. While we use ICCBot to identify intent extras required by an activity, it may not always correctly identify all necessary extras. If an application checks for null values in the intent extras and the required extra is missing, the specific code path leading to an IAB launch will not execute. This issue affected two IAB launch calls.

Specific value of intent extras not provided. Applications may also require specific values within intent extras, such as a particular URL or integer value. Additionally, our current setup does not support parcelables within intent extras. These constraints led to three IAB launch calls not being triggered due to incorrect intent extra values.

Web action required. Specific IAB launch calls may also be called only in response to a web action, such as when the web content logs a specific output, the website takes too long to load, or cannot be loaded. Since our hooked website does not simulate these conditions, five IAB launch calls requiring such triggers were not triggered.

5.2 Static Analysis Limitations

We discuss possible limitations of the static analyzer that may lead to false negatives in IABInspect. False negatives occur when IABInspect fails to trigger an IAB launch call present in the application even though it is reachable. The static analysis phase lays the groundwork by generating blueprints that guide the dynamic analysis into executing the IABs. Unfortunately, static analysis, particularly in Android applications, is known for its inherent imprecision.In IABInspect, the following limitations of the static analysis phase may lead to false negatives.

Reflection. Android applications may invoke methods using reflection. Reflection, a feature inherent to Java, enables developers to dynamically inspect, modify, and access objects and invoke methods during runtime. Due to the dynamic nature of method invocation, where the method name may only be known at runtime, static analysis tools encounter challenges in resolving the call target. Soot and FlowDroid, serving as the underlying tools, offer support for reflection when using SPARK as the underlying pointer analysis and configuring `InfoFlowAndroidConfiguration.setEnableReflection(true)`, as IABInspect does. Consequently, our tool can handle straightforward instances of reflection like the one mentioned above. However, more intricate scenarios involving reflection, such as cases where the string containing the method name is encrypted, as is typically used for obfuscation, are unsupported, thus rendering IABInspect incapable of resolving the call target in such situations.

Dynamic code loading. In addition to reflection, Android applications can load code dynamically at runtime through the `DexClassLoader` class. This class enables the loading of classes from locations external to the application's APK file. Since these dynamically loaded classes are not part of the APK, they are not considered by the static part of IABInspect and can thus not be targeted by the dynamic executor. Nevertheless, the dynamic executor can capture the IAB calls from dynamically loaded code if triggered during the run. It is worth noting that applications distributed via the Google Play Store are not allowed to download executable code from sources other than Google Play [69]. Instead, these apps must use Play Feature Delivery, in which split APKs are downloaded from the Google Play Store on-demand [70]. However, our tool currently does not address the analysis of such on-demand loaded APKs.

Native methods. Android applications can also be (partially) developed in C and C++ code. This code can then be called via the Java Native Interface (JNI). Since we rely on Soot and FlowDroid, which are targeted at analyzing Java code, we are unable to analyze such native code.

Android-specific callbacks and events. Android applications are event-driven, i.e., the application's code is executed in response to events, such as user interactions like a button click or system events such as an incoming text message. Statically detecting all possible events that need to be triggered is inherently difficult for Android applications. Even though we rely on FlowDroid, a state-of-the-art static analysis tool for Android that models the Android lifecycle and considers the peculiarities of Android applications, we may still fail to detect some Android-specific callbacks.

Inter-component communication and fragments. As previously discussed, Inter-Component Communication (ICC) plays a vital role in Android applications, and we integrate ICCBot into IABForce to capture ICCs. Despite ICCBot being a state-of-the-art tool for ICC analysis, it is not flawless and may miss connections. Furthermore, since FlowDroid's capabilities of modeling fragments are limited, and ICCBot, in reality, supports fragments only to a limited extent, we may miss IABs that are triggered inside fragments.

CHAPTER 6

Results

In this chapter, we apply IABInspect to a dataset of 1,000 popular Android applications. We begin by discussing the dataset of applications we analyzed. This is followed by a presentation of our evaluation setup. We then discuss the results of our analysis and illustrate the use of app-to-web interactions with case studies.

6.1 Application Dataset

Since no comprehensive listing of applications is available on the Google Play Store, we use AndroZoo [71], a collection of Android applications aimed at helping researchers conduct reproducible experiments. AndroZoo is operated by the University of Luxembourg and contains, as of February 2024, over 24M applications.

To ensure our dataset includes only relevant and widely used applications, we cross-checked the applications in AndroZoo with those available on Google Play, selecting only those also present on Google Play. Further, we narrowed our selection to applications with over 1 million downloads, resulting in a list of 61, 263 applications.

Out of those applications, we were able to download 46,441 applications (75.8%) successfully using `gplaycrawler` [72]. Note that the reasons for failed downloads could be due to restrictions on the availability of the application, such as on the Android version of the device used, the device's architecture, missing hardware features, or geolocation restrictions, e.g., the application is not available in a specific country[1].

Most of these applications (34,863) were distributed as split APKs. We attempted to merge these split APKs into single APK files. This process failed for 19 applications, leaving us with a dataset of 46, 422 applications.

[1] For our experiments, we simulated the use of a Google Pixel 4 running Android 13 (API level 33) based in Austria.

Due to the considerable amount of time required to analyze each application and hardware constraints, we randomly selected 1,000 applications from the dataset on which we ran IABInspect and left the analysis of the other apps for future work.

6.2 Experiment Setup

The experimental setup is divided into two main parts: the static analysis phase, including the instrumentation of the apps, and the dynamic analysis phase, each executed on different machines.

Static Analysis. We perform the static analysis on a server running Ubuntu 22.04 equipped with 112 AMD EPYC 7702 cores and 298 GB of RAM. We optimize the use of this system by concurrently running 18 instances of IABInspect, with each instance allocated 16 GB of RAM. Furthermore, we set a timeout of 120 minutes to analyze an application.

Dynamic Execution. For dynamic execution, we use an Apple Mac Mini that operates an Android emulator, simulating a Pixel 4a running Android 13 (API level 33) with Google Mobile Services. We opted for an emulator rather than a physical device primarily for reproducibility and stability. First, an emulator ensures that other researchers can precisely replicate the testing environment. Second, when using Frida with the instrumented applications on physical devices, we encountered stability issues. These issues were not present when using an emulator.

6.3 Static Analysis

The static analysis phase of IABInspect timed out for 247 applications, leaving us with 753 applications (75.3%). Additionally, the analysis failed for 158 applications due to exceptions within the analysis pipeline. Among these failures, 115 are due to issues with Soot and FlowDroid, while 43 resulted from problems specific to IABInspect. As a result, we are left with 595 applications (59.5% of the original dataset) that successfully passed through the static analysis. A graphical representation of the static analyzer's success rate is shown in Figure 6.1.

IAB Launch Calls

Our static analyzer identified IAB launch calls, i.e., calls to a WebView's `loadUrl`, `loadData`, `loadDataWithBaseURL`, or `postUrl` method, in 552 of the applications analyzed (92.8%). Only 33 applications did not contain any detectable IAB launch calls. The distribution of IAB launch calls per application is illustrated in Figure 6.2. On average, each application contained on average 25 IAB launch calls, though this number varied widely, with a maximum of 190 IAB launch calls found in a single application.

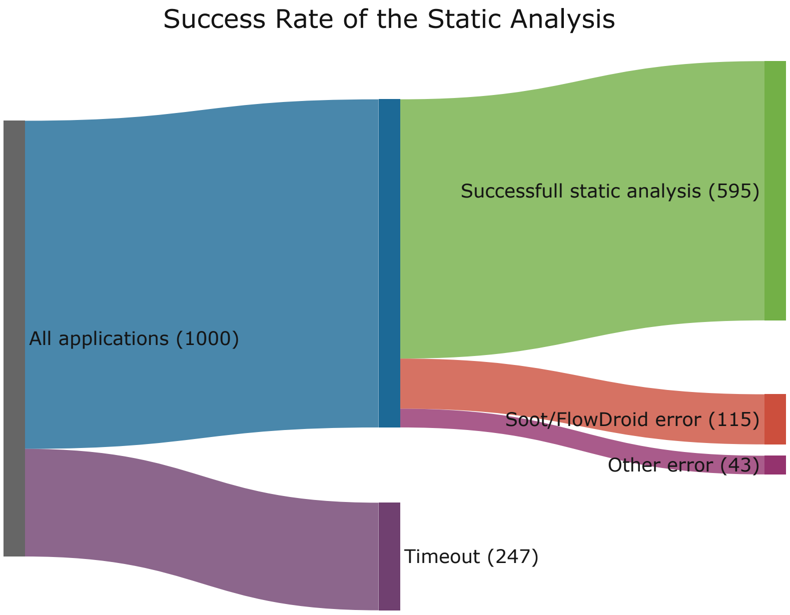


Figure 6.1: Success rate of the static analysis. 595 applications could be successfully statically analyzed.

IAB Launch Method	# call sites		# applications	
	absolute	relative	absolute	relative
`WebView.loadUrl(String)`	9,963	67.10%	546	93.34%
`WebView.loadUrl(String, Map)`	347	2.34%	149	25.47%
`WebView.loadData`	1,160	7.81%	470	80.34%
`WebView.loadDataWithBaseURL`	3,191	21.49%	503	85.98%
`WebView.postUrl`	186	1.25%	108	18.46%

Table 6.1: Distribution of IAB launch methods as determined by the static analyzer. Note that one application can contain multiple IAB launch calls.

The breakdown of the specific methods used for these IAB launch calls is detailed in Table 6.1. The most commonly used method was `WebView.loadUrl(String)`, which appeared in 546 applications, accounting for 67.10% of all calls identified. In contrast, `WebView.postUrl` was the least common, found in 108 applications.

Figure 6.3 shows the sources of the IAB launch calls, i.e., the locations where the calls are made. As can be seen in the figure, the majority of calls originate from third-party libraries embedded in applications. The top 20 libraries are responsible for more than 50% of all IAB calls that the static analyzer could detect. The sources where the IAB

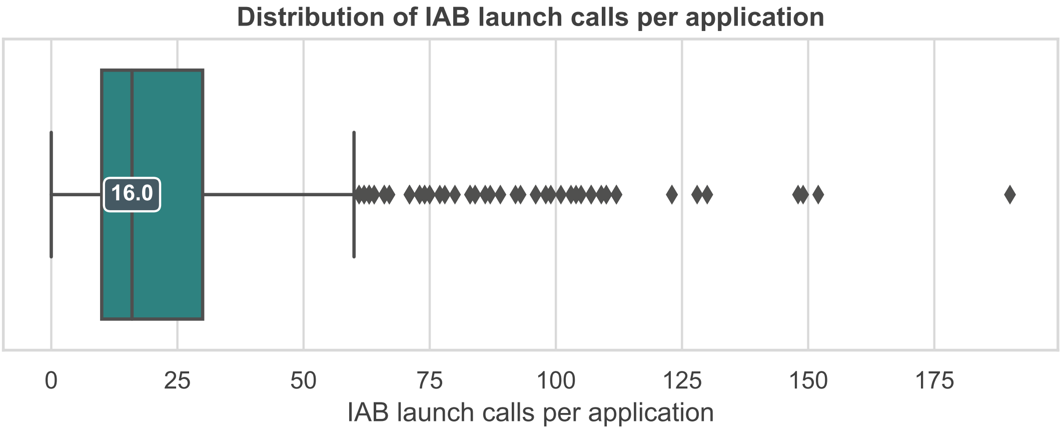


Figure 6.2: Distribution of IAB launch calls per application.

launch calls originate can be categorized into the following groups.

Advertisement and analytics libraries. Advertisement and analytics libraries are responsible for the absolute majority of IAB calls. The most common library is the Google Mobile Ads SDK (`com.google.android.gms.internal.ads` and `com.google.android.gms.ads.internal`). Other libraries include the Meta Audience Network (`com.facebook.ads`) and the AppLovin MAX SDK (`com.applovin`).

IAB libraries. These libraries provide an IAB and serve as a wrapper around the Android WebView. A common library in this category is React Native Webview (`com.reactnativecommunity.webview`), which provides a WebView component for React Native applications.

Other libraries. Other libraries, such as the Unity Services Core SDK (`com.unity3d.services.core`) and SafeDK (`com.safedk.android`) cannot be categorized into the previous two groups. The Unity Services Core SDK library provides core services for Unity games, while SafeDK is a tool that helps developers manage third-party SDKs in their applications.

Others. This category encompasses other sources of IAB launch calls. It includes calls from the application itself but also calls from less commonly used libraries that are not part of those displayed in the figure.

Path Determination

The static analyzer processed a total of 14,847 IAB launch calls, out of which it computed a path for 1,526 calls. Of these, 712 paths were fully resolved, starting at an exported

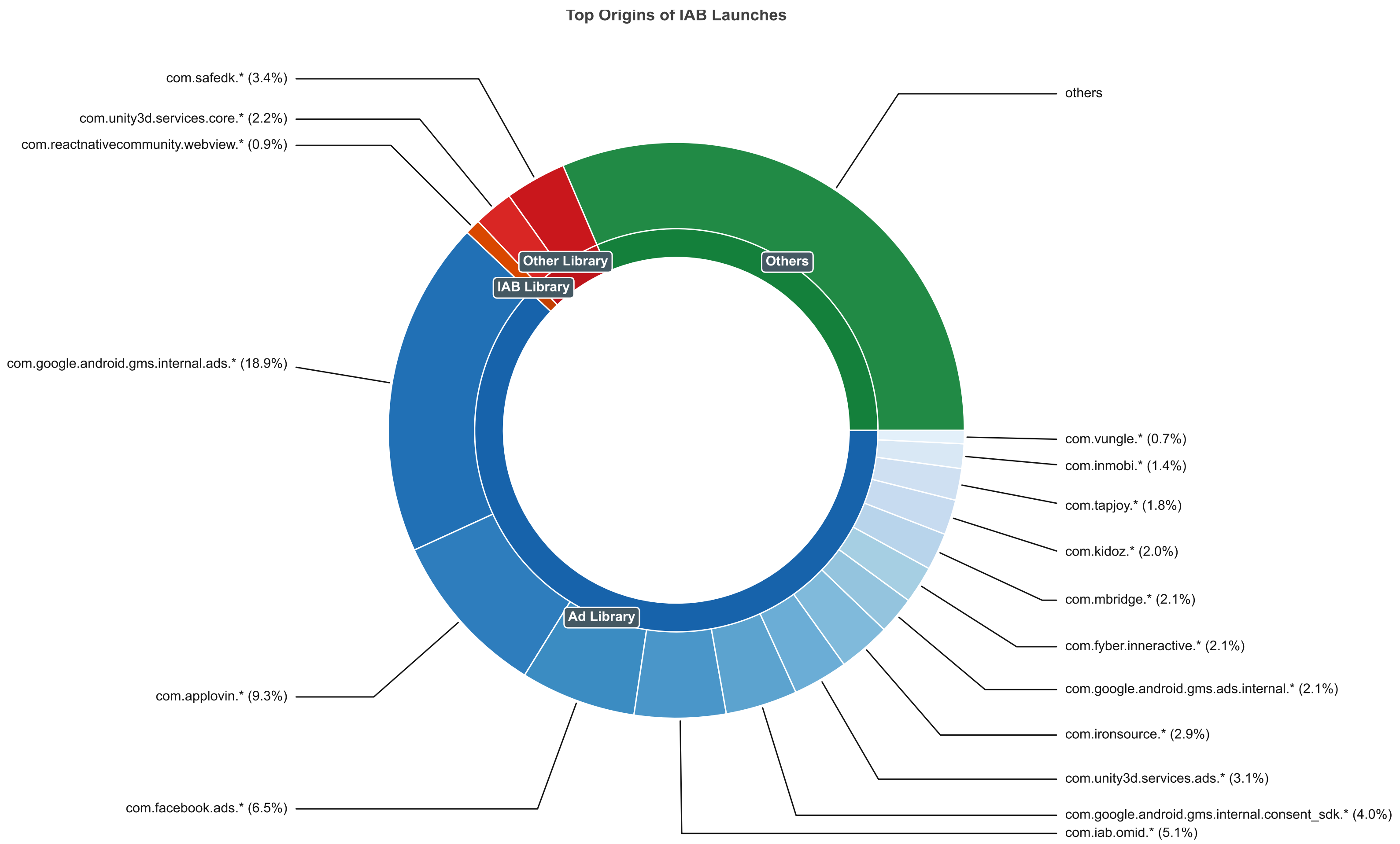


Figure 6.3: Sources of IAB launch calls determined by the static analyzer.

activity, while 814 were partially resolved, beginning at an activity that is not necessarily exported. For 5,648 launch calls, the static analyzer could not determine any path.

Due to timing restrictions and to maintain manageable analysis times, we exclude specific commonly used libraries that contain IAB launch calls from the path-finding process. These libraries, once embedded, function identically across all applications and can later be manually analyzed. Therefore, they are not the focus of our study. Table 6.2 lists the libraries we omitted from our analysis[2].

7,679 call sites were ignored in the path-finding process because they were located in libraries excluded from our analysis. An illustration of the distribution of the path types computed per application is presented in Figure 6.4.

Library	**Package Name**
Google Play Services	`com.google.android.gms`
Google Mobile Ads	`com.google.ads`
Facebook Audience Network	`com.facebook.ads`
Chartboost	`com.chartboost.sdk`
Unity 3D Ads	`com.unity3d.services.ads`
SafeDK	`com.safedk.android`
AppLovin	`com.applovin`
AerServ	`com.aerserv.sdk`
IronSource	`com.ironsource.sdk`
Braze	`com.appboy`
Adobe Experience Platform	`com.adobe.marketing`

Table 6.2: Libraries excluded from path-finding process.

6.4 Dynamic Executor

The dynamic executor attempted to trigger IAB launch calls in all but 90 applications where the static analyzer succeeded. These 90 applications were skipped since either no IAB launch calls could be identified in the app or the static analyzer only identified IAB launch calls that are located in excluded libraries. In total, the dynamic executor followed 2,546 paths computed by the static analyzer. This process led the dynamic executor to trigger 508 IAB launch calls across 196 applications.

Figure 6.5a categorizes these triggered calls by their path types as determined statically: 97 of the triggered IAB launch calls were resolved and 180 were partially resolved. Additionally, the dynamic executor triggered 171 IAB launch calls for which the static

[2] To match a method to a library, we use the method's signature. If the method's signature, such as `com.facebook.ads.AdView.loadAd` matches the package name of a library, such as `com.facebook.ads`, we consider the method to be part of the library.

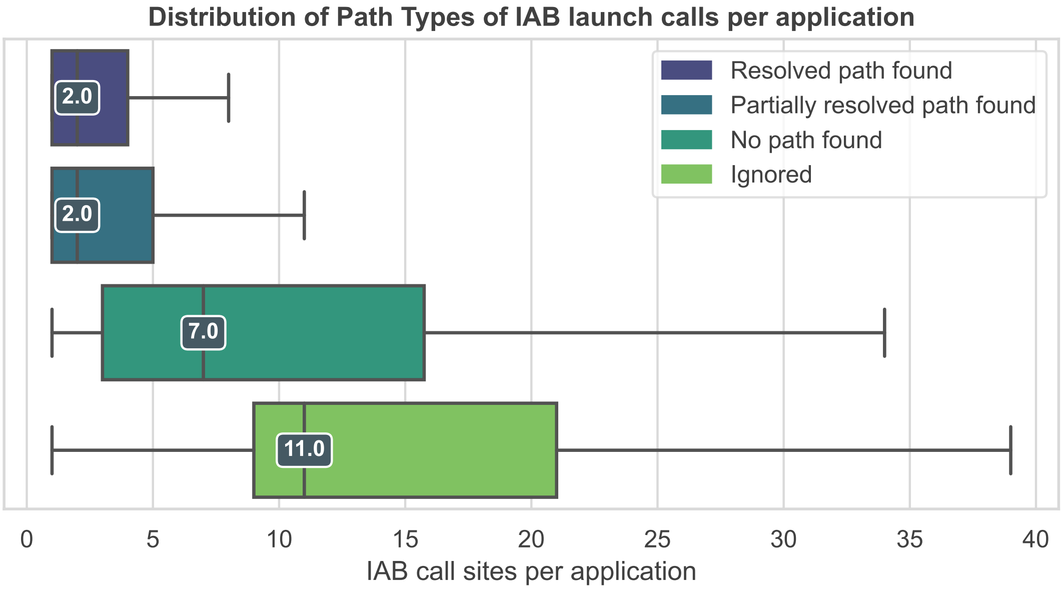


Figure 6.4: Distribution of the path type determined by the static analyzer for IAB launch calls per application.

analyzer had skipped the path-finding process and another 60 for which no path had been found.

The statically-determined path type does not necessarily reflect the circumstances under which the IAB launch calls were triggered. For example, even though a resolved path could be found to an IAB launch call, the dynamic executor may not have been able to trigger it while following this path but while following a partially resolved path to the same call. Moreover, IAB launch calls may be triggered "coincidentally". This is the case when the dynamic executor follows a path intended for an IAB launch call but triggers another IAB launch call on the way.

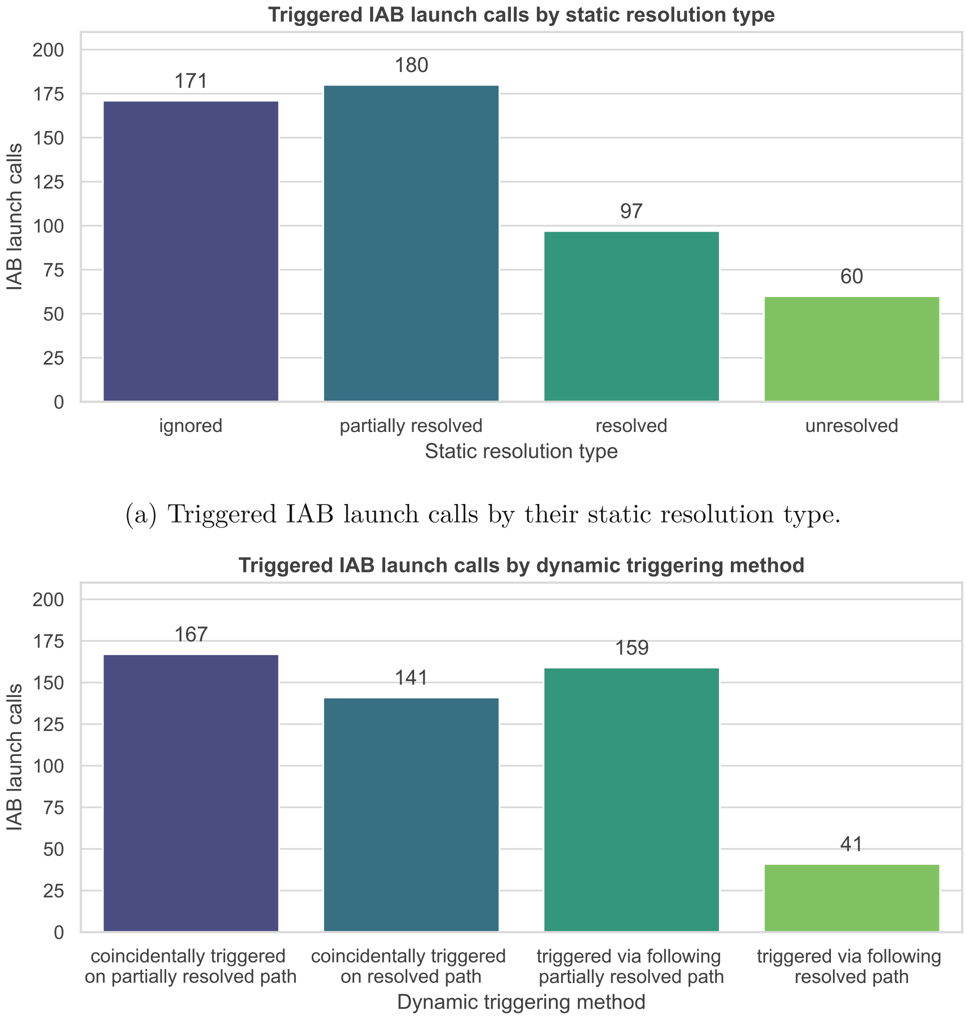
Figure 6.5b illustrates the circumstances under which IAB launch calls were triggered. A majority, 167, were coincidentally triggered while the dynamic executor followed a partially resolved path intended for another IAB call launch. Another 141 calls were triggered under similar conditions but while following a resolved paths. The fewest, 41, were directly triggered by following a resolved path.

6.5 App-to-Web Interactions

This section explores the interactions between applications and the web content. We start by analyzing the content loaded in the IABs, followed by a discussion of the application's API usages. An overview of the API usages in IABs across the analyzed applications is shown in Table 6.3.

WebView API	# Applications
Loading Content	
`WebView.loadUrl(String)`	187
`WebView.loadDataWithBaseUrl`	82
`WebView.loadUrl(String, Map)`	21
`WebView.loadData`	4
`WebView.postUrl`	3
`WebView.reload`	2
`WebView.restoreState`	1
Interacting with Cookies	
`CookieManager.getCookie`	100
`CookieManager.setAcceptThirdPartyCookies`	21
`CookieManager.setAcceptCookie`	19
`CookieManager.flush`	17
`CookieManager.setCookie`	8
`CookieManager.removeAllCookies`	5
`CookieManager.removeSessionCookies`	3
`CookieManager.setAcceptFileSchemeCookies`	2
`CookieManager.removeAllCookie`	1
Influencing Requests	
`WebSettings.setUserAgentString`	126
`WebView.setBlockNetworkImage`	25
`WebView.loadUrl(String, Map)`	21
`WebView.setBlockNetworkLoads`	22
Using JavaScript	
`WebSettings.setJavaScriptEnabled`	213
`WebView.addJavascriptInterface`	155
`WebView.loadUrl(javascript:)`	39
`WebView.evaluateJavascript`	22
Others	
`WebSettings.setSavePassword`	106
`WebSettings.setSaveFormData`	25
`WebView.clearFormData`	2
`WebView.clearCache`	15

Table 6.3: Usage of app-to-web interaction APIs.



(a) Triggered IAB launch calls by their static resolution type.

(b) Triggered IAB launch calls by their dynamic triggering method.

Figure 6.5: Overview of triggered IAB launch calls by static resolution type and their triggering methods during dynamic execution.

6.5.1 Loading Content

We find multiple app-to-web interaction APIs used to load web content. While we have already described the `loadUrl`, `loadData`, `loadDataWithBaseUrl`, and `postUrl` methods, we also consider the `reload` and `restoreState` methods. The `reload` method is used to reload the current URL, while the `restoreState` method is used to restore the state of the WebView from a previously saved state.

To analyze the content that is actually loaded, we consider the parameters of the calls to the `loadUrl`, `loadData`, `loadDataWithBaseUrl`, and `postUrl` methods. The

Content in IAB	# call sites	# applications
googleads.g.doubleclick.net	210	108
file: URL	177	34
about:blank	124	123
Data passed as HTML	82	82
ws.tapjoyads.com	67	6
URL passed in intent extra	50	48
JavaScript code	46	40
null	27	27
empty string	14	13
cdn2.inner-active.mobi	13	13

Table 6.4: Top 10 loaded content in IABs.

majority of the loaded content originates from the domain, specifically googleads.g.doubleclick.net. Other frequently encountered domains include ws.tapjoyads.com and about:blank, the latter indicating the loading of a blank page.

Table 6.4 shows the top 10 content loaded in the IABs across the analyzed applications, categorized by the number of associated IAB launch calls. For calls made via loadData and loadDataWithBaseUrl methods, we extract the content type of the data passed to the WebView. Loading of null or an empty string may indicate that the application requires an intent extra to be set, which our dynamic analyzer could not provide. 48 applications loaded the website that we provided in the intent extra, indicating that the IAB may be used to display arbitrary user-defined web content.

6.5.2 Interacting with Cookies

Applications can interact with websites loaded in them in terms of cookies, i.e., they can read, set, and remove cookies and can define the behavior of the WebView with respect to cookies. To do so, applications can use the CookieManager [28] class.

Reading Cookies. In our dataset, 100 applications read cookies from the WebView. The applications read cookies from 75 unique URLs on 39 unique domains. Cookies from googleads.g.doubleclick.net are the most frequently read, with 81 applications reading cookies from this domain. 6 applications read cookies from codepush.appcenter.ms. Other commonly accessed domains include facebook.com, crashlyticsreports-pa.googleapis.com, and graph.facebook.com.

Setting Cookies. Compared to reading cookies, setting cookies is less common. Only 8 applications set cookies using the CookieManager.setCookie method on 15 unique URLs and 10 unique eTLD+1s. Each URL is used by only one application, which include, among others, gatr.hit.gemius.pl, www.myprotein.com, and ebank.msb.com.vn.

Other Cookie Interactions. Applications can also remove cookies from the WebView using the `CookieManager.removeAllCookie` method and the `CookieManager.removeAllCookies` method, which 6 applications use. Note that the former has been deprecated since Android 5.0. Session cookies, i.e., cookies without an expiration date, are removed by 3 applications.

Applications embedding WebViews can also set specific cookie policies. Using the `CookieManager.setAcceptCookie` method, 19 applications explicitly set whether the WebView should accept and send cookies. The default behavior of a WebView is to accept cookies. Interestingly, none of these applications change the default behavior but rather set it to `true` explicitly.

If cookies, in general, are allowed, applications can also specify whether cookies are allowed when a file scheme URL or third-party cookies are allowed. For the former, the default behavior is to allow such cookies. Even though applications explicitly use APIs to set this policy, they do not overwrite the default behavior, which is to send and accept cookies from file scheme URLs. For third-party cookies, the default behavior varies. For applications that target Android 5.0 or later, the default behavior is to deny third-party cookies. Previous Android versions, by default, accept them. All except one application that uses this API explicitly set the policy to accept third-party cookies to true.

If cookies should be stored across sessions, the `CookieManager.flush` method can be used to save them to storage. 17 applications use this method.

6.5.3 Modifying Requests

Applications can also modify the requests made by the WebView. It is possible to explicitly set the user agent using the `WebSettings.setUserAgentString` method, out of which 105 applications make use of this feature. The website loaded in the WebView can also be loaded with extra headers using the `loadUrl` method. 21 applications use this feature.

The application can pass additional HTTP headers to the server when web content is loaded into an IAB using the `loadUrl(String,Map)` method. Our analysis finds 21 applications using this pattern. Based on our analysis, the injected headers are not standard headers, but custom headers, such as `app-key`, `platform`, and `package-name`.

In 22 applications, we found the usage of `WebView.setBlockNetworkLoads`, which allows applications to set whether the WebView should block network loads. The default behavior is to allow network loads. None of the analyzed applications actually changes the default behavior. One application uses the `WebView.setBlockNetworkImage` method to block the loading of network images.

6.6 JavaScript Injection

Based on our analysis, injecting JavaScript code is a common practice, with 50 applications employing this technique, either by using the `loadUrl` method with a JavaScript string or the `evaluateJavascript` method.

IABInspect detected 137 unique JavaScript code snippets being injected into WebViews. From these snippets, we identified four primary use cases for JavaScript injection:

- modification of the appearance of the website,
- accessing the website's content,
- facilitating bi-directional communication in hybrid applications, and
- altering the behavior of the website.

In the following sections, we will illustrate each use case with case studies of applications from our analysis. Although an application may employ JavaScript injection for various purposes and could thus be categorized under multiple use cases, we assign each application to the use case that best describes the primary intent of the injected JavaScript code.

6.6.1 Modifications of the Appearence of the Website

A common use case among the applications we analyzed is to modify the website's appearance. This includes changing the background color, hiding elements, or changing the font size of a website.

Mapple

Mapple[3] is an application that provides maps and navigation and has more than 1M downloads on Google Play. The application uses a WebView to display its Terms of Use (ToS). Upon loading the ToS in the WebView, the application injects the JavaScript code snippet shown in Listing 6.1 to change the website's appearance. The code snippet changes the text color of the website to black, the background color to white, and the font size to 12pt.

```
document.body.style.color = "#000000";
document.body.style.backgroundColor = "#ffffff";
document.body.style.fontSize = "12pt";
```

Listing 6.1: JavaScript code snippet used by Mapple.

[3] Available at `https://play.google.com/store/apps/details?id=jp.mappleon.android.mapplelink`.

ATV

Similarly, *ATV*[4], an application of a Turkish TV channel with more than 5M downloads on Google Play, includes an exported activity that loads any URL in a WebView. Upon loading the URL, the application injects the JavaScript code shown in Listing 6.2 to set the width of the first image on the website to 100%.

```
(function() {
    document.getElementsByTagName(’img’)[0].setAttribute(’width’,’100%’);
})()
```

Listing 6.2: JavaScript code snippet used by ATV.

6.6.2 Accessing the Website’s Content

Applications also use JavaScript injection to access DOM elements of the website and therefore read the content of the website. This can be used to extract information, such as the value of a specific input field or the content of a specific element.

Video downloader, Story saver

The application *Video Downloader, Story Saver*[5] is an application that enables users to download photos and videos from Instagram. It has more than 10M downloads on Google Play.

To download certain types of videos, users must log into their Instagram account in the application. The application uses a WebView for this purposes and loads Instagram’s login page within it. Upon loading, it registers a JavaScript bridge named `action` and injects JavaScript code to periodically (every 600ms) retrieve the value from the DOM element with the name `username`. This element corresponds to the username input field on the Instagram login page, thus allowing the application to capture the username. The script used to achieve this is shown in Listing 6.3.

When the user succesfully logs in, the application accesses the cookies saved for Instagram and stores the username-cookie pair in the shared preferences of the app. These credentials are then used to authenticate requests to the Instagram API to start the download process.

[4] Available at https://play.google.com/store/apps/details?id=tr.atv.

[5] Available at https://play.google.com/store/apps/details?id=com.story.saver.instagram.video.downloader.repost

```
function getUserName(){
    var user = document.getElementsByName("username");
    if(user.length > 0){
        action.getUserName(user[0].value);
    }
}

setInterval("getUserName()",600);
```

Listing 6.3: Code snippet used by Video downloader, Story saver.

6.6.3 Facilitating Bidirectional Communication

JavaScript code that we identified also enables the bidirectional communication between the application and the website, specifically in hybrid applications. The JavaScript code is used to inform the website about certain actions, or instruct the website to inform the application about events.

Fake call - prank

The application *Fake call - prank*[6] allows users to simulate fake call scenarios. It has more than 50M downloads on Google Play. The UI is implemented as a WebView that loads a local file. Whenever the UI should change, the application injects a JavaScript code snippet that calls a function defined on the website that triggers the change, such as the injection of `showWellDone()` to show a success message.

Driver Pulse by Tenstreet

This application[7] serves as a platform for truck drivers and carriers and has more than 1M installs. The application is also implemented as a hybrid application and uses the `evaluateJavascript` method to retrieve information from the website, such as by calling `evaluateJavascript` with `pulse.current_badge_count`. The application also uses JavaScript injection to send instructions to the website by calling functions defined in the website, such as by injecting `pulse.keep_alive(true)`.

CITRUSS World of Shopping

CITRUSS World of Shopping[8] is a home shopping platform with over 1M installs on Google Play. CITRUSS uses communication from the website to the application to keep track of the user's activity on the website and execute actions based on that. For example, if the user clicks on a specific button on the website, the application will navigate to a specific screen. Interestingly, the application does not use a JavaScript

[6] Available at https://play.google.com/store/apps/details?id=com.fakecallgame.

[7] Available at https://play.google.com/store/apps/details?id=com.mobile.tenstreet.driverpulse.

[8] Available at https://play.google.com/store/apps/details?id=citruss.android.

bridge using `@JavascriptInterface` to communicate between the WebView and the application. Instead, the application injects a script using `loadUrl` that implements a custom JavaScript bridge and facilitates communication between the website and the application.

Specifically, the script registers the object `WebViewJavaScriptBridge` in the global scope. This object acts as the point of interaction for JavaScript running on the website. The website's JavaScript code can interact with the native application using two methods provided by `WebViewJavascriptBridge`. The first method is `WebViewJavascriptBridge.send`, which can be used to send a message to the native side. The second method is `WebViewJavaScriptBridge.callHandler`, which is used to send a message to a specific handler registered on the native side.

To facilitate communication without registering a JavaScript bridge provided by the WebView, the script adds hidden iframes to the website, which we term `iframe1` and `iframe2`. When the website sends a message to the application, the script changes the source of `iframe1` to `yy://__QUEUE_MESSAGE__`. This triggers the `shouldOverrideUrlLoading` method in the application. The application checks if the URL of the request starts with the custom URL scheme `yy://`. If so, the application injects JavaScript code that calls the `WebViewJavascriptBridge._fetchQueue()` method. This method stringifies the message and redirects the hidden iframe `iframe2` using the custom URL scheme `yy://return/_fetchQueue/<stringified_queue>`. This again triggers the `shouldOverrideUrlLoading` method in the application, where the application checks if the URL starts with the custom URL scheme `yy://return`. If so, the application parses the stringified message. In case a response is expected, the application sends the response back to the website by injecting `WebViewJavascriptBridge._handleMessageFromNative(<response>)` into the website.

It is not clear why the application uses this complex mechanism to communicate between the website and the application, as the Android WebView provides a simpler way to communicate between the website and the application using the `@JavascriptInterface` annotation. Nevertheless, using IABInspect, we were able to uncover this complex communication mechanism.

e-Sim Countryball Be President

This application[9] is a MMO (massively multiplayer online) game with more than 1M downloads on Google Play. The game is played in a WebView that loads the game's website. The application injects JavaScript code that monitors the user's actions on the website and informs the application about it. The injected JavaScript code is shown in Listing 6.4. The code registers an event listener for the `ajaxComplete` event. Whenever an AJAX request is completed, the application sends a message to the native side using the `Android` JS Bridge. The handler function on the native side checks the URL and, depending on the URL, takes further actions, such as showing a notification.

[9] Available at `https://play.google.com/store/apps/details?id=com.eworld.mobile`.

```
$(document).ajaxComplete(function (event, request, settings) {
    Android.ajaxDone(’Xena.e-sim.org’, settings.url, request.status);
});
```

Listing 6.4: JavaScript code snippet used by e-Sim Countryball Be President.

Myprotein: Fitness & Shopping

Myprotein: Fitness & Shopping[10] is an application that allows users to shop for fitness products. It has more than 1M downloads on Google Play. The application provides a WebView that loads the URL `https://www.myprotein.com/`, which belongs to the same company as the application. The WebView serves as a means to browse the products available on the website. By injecting JavaScript code, the application hides specific elements of the website, such as the footer and the header, and adjusts the website's padding. During the checkout process, the application injects JavaScript code that removes the payment options of Google Pay and Apple Pay from the website and automatically clicks a button that expands a list with further payment options, as shown in Listing 6.5.

```
if(document.querySelectorAll(’button[data-test="
    PaymentOptionRadioList__expandButton"]’)[0]) {

document.querySelectorAll(’button[data-test="
    PaymentOptionRadioList__expandButton"]’)[0].click()

    document.querySelectorAll(’div[data-payment-option-name="GOOGLEPAY"]’)
    [0].remove();

    document.querySelectorAll(’div[data-payment-option-name="APPLEPAY"]’)[0].
    remove();
}
```

Listing 6.5: JavaScript code snippet used by Myprotein.

Furthermore, the app injects JavaScript code that monitors what a user does on the website. This is made possible by previously registering a JavaScript bridge `ReactNativeWebView.postMessage` that allows the website to send messages to the application. The website keeps a list of events happening in an array. The JavaScript code overwrites the push method of the array to intercept the events and send them to the application via the bridge.

Interestingly, the application sets cookies on the website to set specific preferences, such as the user's country and currency. Furthermore, using cookies, it disables the newsletter prompt on the website that would otherwise be displayed to the user.

[10] Available at `https://play.google.com/store/apps/details?id=com.thehutgroup.ecommerce.myprotein`.

```
try {
    const dataLayer = window.dataLayer;
    if(dataLayer) {
        Object.defineProperty(dataLayer, "push", {
            configurable: true,
            writable: true,
            value: function (...args) {
                const result = Array.prototype.push.apply(this, args);
                const event = args[0];
                if(event.event === "elysiumEvent" && event.eventData) {
                    window.ReactNativeWebView.postMessage(JSON.stringify(args
    [0].eventData));
                }
                return result;
            }
        })
    }
} catch(e) {
    true;
}
true;
```

Listing 6.6: JavaScript code snippet used by Myprotein.

6.6.4 Altering the Behavior of the Website

Injected JavaScript code can also be used to alter how the website behaves, as in the case of Telemicro.

Telemicro

Telemicro[11] is an application of the Dominican TV channel Telemicro. The app provides an internal browser activity, in which the application injects the JavaScript code shown in Listing 6.7. The JavaScript code changes the behavior of video playbacks of videos embedded on the website. Specifically, the JavaScript code loops through all video elements on the website and registers an event listener for the play event. When the play event is triggered, i.e., when the user starts playing the video, the event listener prevents the video from playing, resets the video's current time to 0, and informs the application about the video URL that the user wants to view using the JavaScript bridge `VideoHTML5`. Upon receiving the video URL, the application opens the video in a separate activity and navigates away from the WebView.

[11] Available at `https://play.google.com/store/apps/details?id=com.goodbarber.telemicro`

```
(function(){
    var b = document.getElementsByTagName("video");
    videosCount = b.length;
    for(var a = 0; a < videosCount; a++){
        (function(c) {
            b[c].addEventListener("play", function(f) {
                f.preventDefault();
                var d = b[c];
                d.pause();
                d.currentTime = 0;
                VideoHTML5.viewVideo(b[c].currentSrc)
        }, false)
        })(a)
    }
})();
```

Listing 6.7: JavaScript code snippet used by Telemicro.

CHAPTER 7

Conclusion

This thesis introduced a novel approach to analyzing app-to-web interactions in Android in-app browsers (IABs) through a prototype tool, IABInspect, which integrates both static and dynamic analysis techniques. The static analysis creates a blueprint of the application, identifying paths from an entry method to IAB launch calls, while the dynamic executor actively follows these paths to monitor interactions using a custom WebView provider. The WebView provider forces the loading of a website in our control that is able to record JavaScript code injected in it.

IABInspect was evaluated on a subset of 10 applications, achieving a precision rate of 95%. When extended to 1,000 popular Android applications, it successfully triggered 508 IAB launch calls across 196 applications, revealing that 100 of these read cookies from 75 unique URLs. Furthermore, 50 of these applications practice JavaScript code injection. This injection primarily serves functions related to modifying website appearances, accessing content, facilitating web-to-app communication, and altering website behaviors. We furthermore demonstrated the usage of JavaScript injection in various case studies.

Fortunately, we did not find any clearly malicious behavior in the analyzed applications. However, the high prevalence of JavaScript code injection that we observed is concerning. This highlights the need for further research, especially an upscaling of the analysis to a larger dataset to better understand the implications of such practices. This expansion is among the essential steps we plan to address in our future work.

Future Work

The initial findings of IABInspect, while promising, have revealed several possibilities for improvement and expansion. We plan on addressing three main areas in future work: scaling up the analysis, reducing the false negative rate of the tool, determining the relatedness of the app and the web content, and including the analysis of excluded libraries using WebViews.

Scaling up the analysis. In this thesis, we only analyzed a small subset (1,000 applications) of the bigger dataset we collected (46,441 applications). To generalize our findings more confidently and explore a wider array of app behaviors, we plan to apply IABInspect to the entire dataset. This will validate the prevalence of JavaScript injections in a broader context and help potentially identify rarer privacy-invasive IAB usages that were not observable in our small sample.

False negative rate. A limitation in our current methodology is the potential underestimation of security risks due to false negatives. Even though we have not provided a quantitive evaluation of the false negative rate of IABInspect due to the lack of a ground truth, experimental results suggest that the false negative rate is substantial. This issue stems partly from the static analysis's inability to capture all paths due to the complex structures within apps, such as fragments and dialogs. Moreover, the dynamic executor's limited interaction capabilities restrict the paths that can be followed. To address these challenges, we plan to refine our static analysis algorithms to include these complex constructs and enhance the dynamic executor to support more diverse UI and system interactions. To improve the precision of tracking intent extras, we also plan to dynamically hook the `getExtra` method of the `Intent` class to record the extras that are accessed on the fly.

Relatedness of the app and the web content unclear. Currently, IABInspect only records the interactions between the app and the web content, as well as the web content itself. Security-wise, it is important to determine whether the content loaded in the IAB is related to the app. While interactions from *AppA* published by *A* can safely interact with `a.com` published by *A*, since they are under the same control, interactions from *AppA* to `b.com` published by *B* may be problematic, such as when cookies are read. Therefore, we plan to extend IABInspect with capabilities to determine the relatedness of the app and the web content.

Analysis of libraries using WebViews. In this thesis, we excluded the path-finding process for some commonly used libraries for time reasons. However, these libraries may contain potentially privacy-invasive app-to-web interactions. We plan to further analyze these libraries to understand their behavior and the potential risks they pose.

List of Figures

List of Tables

Bibliography

[1] Statista, "Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 4th quarter 2023." `https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/`, Jan. 2024. (Accessed on 07/02/2024, `https://archive.is/HsZJW`).

[2] Statista, "Global market share held by mobile operating systems from 2009 to 2023, by quarter." `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`, Jan. 2024. (Accessed on 07/02/2024, `https://archive.is/4NdUZ`).

[3] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A large-scale study of mobile web app security," in *MoST, co-located with IEEE SP*, 2015.

[4] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *ACSAC*, 2011.

[5] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View to a Kill: WebView Exploitation," in *LEET, co-located with USENIX*, 2013.

[6] E. Chin and D. Wagner, "Bifocals: Analyzing WebView Vulnerabilities in Android Applications," in *WISA*, Springer International Publishing, 2013.

[7] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews," in *RAID*, Springer, 2018.

[8] L. Zhang, Z. Zhang, A. Liu, Y. Cao, X. Zhang, Y. Chen, Y. Zhang, G. Yang, and M. Yang, "Identity Confusion in WebView-based Mobile App-in-app Ecosystems," in *USENIX*, USENIX Association, 2022.

[9] G. Yang, J. Huang, and G. Gu, "Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities," in *USENIX*, 2019.

[10] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and Scalably Vetting JavaScript Bridge in Android Hybrid Apps," in *RAID*, pp. 143–166, Springer, 2017.

[11] M. A. El-Zawawy, E. Losiouk, and M. Conti, "Vulnerabilities in Android webview objects: Still not the end!," *Comput. Secur.*, vol. 109, oct 2021.

[12] L. Yang, X. Cui, C. Wang, S. Guo, and X. Xu, “Risk Analysis of Exposed Methods to JavaScript in Hybrid Apps,” in *Trustcom/BigDataSE/ISPA*, IEEE, 2016.

[13] F. Krause, “iOS Privacy: Instagram and Facebook can track anything you do on any website in their in-app browser.” `https://krausefx.com/blog/ios-privacy-instagram-and-facebook-can-track-anything-you-do-on-any-website-in-their-in-app-browser`, aug 2022. (Accessed on 11/21/2023, `https://archive.is/0u4vE`).

[14] X. Zhang, Y. Zhang, Q. Mo, H. Xia, Z. Yang, M. Yang, X. Wang, L. Lu, and H. Duan, “An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications,” in *USENIX*, 2018.

[15] Android Developers, “Android’s Kotlin-first approach.” `https://developer.android.com/kotlin/first`, Jan. 2024. (Accessed on 15/02/2024, `https://archive.is/EtWqn`).

[16] J. Bleier and M. Lindorfer, “Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART,” in *EuroSec*, ACM, 2023.

[17] 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,” *SIGPLAN Not.*, vol. 49, p. 259–269, jun 2014.

[18] Android Developers, “Application fundamentals.” `https://developer.android.com/guide/components/fundamentals`, Oct. 2023. (Accessed on 11/02/2024, `https://archive.is/iXl73`).

[19] Android Developers, “Introduction to activities.” `https://developer.android.com/guide/components/activities/intro-activities`, Jan. 2024. (Accessed on 11/02/2024, `https://archive.ph/pJtVD`).

[20] Android Developers, “The activity lifecycle.” `https://developer.android.com/guide/components/activities/activity-lifecycle`, Feb. 2024. (Accessed on 08/04/2024, `https://archive.is/PekBb`).

[21] Android Developers, “Services overview.” `https://developer.android.com/develop/background-work/services`, Jan. 2024. (Accessed on 11/02/2024, `https://archive.is/QEFtQ`).

[22] Android Developers, “App manifest overview.” `https://developer.android.com/guide/topics/manifest/manifest-intro`, Apr. 2024. (Accessed on 11/02/2024, `https://archive.is/PuYkA`).

[23] Android Developers, “Intents and intent filters.” `https://developer.android.com/guide/components/intents-filters`, Feb. 2024. (Accessed on 11/02/2024, `https://archive.is/ed7wS`).

[24] MDN Web Docs, "Push API." https://developer.mozilla.org/en-US/docs/Web/API/Push_API, oct 2023. (Accessed on 09/04/2024, https://archive.is/BzXRK).

[25] T. Steiner, "What is in a Web View: An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser," in *WWW*, International World Wide Web Conferences Steering Committee, 2018.

[26] WebView docs, "WebView Build Instructions." https://chromium.googlesource.com/chromium/src/+/lkgr/android_webview/docs/build-instructions.md, Feb. 2024. (Accessed on 21/02/2024, https://archive.is/eQmsl).

[27] Android Developers, "WebView." https://developer.android.com/reference/android/webkit/WebView, apr 2024. (Accessed on 09/04/2024, https://archive.is/Yway9).

[28] Android Developers, "CookieManager." https://developer.android.com/reference/android/webkit/CookieManager, apr 2024. (Accessed on 09/04/2024, https://archive.is/ZI4Qz).

[29] Android Developers, "WebViewClient." https://developer.android.com/reference/android/webkit/WebViewClient, Feb. 2024. (Accessed on 21/02/2024, https://archive.is/2OOAl).

[30] Android Developers, "WebChromeClient." https://developer.android.com/reference/android/webkit/WebChromeClient, Apr. 2024. (Accessed on 26/04/2024, https://archive.is/wTkpO).

[31] Chrome Developers, "Overview of Android Custom Tabs." https://developer.chrome.com/docs/android/custom-tabs, feb 2024. (Accessed on 09/04/2024, https://archive.is/gXAeb9).

[32] P. Beer, M. Squarcina, L. Veronese, and M. Lindorfer, "Tabbed Out: Subverting the Android Custom Tab Security Model," in *S&P*, IEEE, 2024.

[33] GeckoView, "GeckoView." https://mozilla.github.io/geckoview/. (Accessed on 21/02/2024, https://archive.is/IEIb9).

[34] GitHub, "scrosswalk-project/crosswalk." https://github.com/crosswalk-project/crosswalk. (Accessed on 21/02/2024).

[35] Apple Developer Documentation, "SFSafariViewController." https://developer.apple.com/documentation/safariservices/sfsafariviewcontroller. (Accessed on 11/02/2024, https://archive.is/WVkDI).

[36] Apple Developer Documentation, "WKWebView." `https://developer.apple.com/documentation/webkit/wkwebview`. (Accessed on 11/02/2024, `https://archive.is/KM4rb`).

[37] Apple Developer Documentation, "ASWebAuthenticationSession." `https://developer.apple.com/documentation/authenticationservices/aswebauthenticationsession`. (Accessed on 11/02/2024, `https://archive.is/5F379`).

[38] A. Møller and M. I. Schwartzbach, "Static Program Analysis," October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[39] 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," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[40] Wireshark, "Wireshark." `https://www.wireshark.org/`. (Accessed on 27/04/2024, `https://archive.is/g9a9Q`).

[41] Frida, "Frida • A world-class dynamic instrumentation toolkit." `https://frida.re/docs/android/`. (Accessed on 27/04/2024, `https://archive.is/ejVb7`).

[42] A. Tiwari, J. Prakash, A. Rahimov, and C. Hammer, "Understanding the Impact of Fingerprinting in Android Hybrid Apps," in *MOBILESoft*, IEEE, 2023.

[43] Z. Zhang, "On the usability (in)security of in-app browsing interfaces in mobile apps," in *RAID*, ACM, 2021.

[44] Z. Tang, J. Zhai, M. Pan, Y. Aafer, S. Ma, X. Zhang, and J. Zhao, "Dual-Force: Understanding WebView Malware via Cross-Language Forced Execution," in *ASE*, ACM, 2018.

[45] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-AntiRM: Taming Control Flow Anti-analysis to Support Automated Dynamic Analysis of Android Malware," in *ACSAC*, ACM, 2017.

[46] L. Bello and M. Pistoia, "Ares: Triggering Payload of Evasive Android Malware," in *MOBILESoft*, IEEE/ACM, 2018.

[47] M. Y. Wong and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *NDSS*, 2016.

[48] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in Android with Tiro," in *USENIX*, 2018.

[49] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang, *et al.*, "Understanding Malicious Cross-library Data Harvesting on Android," in *USENIX*, 2021.

[50] Z. Zhang, W. Diao, C. Hu, S. Guo, C. Zuo, and L. Li, "An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps," in *WISEC*, ACM, 2020.

[51] Android Developers, "About Android App Bundles." `https://developer.android.com/guide/app-bundle`, Nov. 2023. (Accessed on 14/02/2024, `https://archive.is/2JN9x`).

[52] Android Developers, "Build multiple APKs." `https://developer.android.com/build/configure-apk-splits`, Apr. 2023. (Accessed on 14/02/2024, `https://archive.is/UHe8l`).

[53] GitHub, "REAndroid/APKEditor." `https://github.com/REAndroid/APKEditor`, Feb. 2024. (Accessed on 14/02/2024).

[54] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?," in *Compiler Construction: 9th International Conference, CC 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software*, pp. 18–34, Springer, 2000.

[55] D. Schmidt, C. Tagliaro, K. Borgolte, and M. Lindorfer, "IoTFlow: Inferring IoT Device Behavior at Scale through Static Mobile Companion App Analysis," in *CCS*, ACM, 2023.

[56] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "ICCBot: Fragment-Aware and Context-Sensitive ICC Resolution for Android Applications," in *ICSE*, ACM/IEEE, 2022.

[57] MDN Web Docs, "Identifying resources on the Web." `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web`, 2023 July. (Accessed on 28/04/2024, `https://archive.is/cIVhq`).

[58] Android Developers, "Configure on-device developer options." `https://developer.android.com/studio/debug/dev-options`, Apr. 2024. (Accessed on 17/04/2024, `https://archive.is/LN3JE`).

[59] Android Developers, "WebSettings." `https://developer.android.com/reference/android/webkit/WebSettings`, Apr. 2024. (Accessed on 05/05/2024, `https://archive.is/vcr28`).

[60] Android Developers, “WebViewDatabase.” https://developer.android.com/reference/android/webkit/WebViewDatabase, Apr. 2024. (Accessed on 29/04/2024, https://archive.is/WG7ku).

[61] WebView docs, “WebView Providers.” https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/webview-providers.md. (Accessed on 29/04/2024, https://archive.is/9mXbG).

[62] Android Developers, “Application Signing.” https://source.android.com/docs/security/features/apksigning, Mar. 2024. (Accessed on 29/04/2024, https://archive.is/g6M8w).

[63] GitHub, “LSPosed/CorePatch.” https://github.com/LSPosed/CorePatch, Apr. 2024. (Accessed on 18/04/2024).

[64] GitHub, “ktorio/ktor.” https://github.com/ktorio/ktor. (Accessed on 29/04/2024).

[65] MDN Web Docs, “Introduction to the DOM.” https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction, Nov. 2023. (Accessed on 29/04/2024, https://archive.is/sJXY1).

[66] MDN Web Docs, “Content Security Policy (CSP).” https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, Mar. 2024. (Accessed on 29/04/2024, https://archive.is/cgOsZ).

[67] MDN Web Docs, “Trusted Types API.” https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API, jan 2024. (Accessed on 29/04/2024, https://archive.is/GHEY4).

[68] Android Developers, “Logcat command-line tool.” https://developer.android.com/tools/logcat, jan 2024. (Accessed on 05/05/2024, https://archive.is/2WYw5).

[69] Google Play Console Help, “Device and Network Abuse.” https://support.google.com/googleplay/android-developer/answer/9888379, 2024. (Accessed on 19/02/2024, https://archive.is/QesVl).

[70] Android Developers, “Overview of Play Feature Delivery.” https://developer.android.com/guide/playcore/feature-delivery, Feb. 2024. (Accessed on 19/02/2024, https://archive.is/5Bquu).

[71] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “AndroZoo: Collecting Millions of Android Apps for the Research Community,” in *MSR*, ACM, 2016.

[72] GitLab, “marzzzello/gplaycrawler.” https://gitlab.com/marzzzello/gplaycrawler, july 2021. (Accessed on 05/05/2024).