# Cross-Boundary Mobile Tracking: Exploring Java-to-JavaScript Information Diffusion in WebViews

Sohom Datta[†], Michalis Diamantaris[‡], Ahsan Zafar[†], Junhua Su[†],
Anupam Das[†], Jason Polakis[*], and Alexandros Kapravelos[†]

[†]North Carolina State University, USA
{sdatta4, azafar2, jus6, anupam.das, akaprav}@ncsu.edu
[‡]Technical University of Crete, Greece
mdiamantaris@tuc.gr
[*]University of Illinois Chicago, USA
polakis@uic.edu

***Abstract*—WebViews are a prevalent method of embedding web-based content in Android apps. While they offer functionality similar to that of browsers and execute in an isolated context, apps can directly interfere with WebViews by dynamically injecting JavaScript code at runtime. While prior work has extensively analyzed apps' Java code, existing frameworks have limited visibility of the JavaScript code being executed inside WebViews. Consequently, there is limited understanding of the behaviors and characteristics of the scripts executed within WebViews, and whether privacy violations occur.**

**To address this gap, we propose WebViewTracer, a framework designed to dynamically analyze the execution of JavaScript code within WebViews at runtime. Our system combines within-WebView JavaScript execution traces with Java method-call information, to also capture the information exchange occurring between Java SDKs and web scripts. We leverage WebViewTracer to perform the first large-scale, dynamic analysis of privacy-violating behaviors inside WebViews, on a dataset of 10K Android apps. We detect 4,597 apps that load WebViews, and find that over 69% of them inject sensitive and tracking-related information that is typically inaccessible to JavaScript code into WebViews. This includes identifiers like the Advertising ID and Android build ID. Crucially, 90% of those apps use web-based APIs to exfiltrate this information to third-party servers. We also uncover concrete evidence of common web fingerprinting techniques being used by JavaScript code inside of WebViews, which can supplement their tracking information. We observe that the dynamic properties of WebViews are being actively leveraged for sensitive information diffusion across multiple actors in the mobile tracking ecosystem, demonstrating the privacy risks posed by Android WebViews. By shedding light on these ongoing privacy violations, our study seeks to prompt additional scrutiny from platform stakeholders on the use of embedded web technologies, and highlights the need for additional safeguards.**

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2026.230910
www.ndss-symposium.org

## I. Introduction

The ubiquity of smartphones has made mobile apps central to modern digital life. These apps often handle extensive amounts of sensitive personal information, and mediate a significant portion of user's online activities, WebViews are an integral technology for embedding web content within mobile apps, thereby enabling a rich user experience. Functioning similarly to browsers, WebViews render HTML, CSS, and JavaScript directly inside apps, allowing seamless integration of web-based content, including advertisements, social media feeds, and interactive elements. A distinctive capability of WebViews, however, is the facilitation of interaction between Java (the app-side context) and JavaScript (the web-side context). Java methods can be exposed to JavaScript using `addJavaScriptInterface`, allowing JavaScript to invoke native functionality, while Java code can trigger JavaScript execution using methods such as `evaluateJavaScript`. A previous study reported that approximately 83% of apps on the Google Play Store incorporate WebViews, including popular apps such as Facebook, Instagram, and Twitter [1].

While Java and JavaScript execution contexts within Android apps are technically distinct—each governed by its respective security model—their bridging via WebViews creates potential privacy risks. Java code can inject device-specific identifiers into WebViews, enabling JavaScript to capture and exfiltrate them. Traditional Android analysis tools predominantly focus on native Java code, often neglecting WebViews, whereas existing WebView-focused analyses typically employ static techniques or target specific frameworks. For example, Bai et al. examined data leakage through JavaScript interfaces primarily in Cordova apps [2], while Lee et al. [3] and Tiwari et al. [4] used methods insufficient to fully capture dynamic JavaScript inclusion and execution. Thus, a comprehensive dynamic analysis that bridges these contexts is currently lacking.

To address this gap, we develop **WebViewTracer**, an automated dynamic analysis system designed to analyze cross-context interactions between Java and JavaScript within Android WebViews. Given an Android APK file, WebViewTracer dynamically executes the application while simultaneously recording two key data sources: (1) JavaScript execution logs captured via VisibleV8 [5] detailing Web APIs invoked by JavaScript within WebViews, and (2) Java-WebView interaction logs collected via Frida instrumentation, which tracks method calls between Java and JavaScript contexts. By synthesizing these logs, WebViewTracer provides detailed visibility into the types and directions of information diffusion occurring between Java and JavaScript, including device-specific information leakage and fingerprinting behaviors.

We demonstrate the capabilities WebViewTracer through the first large-scale, dynamic analysis of JavaScript behavior within WebViews, on a dataset of 10K Android apps from the Google Play Store. Our analysis uncovers novel privacy-invasive behaviors, wherein the cross-context bridging enabled by JavaScript injection into WebViews is leveraged for exchanging tracking identifiers and other sensitive user data that is not directly accessible in a given execution context due to existing access control mechanisms. Essentially, the Java-to-JavaScript bridge violates both isolation and policy enforcement policies. Surprisingly, we find that such behaviors are pervasive, as we detect injections of sensitive data in 65% of the 4,597 apps that load WebViews. Crucially, we find that 59% of the apps with WebViews also exfiltrate injected sensitive data, which is typically inaccessible from within WebViews, over the network. Subsequently, we analyze the actors responsible for injecting the private information into WebViews and receiving it over the network. Using dynamic analysis, we map these flows to specific SDK vendors, and find that this form of information leakage is common among major ad libraries and analytics SDKs. Finally, we also examine whether the same actors engage in broader privacy-invasive activities. Specifically, we investigate evidence of device or user fingerprinting based on WebView content and JavaScript execution traces and find that a subset of the actors responsible for context-restricted data leaks, also carry out other forms of aggressive web tracking (e.g., canvas fingerprinting).

In summary, the main contributions of our paper are:

- We introduce WebViewTracer, an open-source, scalable system that dynamically analyzes JavaScript execution within Android WebViews.
- We present the first comprehensive, large-scale, dynamic investigation of cross-context Java-to-JavaScript interactions in Android apps.
- We provide novel insights into the implications of bridging the Java and JavaScript execution contexts, highlighting pervasive privacy-invasive behaviors in the wild.

## II. Background

**Stateful and Stateless Tracking.** Traditionally, tracking in Android used to depend heavily on persistent device identifiers like the IMEI and IMSI. These identifiers allowed long-term tracking and were immutable, i.e., a user could not alter these identifiers if they did not wish to be tracked. Eventually, Google recognized this as problematic and made these persistent identifiers difficult to access, replacing them with resettable identifiers like the Advertising ID, which can be reset or deleted by the user at any time. However, such resettable identifiers can be combined with other forms of user or device information (e.g., ZIP code [6] and `android.os.Build` data [7]) to create a unique user profile that overcomes the potentially ephemeral nature of these identifiers.

On the web, tracking was traditionally done using cookies, which were persistent pieces of information stored by the browser and bound to a specific website. However, recent years have seen the rise of stateless tracking techniques like fingerprinting to counteract the prevalence of anti-tracking countermeasures. Fingerprinting uniquely identifies users by leveraging APIs present in modern browsers for directly or indirectly collecting information about the user's software and hardware configuration [8]. In essence, no persistent identifiers are set, but rather, the website computes a unique identifier during each visit that will match that of previous visits if the device's configuration remains unchanged.

**Mobile advertising.** Mobile advertising has been a persistent feature of the Android ecosystem since its inception. In its early stages, advertiser-controlled code was often executed directly on users' devices, exposing them to considerable privacy and security risks. To mitigate these concerns, ad SDKs transitioned to using WebViews as a means of isolating third-party advertiser content. This shift introduced a degree of sandboxing that helped address several security and privacy issues.

Over time, the WebView sandboxing model has been significantly strengthened, benefiting from ongoing advancements in the Chromium project, particularly in regards to origin isolation and overall security posture. However, far less attention has been devoted to mitigating privacy threats unique to the WebView environment, such as cross-context data leakage and unauthorized communication via interfaces exposed by host applications (discussed in detail in the following subsection). These classes of attacks were first highlighted by Son et al. [9]. Meanwhile, advertising SDKs have evolved to deliver increasingly complex advertisements and integrate more invasive tracking mechanisms, many of which require access to privileged data and APIs only available through the Android system. This evolution has gradually undermined the original isolation guarantees of the WebView model, leading to a growing entanglement between untrusted advertiser content and sensitive app functionality.

**Android WebViews.** WebViews are embedded browsers in Android used to display ads, load web content, and render HTML/CSS/JS within apps. They can be instantiated in Java using the built-in `WebView` class, either by the app developer

**App.java**

```java
1 public class WebViewActivity extends AppActivity {
2     protected void onCreate(Bundle savedInstanceState) {
3         webview.loadUrl("https://example.com");
4     }
5     private void injectBuildId() {                ①
6         String jsCode = "javascript:window.buildId = '" +
7         Build.ID + "';";
8         webView.loadUrl(jsCode);
9     }
10    private void fetchLocationAndInject() {        ②
11        String jsCode = "window.locationData = '" +
12        location.getLatitude() + "';";
13        webView.evaluateJavascript(jsCode, null);
14    }
15    public class WebAppInterface {
16        @JavascriptInterface
17        public String getAdID() {              ③
18            return adInfo.getId();
19        }
20    }
21 }
```

**example.com/index.html**

```html
1 <script>
2 window.addEventListener('load', () => {
3     function exfiltrateData() {     ④
4         const data = a.getAdId() +
5         ',location=' + encodeUri(window.locationData) +
6         ',buildId=' + window.buildId;
7         navigator.sendBeacon('https://evil.com/data=' +
            ↪  data);
8     }
9     exfiltrateData();
10 })
11 </script>
```

Blue represents a flow of information from Java to JavaScript
Purple represents a flow of information from JavaScript into the web

Listing 1: Example of an Android app that performs privacy-compromising actions through WebViews.

or indirectly through third-party SDKs. These SDKs often leverage WebViews for ads, authentication, and other features, sometimes without the developer's explicit control [10]. Listing 1 illustrates an app constructed using techniques that we have observed in the wild. The first code snippet shows the Java code for embedding a WebView in an Android app. The app developer instantiates a WebView inside the `onCreate` function (lines 3-13), enables JavaScript by calling `setJavaScriptEnabled` on the settings object, and loads third-party web content using `loadUrl`. Direct interaction between Java code and the WebView's JavaScript context is not possible without explicitly establishing a Java-to-JavaScript bridge, via developer or third-party SDK configuration. This separation is crucial, as it prevents the app from extracting information from sensitive web resources such as cookies, `localStorage` data, or page contents, even if the developer controls the loaded domain.

**JavaScript code.** The HTML/JavaScript code in the second snippet shows the JavaScript code of a webpage loaded by the app. The embedded JavaScript code executes when the page loads, and triggers other functions that handle page interactivity. Importantly, this JavaScript code does not inherently have access to variables from the Java execution context, meaning it cannot directly retrieve sensitive system information such as build details, network status, or arbitrary file contents. However, unlike standalone browsers, WebViews introduce a critical security gap by enabling direct interaction between Java and JavaScript code [11]. The app can invoke `evaluateJavascript` (see Listing 1 – ②) to run arbitrary JavaScript inside the WebView or abuse `loadUrl` (see Listing 1 line 8) to inject custom HTML, CSS, or JavaScript. Additionally, WebViews allow JavaScript code to call Java functions via the `addJavascriptInterface` method, creating a bridge between the two execution contexts. When JavaScript invokes a method annotated with `@JavaScriptInterface` (see Listing 1 – ④→③), the Java code executes it and can use the provided values for privileged actions within the Java environment (Listing 1 – ③).

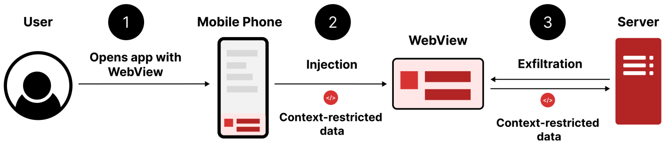


Fig. 1: Flow of privileged information across three stages.

**Cross-context interactions.** These powerful cross-context interactions introduce significant and atypical privacy risks to the WebView environment. The Java code can inject sensitive or tracking data that is only available to the JavaScript context (e.g., identifiers or location information) into the WebView (Listing 1 injections in [①, ②, ③]→④]), which can then be exfiltrated to the ad network or other third parties through JavaScript (see the function at ④). Detecting this behavior with existing dynamic and static analysis techniques prevalent in the Android ecosystem is challenging, since most focus on analyzing the Java code inside the apps. As such, we propose a dynamic analysis system that addresses this challenge by offering deeper visibility into the execution of JavaScript code within WebViews, enabling the characterization of the behaviors seen inside these cross-context communications that traditional analysis techniques often overlook. Throughout the paper, we focus on context-restricted data, which is data that could not be accessed without Java-to-JavaScript interactions, as seen in Table V (e.g., Advertising ID, GPS Location, Build ID). Furthermore, in our analysis, we label as third-party i) Java libraries/SDKs whose package name differs from that of the host application, and ii) any JavaScript code not served from the WebView's original domain.

## III. Threat Model

Our threat model focuses on how privileged information flowing from an app into WebViews may be exposed to third parties through embedded JavaScript code. We assume that the adversary controls the JavaScript loaded into the WebView, either by directly serving remote content or by embedding scripts that exfiltrate injected data. The adversary can either collude with an SDK or exploit JavaScript APIs to "coerce" the Java SDK into revealing context-restricted data. From our perspective, the SDK essentially facilitates the exfiltration of context-restricted data by performing a potentially dangerous practice, leading to privacy loss. We do not assume that the adversary can compromise the app's Java code or the underlying operating system. Figure 1 illustrates the flow of privileged information across three stages.

❶ The user launches an app that contains a WebView, typically unaware of the internal mechanics. This WebView can be initialized by the app developer or indirectly through a third-party SDK, which configures the WebView and determines the content it loads. Critically, these WebViews often enable JavaScript and fetch remote content, providing an avenue for untrusted code to execute within the app context.

❷ Once the WebView is initialized, the app or integrated SDK can inject privileged information into the JavaScript context using Java APIs such as `evaluateJavascript` or `addJavascriptInterface`. This data can include tracking identifiers like the Advertising ID, or environment-specific signals such as the partition space or available memory, signals that would normally be inaccessible to JavaScript running in a standalone browser. While such injections may serve legitimate purposes (e.g., allowing an ad SDK to tailor content or perform analytics), it also introduces a bridge for exchanging information between Java and JavaScript that is not rigorously policed by the platform.

❸ JavaScript running inside the WebView can then access the injected data and transmit it to external servers. There are two possible cases to consider here: (i) JavaScript is controlled by a third party (e.g., an ad network or embedded analytics provider) that opportunistically exfiltrates the injected data, potentially without the knowledge or consent of the app developer or SDK that performed the injection. (ii) the SDK and the external server may be cooperating, with the SDK deliberately injecting tracking information into the WebView to facilitate collection by known JavaScript endpoints. This type of collusion resembles data-sharing patterns seen in web advertising, such as cookie syncing [12] or redirect-based profiling [13]. In both cases, a third-party actor that previously lacked access to platform-specific tracking identifiers can now obtain and exfiltrate them. This occurs due to the tight coupling between the privileged Java context and the WebView's JavaScript environment, often without strong oversight or meaningful user consent. Our work focuses on detecting and analyzing the information flows between the Java and WebView contexts. We do not attempt to distinguish between intentional misuse and unintentional leakage. Instead, we aim to provide empirical visibility into how these cross-context interactions unfold in real-world apps, and highlight dangerous and invasive behaviors regardless of the original intent (i.e., we focus on the outcome not the intent).

## IV. WebViewTracer Design and Implementation

In this section, we describe WebViewTracer's architecture and analysis workflow. Figure 2 provides a high-level overview of our design, and illustrates the various components that comprise our system, which we designed to dynamically analyze JavaScript code executed within WebViews. By building upon existing technologies, we have developed a novel framework capable of crawling and studying cross-context interactions between Java and JavaScript within mobile apps, providing a unique view of the JavaScript code running within WebViews.

### *A. Dynamic App Exercising and Analysis*

The first phase of our analysis workflow focuses on automatically exercising and dynamically analyzing Android apps, which we achieve by modifying existing systems, allowing us to dynamically bridge the semantic gap between interactions from Java to JavaScript and vice-versa. Specifically, (i) we modify the UIHarvester app-crawling framework to fit our needs for recording cross-context interactions [14], (ii) create custom Frida gadgets responsible for hooking WebView-based Java functions inside Android apps, and (iii) modify VisibleV8 to run inside the rendering process of an app-loaded WebView by compiling our own version of a system-wide VisibleV8 WebView provider. Furthermore, to avoid any device or Frida disconnections, we created an execution wrapper responsible for handling and monitoring an Android device using the Android Debug Bridge, managing our Frida server and gadgets, and automatically exploring and traversing Android applications. Finally, a Python-based distributor script alongside Docker is used as the orchestration layer to parallelize our crawls across multiple devices. Next, we provide more details about how we modified and integrated these components into WebViewTracer.

**Dynamic app exercising.** Reaper's UIHarvester [14] module enables the automated exploration of Android applications using a breadth-first search of the app's UI elements and provides better coverage over other tools (e.g., Monkey) [14]. We modified UIHarvester to identify both native Android and web elements by registering it as an `AccessibilityService` [15] and monitoring Android `AccessibilityEvents` whenever the display changes. These modifications enhance coverage, are Android version-independent, and allow our variation of UIHarvester to be installed like any typical Android application. The inspected UI elements along with their properties (e.g., `isClickable`), which are represented as an `AccessibilityNodeInfo`, are exported using `logcat` and are placed in a `UIQueue` along with the set of interactions that need to be performed so as to reach that element, within the app's UI element hierarchy. A custom crawling Python parser processes the elements from the queue performing a breadth-first search of the app's UI elements, and exploring new application paths. While we adopted a breadth-first traversal strategy in our experiments due to its effectiveness in maximizing UI coverage (e.g., [14]), our framework also supports alternative strategies, including DFS, monkey testing, and an experimental record-and-replay feature. As several applications require the user to login, we also automated the login process by using Google's SSO, when available.

**Frida gadgets.** We created several gadgets using Frida [17] to trace calls to Java methods that interact with WebViews. Our hooking gadgets include functions such as `evaluateJavaScript` and `loadUrl`. Frida logs are used to analyze

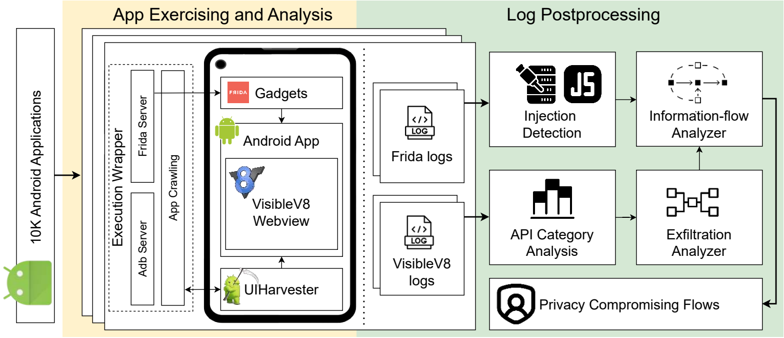


Fig. 2: WebViewTracer's architecture and analysis workflow.

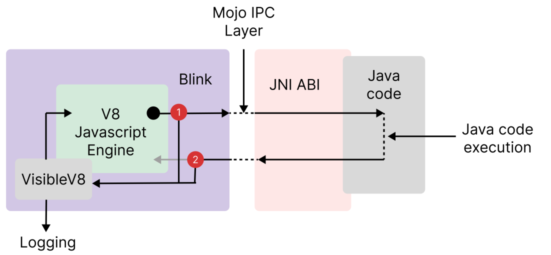


Fig. 3: We monitor JS interface calls by hooking two points: before the call exits V8 into JNI, and after the return value flows back into Blink.

TABLE I: Injection techniques detected by WebViewTracer.

| Injections | API |
|---|---|
| JS code execution | `evaluateJavaScript`, `loadUrl` |
| HTML spoofing | `loadDataWithURL`, `loadData` |
| Java-Javascript bridges | `addJavascriptInterface`, `evaluateJavaScript` |
| URL Leakage | `loadUrl`, `postUrl` [16] |

and correlate data gathered from VisibleV8 to understand how Java and JavaScript interact in the WebView environment. Additionally, we identify whether third-party libraries are responsible for any WebView activity, by analyzing the APIs' stack trace, allowing our system to have a holistic view of Java-to-JavaScript interactions and vice-versa.

**VisibleV8 WebView provider.** VisibleV8 [5], [18] is an instrumented version of Chromium's V8 JavaScript engine that enables the logging of webpages' JavaScript execution. It records the execution of JavaScript code and all web API calls, including function parameters. To integrate VisibleV8 into our system, we extended its functionality to work within Android's WebView environment. Specifically, we developed a system-wide VisibleV8-enabled WebView provider by recompiling the Android System WebView APK [19] to include a customized version of the VisibleV8 patchset. On Android, the default WebView implementation used by apps to render web content via the `WebView` object is provided by this System WebView APK. Changing the WebView provider applies system-wide and affects all apps that rely on WebView. Our integration is feasible because the default WebView provider is based on Chromium, which uses V8 as its JavaScript engine.

Adapting VisibleV8 for Android WebViews was non-trivial due to its original design, which heavily favored desktop compatibility. VisibleV8 logs extensive JavaScript execution data to disk and relies on debugging flags like `-no-sandbox`, which are incompatible with Android's strict sandboxing policies. Using this setup with WebView causes execution failures and severe performance degradation. To make VisibleV8 suitable for Android WebView, we reworked its patches (see Appendix A for technical details). The latest upstream patchset showed suboptimal performance in mobile environments, limiting its utility for logging and tracing. Our modifications focused on optimizing performance by pruning unnecessary instrumentation. Specifically, we restricted logging and tracing to objects defined by the ECMAScript standard [20] and Chrome's implementation of the W3C Web IDL standards [21], [22].

We also introduced new instrumentation features to trace cross-language interactions, particularly focusing on return values of JavaScript functions that bridge into Java via `addJavaScriptInterface`. Tracing this path is essential for analyzing hybrid app behavior and understanding the flow between Java and JavaScript. Achieving this required understanding the path that JavaScript objects take across several runtime boundaries, starting from the Java Native Interface (JNI), through the Blink rendering engine, and into the V8 JavaScript engine. Chrome's WebView architecture employs a custom IPC mechanism between the JNI-based Java ABI and the rendering process, which houses Blink and uses V8 as an embedded library to execute JavaScript code. As shown in Figure 3 our instrumentation hooks into JavaScript interface function calls in two places, ❶ when the function is called, to get the values being passed into Java and once at ❷ where it intercepts the return value just after it emerges from the IPC boundary into Blink, enabling us to observe and trace the object before it is passed into V8 for execution.

One major challenge was VisibleV8's reliance on the `-no-sandbox` flag, as it writes execution logs to physical files. This conflicts with Android WebView's strict process of sandboxing. We experimented with several alternatives, including socket-based logging and IPC with a dedicated Android service. However, these approaches introduced severe performance overheads that slowed down the logging and tracing. Ultimately, we modified the patchset to set an undocumented incremental install flag [23] and configured the device to use the `disable-multiprocess` [24] option in Android. This relaxed WebView's sandboxing enforcement, enabling file read/write operations within the rendering process. These changes allowed us to successfully trace JavaScript interactions with `addJavaScriptInterface` objects injected by Java. We are committed to open-sourcing our modifications to the VisibleV8 patchset to support further research.

TABLE II: API categories analyzed by WebViewTracer.

| API Category | Description |
|---|---|
| Java APIs | Calls into Java code |
| Exfiltration APIs | Used to transmit or store data |
| Sensitive APIs | Used for accessing local storage/state |
| Fingerprinting APIs | Used to uniquely identify devices or users [25] |
| Canvas | Uses known canvas fingerprinting APIs |
| Navigator-based | Queries `navigator` properties |
| Advanced | Audio fingerprinting, font detection, etc. |
| Sensor data | APIs accessing device sensors |
| Obfuscation APIs | Used to disguise data being transmitted |
| Custom JavaScript APIs | APIs not part of Web IDL standards |

### *B. Log Analysis*

The second phase of our workflow consists of the analysis of the logs generated during the dynamic app exercise and analysis phase. The log postprocessor consists of four individual modules that perform parts of the analysis.

**Injection Detection.** The injection-detection module processes raw Frida logs to identify Java-to-JavaScript injection mechanisms (an example of a Frida log where an app loads a Google Ads domain and injects an empty function into the WebView to check if the JS has loaded can be seen in Figure 6 in Appendix E). We categorize these interactions into four types, as detailed in Table I, which we have uncovered through our empirical analysis of WebView use in the wild. In our context, an injection refers to any instance where JavaScript gains access to information originating from the Java runtime. With this module, we cover four broad ways in which Java was injecting code into WebViews:

*JavaScript code execution.* Some apps execute JavaScript within an existing WebView using Java code, typically through calls like `evaluateJavaScript` or by loading a `javascript:` URI.

*HTML spoofing.* Android apps may use `loadDataWithBaseURL` to inject custom HTML, CSS, and JavaScript into a WebView under a spoofed URI, effectively faking the origin and displaying app-controlled content.

*Java-JavaScript bridges.* These expose Java objects to JavaScript running inside a WebView, most commonly via `addJavaScriptInterface`. The module flags such uses as potential interface definitions. It also detects less conventional patterns, for instance, cases where apps inject JavaScript that interacts with custom-defined objects or APIs that fall outside the WebIDL specification [21], [22], thereby avoiding the use of `addJavaScriptInterface` entirely.

*URL leakage.* This occurs when Java-controlled methods load URLs into WebViews that embed Java-only data, such as Advertising IDs into the URL itself. Such data may then be accessed by JavaScript through URL parsing or by being reflected back into JavaScript by the server. These behaviors are also flagged by the module as injection attempts.

**API category analysis.** This module processes VisibleV8 logs to identify API usage patterns across six broad categories, as shown in Table II. Java APIs are detected using a custom annotation present in the VisibleV8 logs. Custom JavaScript APIs are identified by comparing all invoked APIs against the ECMAScript standard [20] and Chrome's implementation of the W3C Web IDL standards [21], [22]; any API not included in these specifications is classified as custom. Exfiltration APIs are based on the list curated by Su et al. [25] in their study on fingerprinting. We extended this list by manually inspecting each API using Mozilla Developer Network (MDN) documentation. Each API was classified into one of two categories: (i) those that perform actual data exfiltration by transmitting information to external servers (e.g., via network requests), and (ii) those that only manipulate the internal WebView state (e.g., cookies or localStorage). During our initial crawl, we also discovered three previously undocumented exfiltration APIs, as well as a novel exfiltration vector involving Java-side APIs. These findings, discussed in Section VI, reveal how sensitive data is transmitted by certain apps using unconventional Java methods, or persisted within the WebView's state for deferred access. Fingerprinting APIs were initially sourced from Su et al.'s list [25], but we further refined this set by categorizing them into subtypes using MDN documentation and techniques recently identified by Nguyen et al. [26]. These subcategories include *Navigator-based fingerprinting* (i.e., basic device and browser attributes), *Sensor-based fingerprinting* (e.g., accelerometer, gyroscope), *Canvas fingerprinting*, and *Advanced fingerprinting* techniques involving APIs such as AudioContext, WebGL, and others beyond standard profiling methods. Sensitive APIs include all APIs that change browser storage (cookies, local/sessionStorage). Obfuscation APIs include SubtleCrypto, TextDecoder and base64 APIs. While JavaScript code could evade the detection of obfuscation APIs, we use it to understand the origin of phantom exfiltrations (see Section VI), not to measure the use of these APIs through Java-JavaScript bridges.

**Exfiltration analysis.** This module builds upon the API category analysis to identify and investigate cases of data exfiltration involving Java APIs. During our initial empirical analysis, we observed that some apps define custom Java interfaces that are later invoked by the WebView to initiate network requests to remote servers, as shown in Figure 7 in Appendix E. The module parses the arguments passed to various known exfiltration and Java APIs to determine whether actual exfiltration occurs, meaning that the URI points to a real web address rather than a custom URI handled within the app. When exfiltration is detected, the module extracts both the destination URLs and the associated payloads. To group similar domains under common ownership, we use the DuckDuckGo Tracker Radar dataset [27], which maps related domains to "parties" that represent real-world entities. The mapping is done by comparing the eTLD+1 portions of the extracted URLs to known party domains in the Tracker Radar dataset.

**Information flow analyzer.** This component aggregates inputs from all modules and employs regex-based heuristics to identify common types of context-restricted data (see Table V). These heuristics are evaluated on the data flagged during the exfiltration analysis step and the injection detection step. This includes the exfiltrated data, arguments passed to JavaScript interface calls, return values from JavaScript interfaces, and

TABLE III: App dataset and breakdown of detected behaviors.

| Metric | Apps |
|---|---|
| Initial dataset | 10,000 |
| Loaded WebViews | 4,597 |
| Performed injections | 4,366 |
| Injected context-restricted data | 2,989 |
| Exfiltrated context-restricted data | 2,711 |

other detected injections. To attribute the origin of these injections, the analyzer uses a combined list of third-party library package names based on prior research [28], [29], [30] and examines the source file and package name responsible for the corresponding Java API method. Specifically, it takes as input stack traces obtained from Frida and identifies the origin (first/third-party) of the Java injections (source). Next, the information flow analyzer traces the origin and recipients of data within the system, identifying how information is distributed between multiple parties. Using VisibleV8 logs, we verify that injected scripts were executed and identify JavaScript APIs that exfiltrate context-restricted information using our API category analysis. Subsequently, we parse the arguments of the exfiltrating APIs to identify where the data was sent (sink). Researchers can query the analyzer to investigate privacy-violating information flows through WebViews, providing insights into potential data exfiltration risks.

## V. Experimental Setup and Evaluation

**Dataset.** We applied our framework to a dataset of 10K Android apps randomly sampled from the AndroZoo dataset [31]. To ensure diversity, we divided the AndroZoo dataset into three sections based on their popularity —top, middle, and bottom— and evenly selected apps from each section. To avoid selecting a disproportionate number of gaming apps (which dominate the Play Store), we opted to balance the dataset by including 1K game apps and 9K non-game apps from the three sections. To download the dataset, we used a physical Android phone with an active Google account to obtain Google Auth tokens and a user-agent string. These credentials were then used with the Raccoon app downloader [32] to bulk-download the apps listed in AndroZoo during February 18-20, 2025.

**Measurement setup.** We install VisibleV8 WebView on a phone connected to a Docker container running an instance of our modified version of UIHarvester (see Section IV-A). UIHarvester explores the app for the predefined crawling period, visiting different app areas so as to load WebViews. After the crawl ends, the system dumps two sets of logs, a Frida log, and a VisibleV8 log, both of which are subsequently processed to extract information about the execution of Java code concerning WebViews, and JavaScript code, respectively.

**Measurement infrastructure.** Measurements took place from March 2025 to April 2025. The crawling was conducted using four Pixel 4a (5G) phones running stock Android 13 (build number TQ3A.230901.001). Each phone was rooted with Magisk [33] and equipped with the Frida gadget [17], along with a custom WebView version based on VisibleV8. The phones were connected to an Intel i7-10700 machine with 30 GB memory and 10 TB storage, running Ubuntu 24.04.1. We used a separate IP space for measurement studies, belonging to a US academic institution.

**Evaluating regex-based heuristics and coverage.** We conducted two separate experiments to evaluate WebViewTracer's capabilities and coverage. First, we evaluated the effectiveness of the information flow analyzer's regex-based heuristics. The second experiment focused on assessing UIHarvester's effectiveness in identifying WebViews. Further experimental details can be found in Appendix C and Appendix D.

The first experiment quantifies the presence of false positives and negatives of our regex-based heuristics. We randomly selected a set of 60 apps and manually inspected the Frida and VisibleV8 logs for privacy-invasive behavior. Our manual evaluation conducted by five researchers (all with backgrounds in security and experienced in JavaScript code analysis), demonstrates that WebViewTracer effectively identifies injections, achieving an average precision of 98.2% and recall of 96.4%, with only a few false positives.

In the second experiment we manually exercised 50 apps and annotated any WebView that was displayed on the screen. We then used the same setup and exercised these apps using UIHarvester. Overall, we found that UIHarvester achieved 49.64% WebView coverage compared to manual analysis, and 52.13% WebView coverage for non-game apps. In prior work, DROIDAGENT [34] and Humanoid [35] achieved activity coverage rates of 61% and 51%, respectively, when evaluated using the manually validated THEMIS benchmark [36], which was specifically designed to assess automated GUI testing tools against real-world bugs. While they focus on measuring activity coverage, whereas we focus specifically on the identification of WebViews, the overall coverage levels they report are comparable to those of our system. While these numbers may appear modest, they are in line with other coverage systems, and we believe that automated approaches are the only viable solution for large-scale analyses. Moreover, our findings concerning privacy-invasive activities in WebViews should be interpreted as a conservative lower bound.

## VI. Measurements and Analysis

In this section, we characterize the outflow of context-restricted information from apps through WebViews. Due to the way our system is set up, we are able to observe these types of exfiltration from two vantage points: (i) where the data is injected into WebViews, and (ii) where the data is exfiltrated over the network. Overall, out of the 4,597 apps that loaded WebViews, we see 65% apps injecting context-restricted data into WebViews and 59% apps exfiltrating information to web servers. A summary is provided in Table III.

**WebView injections.** 4,597 apps loaded WebViews that we were able to trace during the crawls. We found that 95% of these apps contained Java code that performed some form of injection into a WebView. Table IV presents the distribution of the APIs used for these injections. The most frequently used methods were `loadUrl` (commonly used for

TABLE IV: Unique number of apps that call each Java API.

| Java function | Apps | Java function | Apps |
|---|---|---|---|
| loadUrl | 4,243 | loadData | 581 |
| addJavascriptInterface | 3,923 | getUrl | 538 |
| evaluateJavascript | 3,670 | reload | 35 |
| loadDataWithBaseURL | 2,710 | postUrl | 12 |
| destroy | 1,517 | | |

TABLE V: Context-restricted data injections and leaks across all apps that load WebViews.

| Context-restricted info | Inject (%) | Exfil (%) | Inject&Exfil (%) |
|---|---|---|---|
| AdMob SDK version | 61.64 | 60.93 | 59.57 |
| Device model | 58.79 | 49.50 | 42.21 |
| Build ID | 53.15 | 21.33 | 14.05 |
| Internal IP-related info | 33.80 | 33.98 | 31.47 |
| Advertising ID | 23.30 | 18.19 | 17.68 |
| Partition space | 22.64 | 22.06 | 10.18 |
| Battery level | 11.46 | 1.48 | 1.35 |
| Device code name | 9.72 | 0.71 | 0.29 |
| Location | 9.61 | 7.34 | 3.99 |
| Memory space | 9.01 | 10.76 | 1.65 |
| Zip code | 8.12 | 12.54 | 4.40 |
| Kernel build version | 7.32 | 0.00 | 0.00 |
| Bootloader version | 7.32 | 0.00 | 0.00 |
| City | 5.27 | 1.68 | 0.89 |
| Play Store package version | 2.78 | 5.05 | 1.98 |
| Network provider | 2.02 | 1.11 | 0.89 |
| Google Account Name | 0.11 | 0.20 | 0.04 |

```
window.nativebridge.handleCallback([
    ....
    // Unity's asynchronous injection after a
    // JS call to com.unity3d.services.core.api.DeviceInfo
    ["42", "OK", ["d3032f3db0173854806bd652576cf56cb7d08eff"
    ↪  ]],
    // Unity's device hash
    ["48", "OK", ["abfarm-release-rbe-64-2004-0093"]],
    // Kernel build version
    ["50", "OK", [
    ↪  "google\\/bramble\\/bramble:13\\/TQ3A.230901.001\\/"
        + "10750268:user\\/release-keys"]],
    // Build fingerprint
    ["52", "OK", [{
        "firstInstallTime": 1741452683184,
        "lastUpdateTime": 1741452683184,}]]
    // Install/update times, useful for fingerprinting
]);
```

Listing 2: A typical injection from the `com.unity3d` library.

URL leakages and injections), `addJavascriptInterface`, and `evaluateJavascript`, appearing in 4,243, 3,923, and 3,670 apps respectively. Overall, 2,989 of apps with WebViews in our dataset included at least one injection of context-restricted data into a WebView.

**Souce of injections.** A key characteristic of the injections we detect is their origin, identified via stack trace analysis (see Section IV). We find that the vast majority of injections do not originate from app developers' own code. In fact, 96% stem from third-party libraries included in the app's Java code. Only 3.6% of observed injections are attributable to first-party app logic. This suggests the flow of context-restricted data into WebViews is not necessarily under the direct control of app publishers and may be introduced indirectly through SDKs for analytics, advertising, or game development.

Crucially, we observed that certain third-party libraries are disproportionately involved in injection activity. The Google Ads SDK was the most frequently encountered, detected in 3,018 apps, which reflects its dominant role in mobile advertising. The Unity3D SDK, a widely used platform for game development, was found in 484 apps, while the IronSource SDK, commonly employed for monetization and user acquisition, appeared in 235 apps. Notably, 470 apps exhibited injection events that appeared to involve multiple SDKs simultaneously. For example, we observed modules from the Adobe SDK invoking APIs from the Google Ads SDK to execute JavaScript within an advertising WebView. These interactions indicate that SDKs can exhibit complex internal behaviors and engage in cooperative operations with other libraries, often in ways that remain opaque to both developers and end users, highlighting both the complexity of invasive behaviors and the need for additional safeguards.

Moreover, we find that 29 unique SDKs are responsible for all of the third-party library injections that leak context-restricted data inside WebViews during our experiments. We find that `com.applovin`, `com.google.android.gms`, `com.unity3d` and `com.ironsource` inject the most types of data into WebViews, with 19 third-party libraries injecting more than one type of context-restricted data. The AdMob SDK version is by far the most injected piece of information, followed by the device model and the Advertising ID. This suggests a small group of popular SDKs are responsible for the leakage of context-sensitive data and, as a result, have an outsized impact on user's privacy.

**Context-restricted injections.** Among all observed injections, 54.76% involved context-restricted data and were classified as sensitive by our system. Listing 2 illustrates one such case, where the Unity3D SDK injects (among other things) the kernel build version, bootloader version, a device hash, the device fingerprint, and the size of partition space into the WebView. This is done by invoking a custom API, `nativeBridge.handleInvocation`, which is defined by JavaScript inside the WebView. The remaining injections that do not explicitly involve the exfiltration of sensitive or context-restricted data are excluded from our analysis. These typically serve benign purposes such as verifying JavaScript load status, triggering events, or signaling lifecycle changes like the app entering the background. While such behavior may expose contextual signals to JavaScript, it does not constitute a direct leak of sensitive or context-restricted data.

Table V summarizes the types of context-restricted data being injected into and exfiltrated by WebViews. The "Context-restricted info" column indicates the category of leaked information, while the "Inject (%)", "Exfil (%)", and "Inject&Exfil (%)" columns show the percentage of apps exhibiting injections, exfiltration, or both, respectively. In the discussion that follows, we focus primarily on the data categories and injection behaviors, with a detailed analysis of exfiltration presented later in this section. We divide the types of context-restricted data into 15 information types, as described below, grouping related types of information together.

*Build ID.* The Build ID is an identifier that specifies the exact Android OS release installed on a device. Since manufacturers frequently customize builds for specific models, the Build ID can often be used to infer the maker and model of the phone. 53% of apps that loaded WebViews injected Build IDs into the JavaScript context. Among these, the vast majority (91%) came from the `com.google.android.gms` library, followed by `com.unity3d` at 15% and `com.ironsource` at 6.4%.

*Advertising ID.* The Advertising ID is a resettable UUID used by advertisers to track users across different apps. Although JavaScript typically cannot access this identifier, we found that 23.30% of WebViews injected it into the web context. Of those injections, 66% were performed by `com.google.android.gms`, with significant contributions from `com.unity3d` and `com.ironsource`.

*Partition space, battery level, and memory information.* Partition space, battery level [37], and memory usage can serve as high-entropy fingerprinting signals. 53% of apps showed this behavior, most often in apps using the `com.unity3d` SDK (68.53%), with `com.ironsource` also contributing (44.21%). Memory usage information was injected by 9.01% of apps, nearly all of which used `com.unity3d` (86.59%).

*Kernel and bootloader version.* These are high-entropy system-level identifiers that reflect the exact Linux kernel and bootloader version on a device. 7.32% of WebViews injected this data into the JavaScript context, with `com.unity3d` overwhelmingly responsible for these leaks (99.72%). In many cases, these values were accessed and passed to JavaScript at the request of ad-related scripts loaded within the WebView.

*Network information.* Internal IP addresses were injected into JavaScript by 34% of apps, mostly due to `com.google.android.gms`. These can reveal local network setups, such as whether a device is on a campus or corporate network. Network provider names (e.g., ISP) were exposed by 2.0% of apps, led by `com.ironsource`, `com.applovin`, and `com.google.android.gms`. In some cases, these leaked names pointed to specific universities or carrier types, revealing information about socio-economic status and other user context.

*Device code name and model.* Low-entropy identifiers such as the device code name, and model were injected by 59% of apps. For device code names, `com.unity3d` accounted for 84% of injections. Device model values were injected by 92% of apps using `com.google.android.gms`, followed by `com.unity3d` (14%) and `com.inmobi` (2.9%). Interestingly education apps were more likely to use device model information over other forms of context-restricted data, possibly as a way to comply with privacy legislation.

*Location and Zipcode.* Precise geographic information, such as location and zip codes, was injected into the WebView context by 9.59% of apps. Among these, `com.google.android.gms` was responsible for 54.48% of location injections and 87.71% of zip code disclosures. `com.unity3d` appeared in 6.98% of zip code injections, while `com.ironsource` contributed 37.06% of location disclosures. This type of data enables location-based tracking and, when combined with other identifiers, poses a heightened privacy risk since it can be considered personally identifiable information.

*Google Ads SDK version and Play-store package information.* The version of the Google Ads SDK is typically hardcoded. When combined with other identifiers, such as the app name, it can act as a high-entropy identifier. This version string was injected by 62% of apps with WebViews. All these injections originated from `com.google.android.gms`. Given Google Ads' dominance, as a standalone ad library and as a dependency bundled by many third-party ad SDKs, this identifier was among the most frequently injected into the JavaScript context, likely enabling in-context scripts to correlate user sessions across different apps. 2.8% of apps inject the Play Store package version, an identifier only accessible to Google-owned SDKs.

While many categories of sensitive data discussed above are not accessible to JavaScript code, JavaScript can still directly access some of this data via dedicated browser APIs. For instance, the `BatteryManager` interface allows querying the device's battery status, `Performance.memory` exposes memory usage details of the rendering process, and `navigator.geolocation` provides access to a user's physical location. However, these interfaces are subject to increasing restrictions due to their potential for fingerprinting and privacy abuse [37]. Modern browsers like Chrome restrict access to these APIs in insecure contexts, or within iframes, which are governed by restrictive permission policies. In several of the injection cases we observed, JavaScript code received access to these types of high-granularity system-level details, regardless of what context they were loaded in, sometimes exceeding what the standard APIs would normally expose. This strongly suggests a bypass of context-specific restrictions put forth by the browser and a break in the expectation of what kinds of JavaScript code would have access to this information. Nonetheless, we use the term "context-restricted data" for all these pieces of data, for conciseness.

**App popularity.** We see a correlation between the apps that inject context-restricted data into WebViews and their popularity. In more detail, apps with higher install counts have a higher chance of injecting such information. For instance, only 0.1% of apps in the 1–1K install range inject the play store package version, compared to 7.5% of apps with 10M+ installs. Similarly, injection of the Advertising ID rises from 13.7% in the lowest install range to 30.3% in the highest. Injection of the device model increases from 46.7% to 61.8%, and memory space from 23.0% to 31.8%. The injection of the AdMob SDK version climbs from 40.2% in the least popular apps to 70.9% among the most popular. This trend suggests that popular apps are more likely to inject a broader set of identifying or environment-specific information into WebViews.

**App categories.** Apps across categories inject different types of identifiable or sensitive data into WebViews, revealing patterns that reflect the priorities and SDK integrations typical to each domain. Games apps stand out as the most aggressive in terms of injections, with 73% injecting the AdMob SDK version and 73% injecting the device model and build ID suggesting a strong reliance on tracking and device profiling.

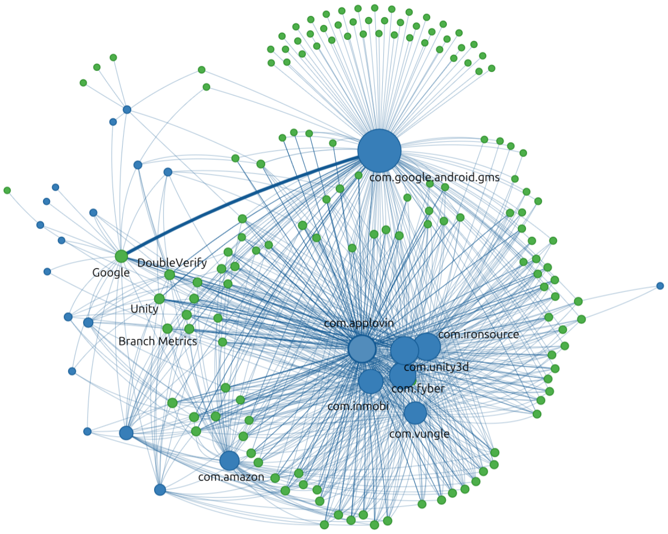
TABLE VI: Number of apps using each exfiltration API.

| Exfiltration API | Apps | Exfiltration API | Apps |
|---|---|---|---|
| XMLHttpRequest.open | 3,992 | Navigator.sendBeacon | 2,443 |
| XMLHttpRequest.send | 3,887 | HTMLIFrameElement.src | 1,709 |
| HTMLScriptElement.src | 3,463 | HTMLVideoElement.src | 686 |
| Window.fetch | 3,004 | HTMLLinkElement.href | 510 |
| Window.postMessage | 2,952 | Location.href | 196 |
| HTMLImageElement.src | 2,935 | HTMLSourceElement.src | 177 |
| Image.src | 2,653 | WebSocket.send | 73 |

Other categories show similarly troubling patterns. Photography apps sometimes inject the amount of memory space available on the phone (11%), kernel build versions (7.9%), and bootloader versions (7.9%). Entertainment apps follow closely, with 8.5% injecting memory data and 6.7% including kernel and bootloader versions. Games, Entertainment and Photography apps frequently request broader device permissions, allowing them to inject memory state, kernel versions, and other low-level identifiers into WebViews. This reinforces our finding: injection behavior is not uniform but highly category-specific, shaped by the SDKs apps rely on and the permissions users commonly grant. For example, Maps and Navigation apps frequently inject network-level and location data, with 9.1% injecting location information, 22% including the Advertising ID, and 25% exposing internal IP addresses. Maps and Navigation apps are often explicitly given access to location, which makes location-related injection more feasible.

**API usage.** From the WebViews that were loaded, 92% used at least one exfiltration API call. Table VI shows the breakdown of the different APIs used for exfiltration. Alongside the traditional exfiltration APIs on the web, we see the emergence of three other APIs, `HTMLScriptElement.src` (used across 75% of apps that loaded WebViews), `HTMLVideoElement.src` (used across 16% apps) and `HTMLSourceElement.src` (used across 3.8%). The prevalence of `HTMLScriptElement.src` suggests that many apps are dynamically loading scripts into WebViews, a common obfuscation technique that makes the newly added scripts undetectable to static analysis methods employed in prior work, such as HybridDroid [3] and LuDroid. [16]. The use of video-related APIs for exfiltration may be linked to mobile advertising practices, where videos are often dynamically fetched and rendered. This behavior exceeds the level of dynamic media loading typically observed on the web. These APIs have not been identified in previous work related to web fingerprinting and it's subsequent exfiltration, including the study by Su et al. [25] and Boussaha et al. [38].

We also observed that 65% of apps that load WebViews use Java APIs to exfiltrate context-restricted information out of the app bypassing typical exfiltration detection techniques on the web. We observed JavaScript code invoking Java methods that, in turn, triggered network requests through internal classes, either from the app itself or bundled third-party SDKs. On manual inspection we found that these behaviors resulted in the app making direct requests to specified URLs outside the context of the WebView.



Each green node represents a company to which data was exfiltrated.
Each blue node represents a third-party library which injected data.

Fig. 4: Network graph of relationships between SDKs/parties.

**Exfiltrated data.** We observe notable trends in the types of context-restricted data exfiltrated via JavaScript, as shown in Table V. While some categories such as the AdMob SDK version exhibit consistently high exfiltration rates (61%), others like the Build ID (21%) and Battery level (1.5%) are far less frequently exfiltrated. Information such as internal IP-related data (34%), partition space (22%), and coarse location (7.3%) are also exfiltrated with non-trivial frequency, indicating attempts to harvest environmental and network-level context from devices. Interestingly, the percentage of apps where we observe both a JavaScript injection and an associated exfiltration (Inj+Exfil%) is consistently lower than the standalone exfiltration rates for several data types, a phenomenon we refer to as phantom exfiltrations. For example, while 33.98% of apps exfiltrate internal IP-related information, only 31.47% exhibit both injection and exfiltration. This discrepancy suggests that some exfiltrations occur without a directly detected JavaScript injection, potentially due to obfuscation, dynamic code loading, or alternative communication paths. One plausible mechanism is that sensitive data is first collected natively (e.g., via Java), and then sent to a remote third-party server, which subsequently reflects the data into the web context using identifiers like device IDs or session tokens. It is important to note that our current approach of parsing the content of JavaScript injections offers a conservative low-bound estimate. More advanced techniques, such as taint tracking, would uncover a wider range and greater volume of such leakages.

**Exfiltrating parties.** Figure 4 illustrates the connections between parties involved in the flow of context-restricted data within WebViews. The most connected node, `com.google.android.gms`, has 147 connections to various domains, most of which lie outside of Google's ecosystem. Overall, a small number of libraries are responsible for enabling large-scale data dissemination. We observe 25 third-party

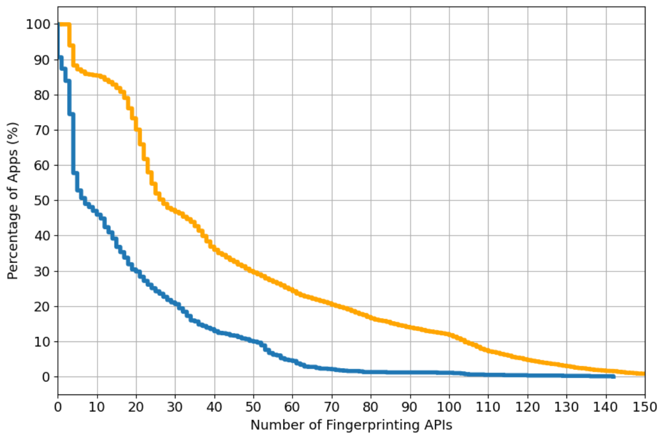
SDKs injecting context-restricted data into WebViews, which is subsequently exfiltrated by scripts on the loaded pages to 152 distinct external parties. For example, the Amazon Ads SDK leads to injected data being exfiltrated to over 52 different domains, demonstrating the widespread diffusion of context-restricted information. In contrast, some libraries such as com.indeed, com.nvidia, and in.juspay load WebViews and inject data, but the exfiltration is limited to their own servers, suggesting a more contained data usage model.

To the left, we observe a cluster of well-connected third-party SDKs commonly used for advertising. These include com.ironsource, com.unity3d, com.inmobi, and com.fyber, all of which inject data into WebViews that are subsequently exfiltrated to a significant number of the same external parties. The overlap between com.ironsource and com.unity3d is expected, as both are owned by the same parent company. However, similar exfiltration patterns from WebViews loaded by unrelated SDKs such as com.inmobi and com.fyber suggest systemic data-sharing or common script inclusion within the advertising ecosystem.

We find that information exfiltration is often not limited to a single external party for a given app. Over 22% of apps that engage in exfiltration send data to two or more external domains. Moreover, many of these recipient domains are shared across apps, implying the existence of persistent tracking infrastructure. We find that 63 distinct recipient parties appear in more than five apps each, and 43 parties show up in more than 10 apps, indicating that a large number of companies have a significant breadth of visibility into many apps simultaneously. Among the top companies, besides Google and Unity, we found that Integral Ad Science and DoubleVerify had a presence across 303 and 251 apps, respectively, and were exfiltrating context-restricted data despite not having any associated SDK.

We also find that 25 parties (companies) involved in exfiltrating information from WebViews are assigned a tracker-radar score of 0 by DuckDuckGo (in their DuckDuckGo tracker-radar [27]), indicating that they are not recognized as tracking domains on the web. Notably, this includes Aarki, a mobile marketing platform observed in 109 apps, which exfiltrated 15 types of context-restricted data during our crawl. Similarly, RevX, an ad tech company present in 67 apps, also exfiltrated 15 types of context-restricted data yet is absent from DuckDuckGo's tracker-radar dataset. These cases point to a set of lesser-known or overlooked companies operating in the mobile tracking ecosystem that collect sensitive data from WebViews but are not flagged by mainstream web-focused tracking detection tools, like DuckDuckGo's tracker radar.

**Fingerprinting.** We find that JavaScript code inside apps that participate in information leakage tends to use more fingerprinting APIs compared to apps that do not. This is illustrated in Figure 5 which depicts two survival functions, one for the number of fingerprinting APIs per app, for apps with injections and exfiltration, and another for the number of fingerprinting APIs per app but calculated for apps where no injection or exfiltration occurred. We find that over 30% of apps that exfiltrate context-restricted data load scripts that, on average, invoke more than 50 fingerprinting APIs. In contrast, among apps where no injection or exfiltration occurs, only 10% execute more than 50 APIs on average. The most widely used fingerprinting APIs are Navigator.userAgent and Navigator.userAgentData, which appear in 3,079 and 2,984 apps, respectively. These APIs are well-known fingerprinting vectors in the web fingerprinting landscape. Additionally, previous work by Tiwari et al. [16] has shown that these APIs can also contain context-restricted data since they can be modified by the Java code by [39].



Orange is for apps that had injections and exfiltrations
Blue is for apps that had no injections or exfiltrations

Fig. 5: Fingerprinting API usage across apps that had injections and exfiltration of context-restricted data compared to apps that had no injections and exfiltration of context-restricted data.

Other commonly used fingerprinting techniques include accessing Screen.width and Screen.height to infer device screen size and leveraging the navigator.connection interface to determine the quality of the device's network connection. Among other techniques, we find evidence of canvas being used inside WebViews. Over 742 apps use WebGLRenderingContext.getExtension, an API that provides access to the optional WEBGL_debug_renderer_info property that provides the script with a description of the graphics capabilities of the device through the WebGLDebugRendererInfo.UNMASKED_VENDOR_WEBGL and WebGLDebugRendererInfo.UNMASKED_RENDERER_WEBGL property. Similarly, 284 apps follow the typical process of converting canvas elements into data URLs using the HTMLCanvasElement.toDataURL function [40].

Surprisingly, we find that sensor APIs, like those described by Nguyen et al. [26], are not being used in the wild to actually collect sensor data. While that study provides a valuable analysis and highlights a potential threat, our investigation reveals a different story. We observed 128 apps invoking Window.Sensor; however, we manually analyzed a random subset of 10 apps and found that they check for the existence of the API (possibly to date the browser version for the fingerprint), but do not actually access sensor data.

**Summary.** Overall, our results show that third-party SDKs leverage dynamic behavior to inject context-restricted data into WebViews, highlighting the significant privacy invasion currently affecting users. Strikingly, over half of these injections involve sensitive, context-restricted data, and over 90% of those apps exfiltrated this data to third-party servers. Just 25 SDKs are responsible for this privacy diffusion, collectively transmitting data to 152 external parties, including several previously undetected trackers. Furthermore, the apps that perform this information leakage exhibit significantly higher usage of web fingerprinting APIs.

## VII. Related Work

**Android privacy.** Several papers have attempted to analyze Android apps, trying to quantify the security and privacy ramifications of apps collecting user's personally identifiable information. One of the first frameworks to do this was the TaintDroid dynamic analysis framework proposed by Enck et al. [41]. Other tools that have been used to perform similar analyses include FlowDroid [42], AmanDroid [43], DroidSafe [44], ReCon [45], AGRIGENTO [46] and antiTrackDroid [47] to name a few. However, these analyses have largely concentrated on examining the Java code within apps or the associated network leaks, overlooking the behavior and interactions of JavaScript code executed within embedded WebViews.

**Dynamic JavaScript analysis.** In contrast to Android, JavaScript analysis for standard web browsers is supported by a comprehensive collection of tooling. One of the earliest dynamic analysis tools, Fourth-Party, was developed by Mayer et al. [48], using browser instrumentation to detect privacy-violating behaviors. Acar et al. [49] followed with FPDetective, focusing on JavaScript fingerprinting. Another widely adopted approach is exemplified by OpenWPM, which injects JavaScript via browser extensions to study specific behaviors at scale [50]. However, OpenWPM is built on Firefox, making it incompatible with the Chromium-based WebViews commonly used on Android. To address this gap, Jueckstock et al. [5] introduced VisibleV8, a set of Chromium patches capable of recording all web API calls. Given its comprehensive coverage and ongoing maintenance aligned with the latest Chromium releases, we modified and integrated VisibleV8 into our toolchain for analyzing JavaScript behavior within WebViews.

**WebView privacy.** Researchers have long recognized that third-party advertising libraries often leak personally identifiable information (PII). These libraries were typically bundled into apps as opaque binary SDKs, which then loaded third-party content remotely. Early research in this area focused on mitigating these risks by sandboxing the libraries and limiting the permissions they could access within the host app [51], [52], [53], [54], [55], [56].

Since 2014, however, third-party advertising libraries have increasingly been moving towards using WebViews to display rich ads. This model, while more secure than previous iterations, has led to privacy and security issues of its own. In 2014, Son et al. [9] was one of the first to propose an attack through which a malicious ad could exfiltrate the location of its users. Since then, WebViews have become more popular, and have been used to create entire apps, called hybrid apps. This has led to a proliferation of research on the security and privacy issues associated with WebViews, including attacks that read and write files, leak sensitive data [57], [11], [58], [59], perform click-fraud [60] and in some cases, confuse users by masquerading as different apps and websites [61], [62]. However, little research has explored the use of these techniques in the wild.

Studies have also shown that third-party ad libraries collect extensive user data by leveraging the requested permissions to mine non-resettable identifiers such as the IMEI and IMSI, along with sensitive information like contact details. Historically, tracking was often achieved by over-requesting permissions [52], [54]. Recently, however, Google has imposed stricter controls, making access to unique identifiers more difficult. Despite these efforts, third-party libraries continue to circumvent these restrictions by exploiting side channels [63], leveraging persistent storage [64], and misreporting data practices in the mandated Data Safety section [65].

**Cross-boundary analysis.** The research community has made notable strides in analyzing JavaScript behavior within WebViews, though key limitations remain. In 2016, Lee et al. [3] introduced HybridDroid, a static analysis tool targeting JavaScript in WebViews. However, it lacked support for dynamic JavaScript behaviors, such as runtime script injection, and did not consider server-side dynamics highlighted by Lekies et al. [66]. In 2019, Bai et al. [2] proposed BridgeTaint to dynamically analyze Java-JavaScript interactions, but their work was limited to Cordova apps and a narrow set of JavaScript APIs. Tiwari et al. [67] later used Frida to instrument Java APIs and study WebView fingerprintability, but they did not analyze JavaScript code directly. In 2023, Pradeep et al. [68] performed an analysis of Android browsers, including WebView-based browsers, and found privacy issues surrounding how Java code communicates with JavaScript. However, their dynamic analysis pipeline relied on pre-caching dynamically generated content, which allowed them to observe dynamic Java-level behavior, but significantly limited their ability to capture dynamic behavior at the web layer. Krause [69] showed the potential to inject JavaScript in Android and iOS WebViews by using an instrumented website, but had limited JavaScript detection capabilities. Zhang et. al [70] conducted a large-scale empirical study on cross-principal manipulation of Web resources in Android using XPMChecker, and found cases of apps stealing/abusing cookies and collecting user credentials. Most recently, Kuchhal et al. [71] conducted a large-scale static analysis of JavaScript injections in WebViews, yet their method could not determine which injections were actually executed or what data flowed into the WebView from Java.

**Comparison to prior work.** To the best of our knowledge, the works by Kuchhal et al. [71], Tiwari et al. [67], [16], Bai et al. [2], Rizzo et al. [11], Son et al. [9], and Lee et al. [3] represent the most closely related efforts to ours. We consider these works to be the current state of the art, and provide a qualitative comparison in Table VIII in Appendix B. Furthermore, while different from our methodology, Kuchhal et al. [71] performed a semi-manual dynamic analysis on a subset of 10 apps. We present a quantitative and qualitative comparison for the same set of apps in Appendix B. Due to the differences in the proposed methodologies, their study is complimentary to ours, and together they serve as a useful resource for the research community.

Prior work on analyzing hybrid mobile applications has primarily focused on studying the interaction between Java and JavaScript through either static or dynamic techniques. Tools such as LuDroid [16], BabelView [11], and Hybridroid [3] rely predominantly on static analysis to reason about possible communication pathways between Java and JavaScript. While these methods model interface exposure and simulate attacker behavior, they often lack visibility into actual JavaScript execution or runtime behavior, limiting their ability to detect dynamic data flows or confirm whether the modeled paths are ever realized during execution.

Other works, such as Son et al. [9], Tiwari et al. [67], Pradeep et al. [68], and Kuchhal et al. [71], incorporate dynamic analysis to varying extents. However, these often rely on replacing the loaded webpages with researcher-controlled dummy pages which limits full coverage of Java-to-JavaScript and JavaScript-to-exfiltration flows. Notably, BridgeTaint by Bai et al. [2] achieves tracking of cross-context information flows. However, their approach traces only a limited set of whitelisted JavaScript APIs and is tightly coupled with Apache Cordova-based applications, making it less suitable for broader WebView use cases like mobile advertising. Additionally, while Pradeep et al. [68] conducted a detailed analysis of WebView-based browsers and identified key privacy concerns, their dynamic analysis used a curated set of webpages, which aligns with their focus on Java-level behaviors. In contrast, our approach emphasizes dynamic web-level activity, requiring full interaction with live, often unpredictable web content.

Our work differs in that it leverages an instrumented version of the V8 JavaScript engine (VisibleV8), enabling complete visibility into JavaScript execution at runtime. This allows us to precisely capture Java-to-JavaScript interactions, detect exfiltration behaviors, and trace how SDK-provided data propagates across contexts. Unlike prior approaches, we are not constrained by static assumptions, limitations on the variety of web content or platform-specific instrumentation, as our system dynamically observes cross-context flows in real-world applications across a wide range of frameworks, including non-Cordova apps. This comprehensive dynamic coverage allows for more accurate identification of privacy-relevant behaviors in WebView-based applications. Indicatively, we find that 59% of apps that loaded WebViews in our dataset, injected and exfiltrated context-restricted data, substantially more than prior studies. For instance, LuDroid [16] reported such flows in 26.2% of apps (1,330 out of 5,083), and BridgeTaint [2] reported only 4.5%. This sharp contrast highlights the broader visibility and effectiveness of our novel dynamic analysis approach in uncovering privacy risks at scale.

## VIII. Discussion, Limitations, and Mitigations

**Impact.** Our research uncovers sensitive information diffusion across multiple actors, and exposes previously-unknown tracking endpoints. Google's ongoing initiative to enhance transparency regarding the collection of users' personal information establishes rigorous and elevated standards (e.g., the Data Safety section), and highlights the importance of privacy-preserving technologies. Additionally, the significance of the privacy-invasive data collection we observe (e.g., of the Advertising ID) is reflected in cases such as [72], where privacy advocacy groups have filed official complaints to European Data Protection agencies over the collection of the Advertising ID. Therefore, it is evident that our framework exposes important privacy-invasive behaviors that violate existing consumer-protection legislation, and can help identify and stop privacy violations in the mobile-tracking ecosystem by shedding light on previously unexplored techniques.

**Anti-tracking tools.** Our study uncovered a number of previously unknown domains involved in cross-context tracking and data exfiltration. We responsibly disclosed them to the Disconnect.me, DuckDuckGo, and EasyList maintainers to aid in the broader ecosystem protection.

**App Store vetting safeguards.** Looking forward, we believe there is an opportunity for platform-level mitigations. One avenue is for Google, as stewards of the Android platform, to leverage their existing infrastructure, such as the Play Protect network [73], WebView-based Chrome Finch instrumentation [74], or academic tools like our framework, to dynamically analyze apps for new exfiltration behaviors, as part of their app vetting process. While dynamic analysis does not offer complete coverage, especially given its input-dependent nature, it can still act as an early-warning system for emerging information leakage techniques like those identified by our paper.

**Privacy-preserving advertising.** Google's ongoing Privacy Sandbox initiative aims to provide privacy-preserving alternatives to existing ad-related mechanisms, through a series of new APIs. While this is an ongoing effort, it would be interesting to see how it is adopted over time, and whether it provides the requisite functionality that trackers seek, such that it would involve ad targeting, without necessitating access to sensitive user data. Moreover, there have been announcements for Android implementations, such as the Topics API for Android [75]; nonetheless, an interesting endeavor would be to retrofit it so as to make it callable from the context of a WebView, thus eliminating the incentive to extract sensitive app-side data through covert channels.

**Limitations and future work.** To facilitate reproducibility, we will open-source our analysis framework, instrumentation patches, and dataset of Frida execution traces and VisibleV8 logs. As of the v139 release [76], we have upstreamed our changes to VisibleV8 with the aim of lowering the barrier for other researchers interested in studying cross-context data flows and exfiltration techniques in mobile ecosystems. While our findings shed light on a range of covert tracking behaviors, they also surface several open questions that merit further exploration. For instance, additional work is needed to understand the full extent of obfuscated identifiers exfiltrated from WebViews, as well as their flow inside JavaScript contexts. Techniques like taint tracking could help map these flows more precisely. Moreover, regex-based heuristics including transformations (e.g., Base64, SHA-256) are commonly used for detecting network leaks [77], [78], [79], but can miss instances of data exfiltration. Differential analysis [46] provides an alternative approach, but is unreliable due to the dynamic nature of web content, making regex-based heuristics the best way of identifying context-restricted information. Similarly, obfuscated or unknown third-party library package names will be missed when attributing the origin of injections. These limitations can result in false negatives, therefore our results present a lower bound of the abuse occurring in the WebView ecosystem. Furthermore, even though we use a sophisticated framework to dynamically traverse Android apps and modify it according to our needs, Android UI element coverage remains challenging. We see our work as a stepping stone, and believe that enabling the community with tools will help identify and stop privacy violations in the mobile-tracking ecosystem.

## IX. Conclusion

Smartphones have become an integral and inseparable component of most facets of modern, everyday life, and are almost always within arm's reach. Consequently, these devices not only mediate many of our most sensitive online activities and communications, but also contain massive amounts of personal data. As privacy remains both a need and a necessity, it is crucial that we continue to investigate new avenues through which privacy invasion can manifest. In this paper, we conducted a novel investigation of how tracking and PII leakage are made possible by code injection techniques that enable Java-to-JavaScript bridging, thereby connecting two separate execution contexts. By developing WebViewTracer, we are able to find that such techniques essentially enable complex invasive behaviors that bypass existing isolation and access control mechanisms. In fact, our large-scale study demonstrated not only that such behaviors are prevalent among the major players in the online advertising ecosystem, but also uncovered colluding behaviors across different third-party libraries that may not be readily apparent to app developers. As a first remediation step, we have reported the previously-unknown domains uncovered by our study to popular privacy and anti-tracking tools. We will open source our tool in an effort to incentivize additional scrutiny from researchers and enable the development of additional safeguards by platform stakeholders.

## Ethical consideration

All crawling activities were conducted at a US academic institution. To isolate experimental traffic, we routed all data through a Tailscale-managed [80] tunnel using a dedicated IP space reserved for measurement studies. At no point was any personally identifiable information (PII) collected or processed. The devices used in the experiment were physically wiped before use, and configured with fresh, anonymized dummy accounts. These phones contained no user data, were not linked to real individuals, and lacked SIM cards or any telephony capability, ensuring no accidental exposure to private or personal information.

## Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We would also like to acknowledge Alex Nahapetyan, Mirko Nikic, Nazia Afreen and Md Atiqur Rahman for their contributions towards reviewing system logs. Finally, we would like to thank Konstantinos Spyridakis and Ioannis Arkalakis for their assistance. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363, CNS-2138138 and CNS-2047260), Horizon Europe (GA No 101168465). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

## References

[1] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Proceedings of the Mobile Security Technologies Workshop (MoST)*, 2015.

[2] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "Bridgetaint: A Bi-Directional Dynamic Taint Tracking Method for Javascript Bridges in Android Hybrid Applications," *IEEE Transactions on Information Forensics and Security*, 2019.

[3] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," in *31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[4] A. Tiwari, J. Prakash, A. Rahimov, and C. Hammer, "Understanding the Impact of Fingerprinting in Android Hybrid Apps," in *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*.

[5] J. Jueckstock and A. Kapravelos, "VisibleV8: In-browser Monitoring of JavaScript in the Wild," in *Proceedings of the Internet Measurement Conference*, 2019.

[6] E. Rosado., "PII CODEX - PII Detection, Categorization, and Severity Assessment," https://github.com/EdyVision/pii-codex, 2022.

[7] K. Heid and J. Heider, "Haven't we met before?-Detecting Device Fingerprinting Activity on Android Apps," in *Proceedings of the 2024 European Interdisciplinary Cybersecurity Conference*.

[8] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[9] S. Son, D. Kim, and V. Shmatikov, "What Mobile Ads Know About Mobile Users," in *Proceedings 2016 Network and Distributed System Security Symposium*.

[10] J. Kim, J. Park, and S. Son, "The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud." in *NDSS*, 2021.

[11] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews," in *Research in Attacks, Intrusions, and Defenses*, 2018.

[12] P. Papadopoulos, N. Kourtellis, and E. Markatos, "Cookie Synchronization: Everything You Always Wanted to Know But Were Afraid to Ask," in *The World Wide Web Conference*, 2019.

[13] M. Koop, E. Tews, and S. Katzenbeisser, "In-Depth Evaluation of Redirect Tracking and Link Usage," 2020.

[14] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "REAPER: Real-time App Analysis for Augmenting the Android Permission System," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, 2019.

[15] AccessibilityService - API reference. [Online]. Available: https://developer.android.com/reference/android/accessibilityservice/AccessibilityService

[16] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, "LUDroid: A Large Scale Analysis of Android – Web Hybridization," in *19th International Working Conference on Source Code Analysis and Manipulation*, 2019.

[17] T. Frida. Frida docs. [Online]. Available: https://frida.re/docs/home/

[18] J. Jueckstock., "VisibleV8 - A instrumented variant of the V8 JavaScript Engine," https://github.com/wspr-ncsu/visiblev8, 2022.

[19] Chromium, “WebView docs - WebView Build Instructions,” https://chromium.googlesource.com/chromium/src/+/lkgr/android_webview/docs/build-instructions.md#Installing-WebView-and-switching-provider, 2022.

[20] ECMAScript® 2025 language specification. [Online]. Available: https://tc39.es/ecma262/

[21] WebIDL README.md - Chromium Code Search. [Online]. Available: https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/bindings/scripts/web_idl/README.md

[22] Web IDL interfaces. [Online]. Available: https://www.chromium.org/developers/web-idl-interfaces/

[23] Chromium, “Incremental Install README.md - Chromium Code Search,” https://source.chromium.org/chromium/chromium/src/+/main:build/android/incremental_install/README.md.

[24] Chromium., “Android-WebView renderer,” https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/renderer/README.md.

[25] J. Su and A. Kapravelos, “Automatic Discovery of Emerging Browser Fingerprinting Techniques,” in *Proceedings of the ACM Web Conference 2023*.

[26] T. T. Nguyen and B. Stock, “Open Access Alert: Studying the Privacy Risks in Android WebView’s Web Permission Enforcement,” *Asia CCS 2025*.

[27] DuckDuckGo Tracker Radar. [Online]. Available: https://github.com/duckduckgo/tracker-radar

[28] L. Li, T. F. Bissyande, J. Klein, and Y. Le Traon, “ An Investigation into the Use of Common Libraries in Android Apps ,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[29] M. Backes, S. Bugiel, and E. Derr, “Reliable Third-Party Library Detection in Android and its Security Applications,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[30] J. Samhi, M. Alecci, T. F. Bissyandé, and J. Klein, “A Dataset of Android Libraries,” *arXiv preprint arXiv:2307.12609*, 2023.

[31] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “AndroZoo: Collecting millions of Android apps for the research community,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016.

[32] P. Ahlbrecht. Racoon APK downloader for PC. [Online]. Available: https://raccoon.onyxbits.de/

[33] J. Wu, “Magisk.” [Online]. Available: https://github.com/topjohnwu/Magisk

[34] J. Yoon, R. Feldt, and S. Yoo, “Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing,” *arXiv preprint arXiv:2311.08649*, 2023.

[35] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[36] T. Su, J. Wang, and Z. Su, “Benchmarking Automated GUI Testing for Android against Real-World Bugs,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[37] Ł. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, “The leaking battery: A privacy analysis of the HTML5 Battery Status API,” in *Data Privacy Management, and Security Assurance: 10th International Workshop, DPM 2015, and 4th International Workshop, QASA 2015*, 2016.

[38] S. Boussaha, L. Hock, M. Bermejo, R. C. Rumin, A. C. Rumin, D. Klein, M. Johns, L. Compagna, D. Antonioli, and T. Barber, “FP-tracer: Fine-grained Browser Fingerprinting Detection via Taint-tracking and Entropy-based Thresholds,” *Proceedings on Privacy Enhancing Technologies*, 2024.

[39] WebSettings.setUserAgentString | API reference. [Online]. Available: https://developer.android.com/reference/android/webkit/WebSettings

[40] K. Mowery and H. Shacham, “Pixel perfect: Fingerprinting canvas in HTML5,” *Proceedings of W2SP*, 2012.

[41] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones,” *Commun. ACM*, 2014.

[42] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[43] F. Wei, S. Roy, X. Ou, and Robby, “Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps,” *ACM Transactions on Privacy and Security*, 2018.

[44] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, “Information-Flow Analysis of Android Applications in DroidSafe,” in *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society.

[45] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, “ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications and Services*, 2016.

[46] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, “Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis.” in *NDSS*, 2017.

[47] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, “The Long-Standing Privacy Debate: Mobile Websites vs Mobile Apps,” in *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[48] J. R. Mayer and J. C. Mitchell, “Third-Party Web Tracking: Policy and Technology,” in *2012 IEEE Symposium on Security and Privacy*.

[49] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, “FPDetective: dusting the web for fingerprinters,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. Association for Computing Machinery.

[50] S. Englehardt and A. Narayanan, “Online Tracking: A 1-million-site Measurement and Analysis,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[51] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, “Unsafe exposure analysis of mobile in-app advertisements,” in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[52] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, “AdDroid: privilege separation for applications and advertisers in Android,” in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[53] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, “Don’t kill my ads!: balancing privacy in an ad-supported mobile application market,” in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications - HotMobile ’12*.

[54] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, “Investigating user privacy in Android ad libraries,” in *Workshop on Mobile Security Technologies*, 2012.

[55] J. Huang, O. Schranz, S. Bugiel, and M. Backes, “The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[56] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, “FLEXDROID: Enforcing In-App Privilege Separation in Android.” in *NDSS*, 2016.

[57] P. Beer, L. Veronese, M. Squarcina, and M. Lindorfer, “The Bridge between Web Applications and Mobile Platforms is Still Broken,” in *Workshop of Designing Security for the Web*, 2022.

[58] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, “A Tale of Two Cities: How WebView Induces Bugs to Android Applications,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering*.

[59] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, 2018.

[60] G. Chen, W. Meng, and J. Copeland, “Revisiting Mobile Advertising Threats with MAdLife,” in *The World Wide Web Conference*, 2019.

[61] Z. Zhang, Z. Zhang, K. Lian, G. Yang, L. Zhang, Y. Zhang, and M. Yang, “TrustedDomain Compromise Attack in App-in-app Ecosystems,” in *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps*, 2023.

[62] G. Yang, J. Huang, and G. Gu, “Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities,” in *28th USENIX Security Symposium*, 2019.

[63] J. Reardon, A. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, “50 Ways to Leak Your Data: An Exploration of Apps’ Circumvention of the Android Permissions System,” in *28th USENIX Security Symposium*, 2019.

[64] Z. Dong, T. Liu, J. Deng, L. Li, M. Yang, M. Wang, G. Xu, and G. Xu, “Exploring Covert Third-party Identifiers through External Storage in the Android New Era,” in *33rd USENIX Security Symposium*, 2024.

[65] I. Arkalakis, M. Diamantaris, S. Moustakas, S. Ioannidis, J. Polakis, and P. Ilia, “Abandon All Hope Ye Who Enter Here: A Dynamic, Longitudinal Investigation of Android’s Data Safety Section,” in *33rd USENIX Security Symposium*, 2024.

[66] S. Lekies, B. Stock, M. Wentzel, and M. Johns, “The Unexpected Dangers of Dynamic JavaScript,” in *24th USENIX Security Symposium*, 2015.

[67] A. Tiwari, J. Prakash, A. Rahimov, and C. Hammer, “Our fingerprints don’t fade from the Apps we touch: Fingerprinting the Android WebView,” *arXiv preprint arXiv:2208.01968*, 2022.

[68] A. Pradeep, A. Feal, J. Gamba, A. Rao, M. Lindorfer, N. Vallina-Rodriguez, and D. Choffnes, “Not Your Average App: A Large-scale Privacy Analysis of Android Browsers,” in *Proceedings of the 23rd Privacy Enhancing Technologies Symposium*, 2023.

[69] F. Krause, “iOS Privacy: Announcing InAppBrowser.com - see what JavaScript commands get injected through an in-app browser,” https://tinyurl.com/InAppBrowser, 2022.

[70] 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 *27th USENIX Security Symposium*, 2018.

[71] D. Kuchhal, K. Ramakrishnan, and F. Li, “Whatcha Lookin’ At: Investigating Third-Party Web Content in Popular Android Apps,” in *Proceedings of the 2024 ACM on Internet Measurement Conference*.

[72] TechCrunch, “Google’s Android ad ID targeted in strategic GDPR tracking complaint,” https://techcrunch.com/2020/05/13/googles-android-ad-id-targeted-in-strategic-gdpr-tracking-complaint/#:~:text=noyb%20has%20now%20filed%20a,ID%20without%20legally%20valid%20consent.

[73] Use Google Play Protect to help keep your apps safe & your data private - Google Play Help. [Online]. Available: https://support.google.com/googleplay/answer/2812853?hl=en

[74] What is a Chrome Finch experiment? | Web Platform. [Online]. Available: https://developer.chrome.com/docs/web-platform/chrome-finch

[75] Topics API for Mobile overview | Privacy Sandbox. [Online]. Available: https://privacysandbox.google.com/private-advertising/topics/android

[76] J. Jueckstock, “wspr-ncsu/visiblev8, release df6ccfd-139.0.7258.127,” Aug. 2025, original-date: 2019-10-21T17:51:07Z. [Online]. Available: https://github.com/wspr-ncsu/visiblev8/releases/tag/visiblev8_df6ccfd-139.0.7258.127

[77] M. Diamantaris, S. Moustakas, L. Sun, S. Ioannidis, and J. Polakis, “This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration,” in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021.

[78] T. T. Nguyen, M. Backes, N. Marnau, and B. Stock, “Share First, Ask Later (or Never?) Studying Violations of GDPR’s Explicit Consent in Android Apps,” in *30th USENIX Security Symposium*, 2021.

[79] Q. Chen, P. Ilia, M. Polychronakis, and A. Kapravelos, “Cookie Swap Party: Abusing First-Party Cookies for Web Tracking,” in *Proceedings of the Web Conference 2021*.

[80] What is Tailscale? · Tailscale Docs. [Online]. Available: https://tailscale.com/kb/1151/what-is-tailscale

[81] Kik, “Temporary removal of Kik from Google Play Store,” https://help.kik.com/hc/en-us/articles/38425110509467-Google-Play-Store.

[82] ——, “Web archive: "Temporary removal of Kik from Google Play Store",” https://web.archive.org/web/20250704153850/https://help.kik.com/hc/en-us/articles/38425110509467-Google-Play-Store.

[83] R. Duan, W. Wang, and W. Lee, “Cloaker Catcher: A Client-based Cloaking Detection System,” *arXiv preprint arXiv:1710.01387*, 2017.

[84] S. Datta, “wspr-ncsu/WebViewTracer: A browser instrumentation and crawling framework for Android apps with WebViews,” 2025. [Online]. Available: https://github.com/wspr-ncsu/WebViewTracer

[85] ——, “Artifact for WebView Tracer,” 2025. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.16687648

[86] “Install Android Studio.” [Online]. Available: https://developer.android.com/studio/install

## Appendix A
## Additional Technical Details.

Here we present additional technical details concerning the adaptation of VisibleV8 for Android WebViews. The VisibleV8 WebView provider used in our experiments is based on VisibleV8 v131.0.6778.81, which was the latest version at the time of implementation. We have since verified that our modifications remain compatible with VisibleV8’s current release v138.

The modified WebView VisibleV8 patchset contains three major changes on top of the existing VisibleV8 patches. In the first change we modify the Blink-Java layer to log calls from JavaScript code, this is done by capturing a JavaScript interface call once when it leaves the V8-Blink layer and once when it enters the V8-Blink boundary from the IPC layer separating the Java code from the C++ implementation. We modify the V8 interface to expose VisibleV8 logging functions to the Blink codebase which is used to log the return value and the arguments sent to the call into the Java code.

In the original VisibleV8 patchset calls to the `visv8_should_log_object()` function that checked if a script used a web API that was either a builtin API or a property of that global object occurred after the VisibleV8 context had been initialized. This means that every function call or assignment to a global object triggered initialization of the VisibleV8 context. This process involves expensive operations, such as retrieving the currently executing script and its script ID, followed by lookups in VisibleV8’s global table. This additional computation overhead, while trivial in the context of desktop browsers caused the WebViews which were loaded on devices with much more restricted resources to freeze and crash. As a result, in our modified patchsets, we changed the order of operations for VisibleV8 to first check if a log statement was to be emitted before initializing the VisibleV8 context.

In addition to the above, the following flags are enabled to allow the WebViews to log data directly to the sdcard. The `disable-multiprocess` flag [24] locks the renderer process to a single renderer. It doesn’t remove core functionality or alter the behavior of Java-JavaScript communication/network APIs. The `incremental_install` flag [23] is a feature designed to speed up the Chromium build/installation. The relaxed sandboxing mechanism disables isolated processes for faster dex loading, and allows us to extract VisibleV8’s logs to the sdcard.

## Appendix B
## Quantitative & Qualitative comparison with prior work.

Table VIII provides a comparison to prior studies with WebView analysis. More directly related to our work is the work by Kuchhal et al. [71], who conducted a large-scale static analysis of JavaScript injections in WebViews and followed it with a semi-manual dynamic analysis on a subset of 10 apps. While a direct comparison isn’t possible due to differences in methodology, app versions, and timing of experiments, as well as the dynamic, ever-changing nature of the web, a quantitative and qualitative comparison still offers a valuable, complementary perspective on the mobile web ecosystem. We use WebViewTracer to identify cross-context Java-to-JavaScript interactions and manually exercise these

TABLE VII: Manual evaluation of regex-based heuristics for privacy-invasive injections on a set of 60 apps.

| Reviewer | TP | FP | FN | TN | Precision (%) | Recall (%) |
|---|---|---|---|---|---|---|
| A | 1037 | 10 | 118 | 4193 | 99.04 | 89.78 |
| B | 909 | 4 | 1 | 3771 | 99.56 | 99.89 |
| C | 879 | 57 | 28 | 3638 | 93.91 | 96.91 |
| D | 1079 | 1 | 34 | 4839 | 99.90 | 96.94 |
| E | 872 | 11 | 13 | 3392 | 98.75 | 98.53 |

apps for 10 minutes. Subsequently, we manually analyzed the injected scripts and the Java-to-JavaScript bridges. We were not able to analyze Kik since it is temporarily removed from the Google Play Store [81], [82]. After installing the latest version of Kik from AndroZoo, the application indicates that it is outdated and requests an update that cannot be completed. Table IX shows the intent of the injected scripts and Java-to-JavaScript bridges for the remaining apps. In the majority of analyzed applications, we confirmed the presence of injections reported by [71], in addition to discovering several new ones. Specifically we found seven new types of injections for Facebook, Instagram, Snapchat, Twitter, LinkedIn and Moj and in total we have identified nine new Java-to-JavaScript bridges. Noteworthy examples are Facebook, Instagram and Moj. We observed that Facebook and Instagram extract all image URLs from WebViews and also log the number of clicks and the scroll depth. Furthermore, we identified that Unity is injecting and exfiltrating context-restricted information (e.g., Advertising ID) from the Moj application to domain `https://impression.link`. According to the DuckDuckGo Tracker Radar dataset this domain is assigned a fingerprinting score of zero. Finally, we were unable to identify three injections and four Java-JavaScript bridges reported in prior work, which we attribute either to the removal of this functionality or to the possibility that our manual analysis did not reach the relevant parts of the application.

## Appendix C
## Regex-based heuristics evaluation.

We assess the prevalence of false positives and false negatives in our regex-based heuristics by randomly selecting 60 applications and manually analyzing the Frida and VisibleV8 logs for privacy-invasive behavior. This process was performed by five security researchers, each assigned to review the logs of 12 applications produced by WebViewTracer. Table VII presents false positives and negatives produced by our system. We observed that, in most cases, WebViewTracer accurately identified injections involving context-restricted data, with only a small number of false positives. Furthermore, through manual inspection of the logs, we identified instances where context-restricted data were injected using gzip compression and Base64 encoding, or subjected to multiple transformations (e.g., URL encoding followed by Base64), which resulted in false negatives by our system.

## Appendix D
## WebView coverage evaluation.

To evaluate the number of WebViews identified by our framework through automated interaction, we conducted an experiment on 50 applications and compared the results with those obtained via manual interaction. In both cases we interacted with each application for five minutes and set the `highlight-all-webviews` flag from the WebView DevTools in order to highlight and annotate all WebViews displayed on the screen. Because the content of the WebViews varied between sessions (manual and automated), we counted WebViews for each app based on their on-screen position and the associated application activity. Among the apps executed the most common use of WebViews was for interstitials and banner ads many of which showed up randomly or when a specific UI element or action was triggered. We observed instances where WebViews were absent in a specific activity during the manual session but appeared during the automated session, and vice versa. During the manual analysis of the apps we identified 141 WebViews, while UIHarvester identified 70 WebViews (49.64% WebView coverage). Furthermore, UIHarvester is unable to navigate games correctly which applied to 11 apps in our dataset. Upon filtering out game apps, UIHarvester achieved a coverage of 52.13%. We note that due to the non-deterministic behavior of applications, comparing the number of WebViews identified in one app session with those from another, regardless of whether the analysis is manual or automated, is not directly applicable. The results of this manual comparison indicate that, when analyzing a large set of applications, as in our study, automated interaction is sufficient to uncover privacy-invasive activities in WebViews. Consequently, the findings of our study should be interpreted as a conservative lower bound.

## Appendix E
## Examples of Frida and VV8 logs.

Figure 6 and Figure 7 present examples of Frida and VV8 logs respectively. Figure 6 shows an example of a Frida log in which an application loads a Google Ads domain and injects an empty function into the WebView to verify whether the JavaScript has successfully loaded. Figure 7 shows a JavaScript snippet (top) and the associated VisibleV8 log for the network request (bottom) of an exfiltration call through Unity's Java code.

## Appendix F
## Artifact Appendix

*In our NDSS submission, we proposed WebViewTracer, a framework designed to dynamically analyze the execution of JavaScript code within WebViews at runtime. The following section was our submission to NDSS 2026 Artifact Evaluation Committee.*

Our artifact consists of three parts, the first is the modified VisibleV8 patchsets that were used to create the VisibleV8 WebView. We provide detailed instructions to build and install it on an Android phone alongside prebuilt versions of the apk for x86_64 and ARM64. The second is the crawler that was

TABLE VIII: Qualitative comparison to prior studies with WebView analysis.

| Papers | Analysis | | | Information flows identified | | | |
|---|---|---|---|---|---|---|---|
| | Type | Analyzed JavaScript | Analyzed Java | Java to Java | Java to JS | JS to Java | JS to exfil |
| **WebViewTracer** | Dynamic | ● | ● | ○ | ● | ◐ | ● |
| Kuchhal et al. [71] | Static, Dynamic | ◐ | ● | ○ | ● | ○ | ○ |
| Tiwari et al. [67] | Static, Dynamic | ○ | ● | ○ | ● | ◐ | ○ |
| Tiwari et al. (LuDroid) [16] | Static | ◐ | ● | ○ | ● | ○ | ○ |
| Bai et al. [2] | Dynamic | ◐ | ● | ●* | ●* | ●* | ○ |
| Rizzo et al. [11] | Static | ○ | ● | ○ | ● | ◐ | ○ |
| Son et al. [9] | Dynamic | ○ | ● | ○ | ● | ○ | ○ |
| Lee et al. [3] | Static | ◐ | ● | ○ | ● | ◐ | ○ |

● full coverage, ○ no coverage, ◐ partial coverage, ●* full coverage, but only in Apache Cordova

TABLE IX: Quantitative and qualitative comparison of our system WebViewTracer (👁) with Kuchhal et al. [71].

| **App Name** | **Injection Intent** | 👁 | [71] | **Java-JavaScript Bridge Intent (bridge name)** | 👁 | [71] |
|---|---|---|---|---|---|---|
| Facebook<br>Instagram | WebView/Webpage language preference | ✓ | ✖ | Scroll logging (enableVerticalScrollDepthLogging) | ✓ | ✖ |
| | Extract images and metadata | ✓ | ✖ | Meta Checkout | ✖ | ✓ |
| | Logging numbers of clicks and scroll depth | ✓ | ✖ | Facebook Pay | ✖ | ✓ |
| | Returns DOM Tag Counts | ✖ | ✓ | Performance metrics (navigationPerformanceLoggerJavascriptInterface) | ✓ | ✓ |
| | Logs performance metrics | ✓ | ✓ | AutoFillExtensions | ✓ | ✓ |
| | Returns simHash for page to detect cloaking [83] | ✓ | ✓ | | | |
| | Insert FB Autofill SDK JS script | ✓ | ✓ | | | |
| Snapchat | Logs performance metrics | ✓ | ✖ | Google Ads (googleAdsJsInterface) | ✓ | ✖ |
| Twitter | Set Grok model id/name | ✓ | ✖ | Google Ads (googleAdsJsInterface) | ✓ | ✖ |
| LinkedIn | DoubleVerify measurements & analytics | ✓ | ✖ | Event signaling (androidLIWebsiteSignalMessageHandle, Android.sendWebMesage, challengeCompleted) | ✓ | ✖ |
| | Calls to Cedexis traffic management API | ✖ | ✓ | | | |
| Pinterest | No injection | - | - | (Obfuscated) | ✖ | ✓ |
| Moj | Context-restricted information injections by Unity | ✓ | ✖ | Unity (window.nativeBridge.handleInvocation, window.Android.messageHandler) | ✓ | ✖ |
| | Insert and manage a video Ad via Google Ads SDK | ✓ | ✓ | Google Ads (googleAdsJsInterface) | ✓ | ✓ |
| Chingari | Insert and manage a video Ad via Google Ads SDK | ✖ | ✓ | Chingari events (trackAnalyticsEvent) | ✓ | ✖ |
| | | | | Google Ads | ✖ | ✓ |
| Reddit | No injection | - | - | recaptcha.net obfuscated data (RN.zzlce) | ✓ | ✖ |

```
"data": {
  "action": "INJECT-LOAD",
  "hashcode": 74450829,
  "func": "loadUrl",
  "params": ["\"https://googleads.g.doubleclick.net...\""]
},
"stacktrace":
↪  "dalvik.system.VMStack.getThreadStackTrace..."

"data": {
  "action": "INJECT-JS",
  "hashcode": 74450829,
  "func": "evaluateJavascript",
  "params": ["\"(function(){})()\""]
},
"stacktrace":
↪  "dalvik.system.VMStack.getThreadStackTrace...."
```

Fig. 6: Frida log showing a Google Ads controlled WebView, loading a doubleclick.net page and injecting an empty self-invoking function to verify page load.

```
window.nativeBridge.handleInvocation(
    "com.unity3d.services.core.api.Request",
    "post",
    [...,  "https://auction-load.unityads.unity3d.com?...."]
)
```

```
c370:%<anonymous>:{409,Object}:
"[["com.unity3d.services.core.api.Request","post",
["3","https://auction-load.unityads.unity3d.com?"....]...]]"
```

Fig. 7: JavaScript snippet (top) and the associated VisibleV8 log for the network request (bottom) of an exfiltration call through Unity's Java code.

used to run our experiments. This uses docker to start a prebuilt emulator that already has a version of the VisibleV8Webview installed. This component allows us to exercise the apps and helps obtain the Frida logs used to perform our experiment. The third component of the artifact is our dataset. We provide a `apk_artifact.json` file that contains the hashes, version and app names of all the apps used in our experiment as well as a small zipped subset of 195 apps that can be used to run a small scale experiment.

Due to the highly dynamic nature of the data being observed and the scale (over a month of crawling), the experiment conducted in the paper is hard to reproduce. We are providing a set of 195 apps, from our dataset, for which we downloaded the x86_64 version (to work in an emulated environment) that we believe will provide a good-enough approximation of the high-level results we obtained from the crawl. To build this dataset of 195 apps, we took the list of apps that had webviews from our original dataset, picked 500 apps at random, and then re-downloaded the apps from the Google Play App Store using the credentials of the emulated phone. Out of these 500, 198 apps were successfully downloaded for the specific architecture and credential combination on July 14th 2025.

Overall, we provide the system with which we crawled over 10K apps, the modified VisibleV8 patches, and a JSON dump of all apks downloaded as part of our experiment in our Github repository [84]. We also provide the original dataset of

JavaScript execution traces and logs dumped by Frida's tracing of Java SDKs that we used to obtain the results in the paper.[1]

### A. Description & Requirements

*1) How to access:* The artifact files can be downloaded from [85]. The `WebViewTracer-main.zip` archive in Zenodo represents a Git repo that we will open-source with the paper. The `SystemWebView.apk` file is the prebuilt x86_64 version of the VisibleV8 WebView provider. The `trace-apis.patch` and `chrome-sandbox.patch` are the patches that can be used to build a version of VisibleV8 Chromium for v138. The APK dataset is at `x86_64_apps.zip`. The `avd.zip` is the archive containing a `x86_64` emulator that can be used to run the small scale experiment.

*2) Hardware dependencies:* The scaled down version of our system requires a x86_64 system with CPU virtualization support. In addition, the system running the experiment must have CPU and memory requirements to run Android Studio. [86] To replicate the larger version of the full experiment run will require at least one Android Pixel phone based on the ARM64 architecture. To run the results scripts to reproduce the experiment on the original data, a significant amount of SSD space is required.

*3) Software dependencies:* Docker, docker compose (higher than v2), Python $\geq$ 3.10 and venv need to be installed on the system and are necessary to run these experiments. Note that rootless docker alternatives like `podman` are not supported since they interfere with experiment setup. Python is required to run the orchestration scripts and docker is used to create databases, and setup a reproducible emulator (we tested emulator version 36.1.9.0, build_id 13823996). The emulator-based mode requires a Linux system with kernel-based KVM support as a precondition to the Android emulator working. For the larger experiment, an Android Pixel phone (our experimental system used a Pixel 4a) is required that would need to be rooted with some kind of rooting software like Magisk or KernelSU and a version of "systemizer" a method of manually replacing system apps with different variants should be installed. We've tested our scaled-down experimental system on Ubuntu 24.04.01, with a docker version of 28.3.3 (build 980b856) and a docker compose version of v2.39.1. Our original large-scale experiment was also run Ubuntu 24.04.1, with a docker version of 28.2.2 (build e6534b4) and docker compose version of v2.36.2. We have even tested our system on Arch Linux (running a kernel version of 6.15.9) with a docker version of 28.3.3, build 980b856 and docker compose version of 2.39.2 (note that any kind of aliasing of docker to alternatives are not supported).

*4) Benchmarks:* While there are no explicit benchmarks being performed, we do provide two datasets as part of our artifact, the first one is the dataset of apps, this is a list of all apps that were downloaded and used to perform our analysis. These can be used to perform our experiment. Another dataset contains a set of files containing the browser execution traces and Frida logs that were subsequently used to obtain our results.

We provide two scripts, `android-check.sh` and `python ./scripts/wvt-cli.py` (see Section B) to check if your system is capable of running the artifact. If the `android-check.sh` is capable of running and displaying an emulator window and the `python3 ./scripts/wvt-cli.py` script exits without a 255 exit code, you should be able to run the small-scale experiment.

### B. Artifact Installation & Configuration

To run the scaled down experiment the following steps must be followed,

- Git clone [84]
- Change directory to the webviewtracer-crawler directory
- Setup a python virtual environment by running `python3 -m venv env && source env/bin/activate`
- Install the dependencies using `pip install -r scripts/requirements.txt`
- Run `python3 ./scripts/wvt-cli.py` and make sure it does not exit with a 255 exit code and does not output any error messages.
- Run the `android-check.sh` to check if you are able to correctly run a small scale emulator on your system.
- Download the AVD emulator image avd.zip in [85] and unzip it into the `celery_workers/avd/` directory
- Run `python3 ./scripts/wvt-cli.py setup`, the CLI will ask you a few questions, you can choose the default option by pressing enter once the questions are asked.
- Navigate to the "`apps/split_1`" directory and unpack x86_64_apps.zip in [85] into the directory
- Run `python3 ./scripts/wvt-cli.py crawl`. If this command does not work, please use `docker compose -env-file .env up -build -d -V -force-recreate -remove-orphans`, the python script is a transparent wrapper to this command and it should start running the experiment.
- Navigate to `http://0.0.0.0:6901` and observe the apps being crawled
- Once all the apps are done crawling, run `ls raw_logs/ > tmp` in the webviewtracer-crawler directory.
- Run `python3 ./scripts/wvt-cli.py postprocess -f tmp -pp 'Mfeatures+androidflow+exfil+frida'`, open `0.0.0.0:5559` and wait for all the tasks to succeed or fail. The jobs will fail if no webviews are loaded at all while crawling the app and it is normal for a lot of them to fail.
- Run `python3 ./scripts/wvt-cli.py results` to view the results
- You can use `python3 ./scripts/wvt-cli.py shutdown` to shutdown the crawler

**Region specificity.** Many of the apps that we provide have differing behavior based on the region in which they are run, (for example, when the apps are run in the Europe, Google Ads will display a consent banner to align with GDPR laws), to reproduce our experiments, we do recommend using a VPN

[1] https://doi.org/10.5061/dryad.05qfttffz

to connect to a server in the US to more closely replicate the results we got in our paper. We include `openvpn` as part of our Docker container, you can drop a valid .ovpn file in the `vpn/` directory and then load the VPN by using `sudo openvpn /app/vpn/<config.ovpn>` inside the docker container.

**Debugging steps.** Depending on hardware configurations and regional differences, there might be issues with emulators crashing when opening and running specific apps, we recommend analyzing the logs in the "webviewtracer-crawler/raw_logs" directory (of the unzipped WebViewTracer-main.zip file) to understand the errors that being hit during crawling an app. Each app will have its own subdirectory which will contain the a directory called "logcat/" which contain a logfile (called "full_logcat") which contains the logcat output during the run of the app. Similarly, often the issue can be incompatibilities with Frida hooking, our "raw_logs" dumps also contain the logs of all frida calls in "frida/logfile". Finally, every logfile contains a set of images taken during the traversal of the app itself, which can verify how the crawling setup navigated the app and a set of raw VisibleV8 logs in the "Documents" directory for each app which can be inspected to understand issues with JS execution inside the WebViews. These logs later get postprocessed and are used during the analysis.

**Running the full-scale experiment.** If you would like to run the full scale experiment on newer browser versions, you can find newer version of our VisibleV8 WebView patches in the upstream VisibleV8 repo [18] starting with Chrome 139. The patch files provided in the "patches" directory can also be built from scratch by following `patches/webview_build_instructions.md` of the unzipped WebViewTracer-main.zip file).

We would recommend starting with downloading a set of apps using the methodology we outline in `dataset/Downloading_apps.md` (of the unzipped WebViewTracer-main.zip file) and then setting up a ARM64 phone with VisibleV8 WebViews, before running the same commands that were run for the small-scale experiment, except that during the setup command (step 5) the 'physical' prompt must be chosen when the script asks the question "What type of devices are you using?". Note that the devices must be plugged in and connected and authorized to connect be remotely debugged through adb by the computer running the experiment.

### C. Experiment Workflow

The experiment starts by setting up a set of dockerfiles and then subsequently starting an already prepared Pixel 6a emulator with WebView version 138 installed, installing apps one by one on the emulator and using UIHarvester to perform a depth-first-search of the UI of the apps. It does this for 5 minutes per app and retries them two times if there is unforeseen failure of the Frida-based instrumentation at any moment.

In the experiment conducted in our paper, we used 4 real Google Pixel 4a devices that had a the VisibleV8 Webview provider for version 131. We also used a timeout of 20 minutes and retried every single app 5 times across 1K apps. The amount of time the system spends crawling an app can be tweaked by editing `webviewtracer-crawler/celery_work ers/vv8_worker/uiharvester/execution_wrapper/application_runner/mode.py` line 180, the number of times an app can be retried can be changed by changing the last number at `webviewtracer-crawler/celery_workers/vv8_wo rker/entrypoint.sh`.

To use physical phones, the phones need to be connect to the computer using USB ports and the computer should have ADB installed. The python orchestration module should figure out the number of phones and assign each to it's own crawling module.

The `webviewtracer-crawler/raw_logs` contains all the logs for each app and `webviewtracer-crawler/crawl-data/` and the command `docker compose logs -f` can be used to obtain the logs for the traversals of each instance of emulator used.

### D. Major Claims

The following are the claims being reproduced by the artifact, note that the claims are quantitative rather than numerical.

- (C1): SYSTEM discovers the leakage of context-restricted data from the app through injections in WebViews and is exfiltrated to the outside world.
- (C2): SYSTEM shows the presence of phantom exfiltrations, i.e. exfiltrations of data without associated injections.

### E. Evaluation

The following experiments are being reproduced

*1) Experiment (E1):* Crawl [30 human-minutes + 16.5 compute-hours]: This experiment will run a small-scale crawl and provide numbers to prove C1 and C2.

*Preparation and Execution* Follow the steps at Section B

*Results* To fully reproduce our experiment, the user should see some apps loading WebViews and subsequently using WebViews to exfiltrate context restricted information. The results command (`python3 ./scripts/wvt-cli.py results`) should provide a table of the kinds of context restricted information being exfiltrated out. Some apps might load test ads due to non-standard nature of the emulator in which case there might be lesser injections and exfiltrations of context-restricted information. There should at least a few categories where the exfiltrations is very low compared to those reported by our paper. This is expected since data like "City" and IP information will vary from area to area. We provide an existing list of regexes at `webviewtracer-crawler/scripts/results.py` lines 8-26 which can be modified and swapped out entirely based on the environment the reviewer is using. We encourage the users to try and swap out the regex and try their own ones to test the system! The regexes we used are documented in `webviewtracer-crawler/scripts /experiment_pii_regexes.py`