Tracking Without Borders: Studying the Role of WebViews in Bridging Mobile and Web Tracking

Nipuna Weerasekara
IMDEA Networks Institute / Universidad Carlos III de Madrid

José Miguel Moreno
Universidad Carlos III de Madrid

Srdjan Matic
IMDEA Software Institute

Joel Reardon
University of Calgary / AppCensus

Juan Tapiador
Universidad Carlos III de Madrid

Narseo Vallina-Rodríguez
IMDEA Networks Institute / AppCensus

Abstract

WebViews are a core component of today's in-app browsing technologies on mobile platforms, playing a central role in rendering web content like mobile advertisements. However, their use and potential to bridge web and mobile tracking paradigms comes at a significant privacy cost for users. Although prior work has highlighted privacy risks associated with WebViews, the real-world scale and privacy impact of their misuse and abuse remain unexplored due to the hybrid nature of WebViews—combining Java, native, and dynamically-loaded JavaScript (JS) code. In this paper, we present the first large-scale empirical study of WebView abuse in Android apps. We analyze how app developers and third-party SDKs facilitate user tracking by configuring WebViews to bypass default platform privacy protections and enable invasive tracking through JavaScript code. Using a novel analysis pipeline that combines static and dynamic analysis of Java/Kotlin code and JavaScript, we reveal how numerous actors undermine users' privacy and exploit WebViews in the wild. We show that harmful JavaScript code, often distributed via unvetted Real-Time Bidding (RTB) processes, exploits WebViews to perform advanced tracking techniques such as cookie sync-ing, canvas fingerprinting, and misuse of the Java-JS interface and permission-protected JavaScript APIs to silently leak unique user identifiers and geolocation data without user awareness for cross-platform tracking.

Keywords

Android WebViews, Cross-platform Tracking, Fingerprinting, Mobile Platforms, Privacy

1 Introduction

In-app browsing refers to the process of rendering web content within a mobile application (app) user interface instead of redirecting users to an external browser. This capability is typically achieved using WebViews and Custom Tabs (CTs), but there are significant differences between these two technologies. WebViews are customizable embedded browser engines that allow apps to render external links, load dynamic web content, or integrate into apps' web-based authentication flows. Instead, CTs allow integrating a full browser on mobile apps (i.e., Chrome Custom Tabs on Android or SFSafariViewController on iOS). However, while WebViews offer greater customization and control over the web content, they lack security, performance optimizations, and shared browser features like autofill, navigation bar, and password management which are available on CTs [42].

WebViews are the most common in-app browsing technology in Android due to their versatility and customizability [58, 142]. For example, advertising SDKs leverage WebViews for rendering advertisements distributed through Real-Time Bidding (RTB) processes [96].[1] However, their customizability and architectural design pose significant privacy and security risks to users as reported by prior research [91, 103, 115, 132, 139, 144, 158]. On the one hand, the ability to run arbitrary JS programs on WebViews allows developers, third-party SDKs, and web services to track mobile app users using well-known web tracking techniques like web fingerprinting [1, 63, 94] or cookie sync-ing [18, 64]. On the other hand, WebViews enable communication channels between JS and the apps' code to exchange sensitive data between native and web contexts [139]. Finally, discrepancies between the WebView and Android permission models enable JS code in WebViews to *silently* access permission-protected data such as GPS coordinates [57]. In fact, JS code dynamically loaded in WebViews like advertisements does not necessarily abide by the checks performed by vetting processes deployed by app stores to limit abuse, like Google Play Protect, which relies primarily on static analysis.

Many WebView privacy abuses, however, remained to date as a theoretical possibility demonstrated by proofs-of-concept. No prior study has organically explored the (ab)use of WebViews by mobile apps, third-party SDKs and web services, nor gathered broad evidence of privacy abuses in the wild. Recent work by Kuchhal et al. [96] provides a large-scale static analysis of WebView and CT usage in 146.5K Android apps, using app metadata for SDK attribution. Complementing this, they use dynamic analysis to explore WebViews' behavior and security implications by injecting and testing 100 popular desktop websites into the WebViews embedded in a subset of 1K apps. However, this study does not analyze how WebViews are abused in the wild by real-world advertising and

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
Proceedings on Privacy Enhancing Technologies 2025(4), 745–762
© 2025 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2025-0155

[1] We consider CTs out of scope for this study, as they serve different purposes and operate under distinct security constraints.

tracking actors to silently track users and perform cross-platform tracking. This paper fills this fundamental research gap by conducting the first large-scale empirical study of privacy-intrusive practices in Android WebViews, analyzing how online tracking services abuse WebViews to bridge native and web tracking paradigms. Specifically, we aim to answer the following research questions:

(1) To what extent are WebViews present in mobile apps, and how do they allow other actors to distribute and execute privacy-harmful JS code on them?
(2) How do WebViews facilitate user tracking by bridging web and mobile tracking approaches?

To systematically answer these questions, we design for the first time a comprehensive methodology that leverages static and dynamic analysis methods to automatically capture the execution of both Java and JS code within WebViews across 13,045 apps and analyze their privacy consequences. To account for potential geographical differences resulting from the probabilistic nature of RTB and to maximize coverage, we use a geo-diverse farm of instrumented Android devices to run the apps from multiple world regions. This approach allows us to identify the execution of harmful JS code on WebViews across the majority of the analyzed apps (88%), including WebViews from advertising SDKs like Google Mobile Services [80], Facebook Ads [106], Applovin [15], ByteDance [25], and Unity Ads [149] (§5). Our instrumentation allows us to empirically demonstrate how WebViews facilitate privacy-intrusive practices like device fingerprinting, cookie sync-ing, stealthy access to permission-protected geolocation data, and cross-platform tracking through ID bridging between web and native contexts. Specifically:

- Nearly all (99%) of the inspected apps integrate at least one WebView. Out of these, the vast majority customize WebViews' default configuration to allow arbitrary JS execution (98%), enable DOM storage sharing (79%), or customize UA strings (88%), all of which have security and privacy consequences. Third-party SDKs are responsible for 84% of such customizations (§5.1).
- Through traffic analysis, we find that WebViews and unvetted RTB processes amplify the number of actors that can potentially track mobile app users (§5.2). We find 1,211 different Second-Level Domains (SLDs) executing JS code in WebViews from SDKs like Google Mobile Services (GMS), which load JS code from 119 SLDs, Applovin (104 SLDs), and Unity Ads (51 SLDs).
- We observe JS code setting at least one cookie or an impression pixel in 39.3% of WebViews. We determine that 49% are potential tracking cookies and 37% are associated with Advertising and Tracking Services (ATSes) that exploit the default same-site security policy disabled by SDKs. Similarly, 60.4% of the captured impression pixels are associated with known ATSes. We also find instances of CNAME tracking attributable to organizations such as Ensighten [10] or Adobe Experience Cloud [4] (§6.1).
- We find instances of JS code executed on WebViews using browser fingerprinting methods. Three of the JS APIs commonly used for fingerprinting are invoked by nearly all the hostnames that the WebViews contact. We report instances of browser and canvas fingerprinting supposedly used for anti-fraud [36, 125] but also for secondary purposes like advertising (§6.2).
- Through dynamic analysis, we observe JS code trying to access permission-restricted resources such as geolocation coordinates, microphone and camera. Most of these attempts (54%) are successful as the host app or SDK granted the associated permission to the WebViews. This occurs without user awareness (§6.3).
- We detect a wide use of data-sharing channels between JS and Java code to leak PII like the Android ID and the Android Advertising ID (AAID). These practices facilitate ID bridging between the web and native contexts for building rich cross-platform user profiles (§6.4).
- We observe 1,190 SLDs performing cookie sync-ing and ID bridging, with 44% of the hostnames associated with ATSes. We find evidence of cookie sync-ing and ID bridging practices in flows containing the AAID, Android ID, hashed email addresses (HEMs), and GPS coordinates (§6.5).

Our study highlights the lack of control over SDK WebView customizations, the JS code they run—particularly when distributed through ad networks—and the challenges in mitigating privacy abuses in in-app browsing with existing platform vetting processes (e.g., Google Play Protect) and privacy controls. As a result, WebViews facilitate silent cross-domain user tracking for post-cookie identity graph solutions.

Research Artifacts. The dataset, Proofs-of-Concept (PoC), and code artifacts from this paper are available and documented at https://github.com/WebViews-2025/Artifacts.

Responsible Disclosure. We disclosed our results to the Android Security Team through bug reports describing canvas fingerprinting, silent permission piggybacking, and WebView hijacking and customization by malicious SDKs. We were awarded a bug bounty for our finding on canvas fingerprinting in Android WebViews. The Android Security Team acknowledged the privacy risks of the other reports but has considered them infeasible to fix so they remain exploitable in Android 16. We reported our results to the European Data Protection Supervisor (EDPS) and CNIL.

2 Background

In-app browsing is a hybrid mobile technology that allows app developers to load and render web content within an app's user interface (UI). Android offers two technologies to perform in-app browsing: WebViews [50] and CTs [53]. This section describes the main usages of WebViews (§2.1), their security model (§2.2), and discusses the privacy implications of allowing developers and third-party SDKs to customize WebViews' default privacy and security properties (§2.3).

2.1 WebViews Introduction

WebViews [50] are an extension of the Android View class that acts as an embeddable—and fairly customizable, as we will see in §2.2—web engine to render web pages directly within the native app's UI, thus eliminating the need for navigating users to an external, fully-fledged browser. WebViews allow developers to dynamically display content generated on the server side without releasing a new app version like showing terms of service pages and user guides to registration forms, but mostly for rendering mobile ads [39, 84]. In fact, advertising SDKs prefer WebViews for delivering ads due to their customizability, including banner and interstitial formats,

as noted in the Google Ad Manager documentation [9]. Although Android's default web engine relies on Chromium, developers can freely invoke or embed alternative WebView implementations, such as Mozilla's GeckoView [56] or Facebook's WebView [129].

2.2 WebView Security Model

The Android WebView security model implements a sandbox to isolate web content from the native host app and limit data leakage. This separation aims to ensure that bugs or exploits within the renderer do not compromise the host app. However, as we describe in §2.3, app developers—or embedded third-party SDKs—can programmatically customize the default security properties runtime behavior of WebViews to enable arbitrary JS execution and data sharing channels between the Java (native) and JS (web) execution environments, thus breaking the sandbox boundaries.

One key security element of WebViews is their permission model for controlling how JS code accesses sensitive data and resources. This model is inherited from Chrome's, establishing a direct mapping between WebView JS permissions and runtime Android permissions (e.g., GPS sensor, camera and microphone) as shown in Table 11 (Appendix A). However, WebView's access control mechanisms are already known for presenting fundamental issues that facilitate silent data access from JS code [57].

Through manual experimentation and code reviews, we confirm that the WebView, Chrome, and Android permission models differ: while Android and Chrome prompt users when apps access dangerous permissions, WebViews allow developers to programmatically grant such permissions to JS code via the `WebChromeClient` class [47, 88] without notifying users, provided the app has the necessary permissions. In contrast, iOS displays visual warnings when JS code in a WebView accesses such resources, e.g., the microphone. Android 12 introduced privacy indicators to show when apps access permission-protected resources like GPS [46], but users still cannot distinguish whether access comes from the host app or arbitrary JS code loaded via RTB. This increases the number of actors potentially tracking users [57], as discussed in §3.3.

2.3 WebView Customization

Android allows app developers to programmatically customize default WebView properties [115]. However, these capabilities can expose users to privacy risks, as app developers may intentionally or unknowingly modify WebViews' default security guarantees and capabilities. Such customizability allows tracking actors to distribute and load harmful JS code or exfiltrate sensitive data:

- **Shared Local Storage.** By default, both DOM and local storage are isolated among apps to prevent unauthorized access. However, developers and third-party SDKs can invoke the `setDomStorageEnabled` method provided by the Web Storage API [49] to allow accessing local storage across apps if they originate from the same domain, enabling cross-app tracking. We developed a PoC to showcase this attack, reporting it to Google. To the best of our knowledge, no prior work has reported this abuse vector.
- **Web Cookie Management.** It is well known that websites use cookies to store and manage user and session information, but also for tracking. WebViews allow developers to manage cookies in two ways: (1) using JS APIs like `document.cookie` [116], or (2) using the Java-native API `CookieManager` [40]. As in the case of local storage, developers can programmatically enable third-party cookie storage in apps' data storage and control how they are handled within WebViews using `setAcceptThirdPartyCookies` [41]. However, third-party SDKs can abuse these APIs to gain access to cookies across apps without the app developer or user awareness. We built a PoC to demonstrate the feasibility of this attack and its privacy implications, also reported to Google. To the best of our knowledge, no prior work has studied the potential for abuse of these methods.
- **Java to JS Code Bridging.** Android supports APIs for bidirectional communication between Java code and JavaScript code [55], bridging native and web contexts and facilitating cross-platform tracking. Specifically, the `addJavascriptInterface` API allows developers (or third-party SDKs embedded in them) to expose Java Android methods to the web content, thus enabling JS running in a WebView to directly interact with native code [51] and through the JNI interface too. Similarly, the `evaluateJavascript` method allows developers to execute JS code within the WebView [52]. Additionally, developers can share sensitive information accessed from Java code (e.g., AAID) with web elements running on the WebViews by embedding it in the User-Agent (UA) or as URL parameters [39], facilitating ID bridging between web and native contexts. While prior work has proposed methods to detect privacy leaks in JS-Java channels [135], we are the first to empirically demonstrate their abuse at scale.

3 Privacy Attacks Using WebViews

This section describes the WebView privacy attacks that we empirically study in this paper. While some of these attacks have been theoretically hypothesized in prior work (See §8), our study is the first to systematically analyze how real-world app developers, SDKs, and RTB actors use WebViews and JS code for tracking purposes.

3.1 Fingerprinting

As in the case of web browsers [1, 13, 16, 23, 98, 111], WebViews allow JS code to use fingerprinting techniques that exploit both device-specific characteristics and metadata to track users without consent. Canvas fingerprinting, for example, exploits the HTML5 Canvas API by analyzing how their HTML5 canvas elements are rendered by a user's browser—which is influenced by device-specific differences in hardware, operating system, and graphic drivers—to generate device identifiers. On mobile platforms, this is particularly concerning: attackers can use JS APIs to produce user-unresettable and cross-application identifiers using only the standard INTERNET permission and WebViews. For example, browser fingerprints can be used to bridge resettable identifiers like the Android Advertising ID (AAID). In June 2021, we submitted a bug report to the Android Security Team demonstrating the possibility of canvas fingerprinting in Android WebViews and its privacy implications. The report was acknowledged and we were awarded a bug bounty in February 2022. In November 2024, this was not classified as a security concern and that WebViews work as intended.

3.2 Cookie Sync-ing and ID Bridging

Cookie sync-ing is a common web-tracking technique to link user IDs across domains that circumvents the Same-Origin Policy [1,

Listing 1: WebChromeClient implementation for geolocation permission.

```
webView.setWebChromeClient(new WebChromeClient() {
    @Override
    public void onGeolocationPermissionsShowPrompt(String origin,
            GeolocationPermissions.Callback callback) {
        callback.invoke(origin, true, true);
    }
});
```

120, 151]. It involves the exchange and mapping of cookies between different tracking entities and clients to synchronize user profiles.

WebViews' ability to bridge JS and Java execution environments facilitate bridging web-based cookies and IDs with device-specific PII obtained in Java code—including the AAID and Android ID—for persistent cross-platform tracking. Cookie sync-ing and ID bridging may occur alongside browser fingerprinting as WebViews facilitate trackers to aggregate multiple tracking paradigms to link users' online habits with Java IDs and sensitive data. In fact, cookie sync-ing is closely connected with *identity graph* services and cookieless tracking, like those offered by Adobe's Experience Platform [6], Lotame's Panorama Identity [102], or ID5 [92], among others.

Allowing third-party SDK WebViews to load arbitrary JS dynamically creates a favorable tracking environment where cookie sync-ing can occur without the user's knowledge or consent. However, unlike traditional browsers, where users have more visibility and control over cookie settings, WebViews often bypass such default protections by granting developers (and SDK operators) the opportunity to customize cookie management controls and the JS execution environment. This leads to a lack of transparency about how cookies are handled in WebViews.

3.3 Permission Piggybacking

WebViews allow JS code to programmatically access privacy-sensitive data like the GPS and microphone through the Navigator API [117]. App developers and even third-party SDKs can explicitly grant these permissions—inadvertently or intentionally, and potentially without the knowledge of host app developers—to JS code running on WebViews (§2.2), increasing the attack surface. Consequently, arbitrary organizations with the ability to load JS code on customized WebViews (e.g., advertisers reaching eyeballs through RTB) can opportunistically and silently collect sensitive geolocation and microphone data through these APIs in apps where such permissions are granted by the user [58]. The data obtained via those APIs can be linked with data obtained by fingerprinting, cookie sync-ing, or ID bridging methods to create rich user profiles.

Specifically, when developers implement WebViews with the `WebChromeClient` class in their apps, they simply need to declare the `onPermissionRequest` callback method to handle and grant permission requests to JS code without displaying a user prompt or notification. Although discouraged by Android's developer documentation [44], this abuse vector remains exploitable as of Android version 16 (Beta BP31.250502.008) and Android System WebView version 136.0.7103.60. While we build on the seminal work by Diamantaris [58], this study is the first to demonstrate this type of privacy abuse in real apps for tracking purposes, highlighting the lack

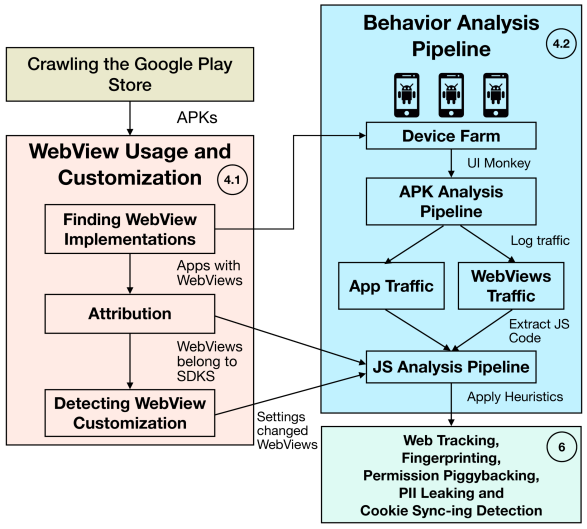

Figure 1: Overview of the methodology used in this study.

of user prompts caused by WebView customizations. We demonstrated the ability to access PII without user awareness with a PoC. Listing 1 exemplifies the case for accessing geolocation data.

4 Methodology

Figure 1 illustrates the research methodology that we designed to answer the two research questions defined in the introduction. We first use static analysis to identify the use of WebViews in Android apps, attribute them to first-party code or third-party SDKs, and identify WebView property customizations in §5. We then use dynamic analysis to capture execution behaviors in both the JS code embedded in WebViews and the app's Java code, and the communication channels enabled by developers between these two programming paradigms. Dynamic analysis is essential for detecting privacy-concerning behaviors such as access to sensitive JS APIs, fingerprinting, PII exfiltration, and cookie sync-ing in §6.

Dataset. We apply our methodology to a dataset of 13,045 Android apps downloaded from the Google Play Store between December 2023 and May 2024. We collected the apps using the *google-play-scraper* tool [118] from Spain. Our dataset covers 35 app categories, and 68% of the apps have at least 1M installs.

4.1 WebView Usage and Customization

We use static analysis to identify Java classes implementing WebViews, attribute them to the responsible actors (e.g., third-party SDKs), and detect WebView property customizations that alter their default privacy and security settings.

Finding WebView Implementations. The detection of WebView implementations across mobile apps is performed by looking for manually-curated unique WebView code signatures (i.e., class names and methods) extracted from popular open-source WebView implementations. Specifically, we use APKTool v.2.9.3 [147] to extract the Smali code of the 13,045 apps and detect the presence of the default Android WebView class `Landroid/webkit/WebView`,

alongside four alternative implementations: GeckoView [56], React Native, Apache Cordova, and Facebook WebView. Yet, SDKs often customize the Android default WebView, as reported in prior work [96]. We also search for alternative WebView engines by manually inspecting curated signatures and strings like "webview" in class names—validated through open-source references—in the Smali files. Yet, no other WebView implementations were identified in our dataset through manual code inspection.

Attribution. Third-party SDKs often integrate a customized version of the default WebView as §5.1 describes. We find the party responsible for its inclusion, distinguishing between first- and third-party WebViews by extracting the package name of the classes implementing them. We follow well-established academic methods to map code-level attribution signals to third-party SDKs [69, 96, 104] Yet, we update and curate the attribution signals offered by SDK detection tools like LibID [156], LibRadar [104], and Exodus [67], complemented with manual tagging if there is no match. Specifically, if the package name of the class implementing the WebView shares the prefix of the app package present in the Android Manifest, we classify the WebView as first-party; otherwise, it is classified as third-party and attributed to the relevant party by matching its attribution signals with our curated SDK attribution signatures. For example, from the structure of the package name of the AccuWeather app (`com.accuweather.android`) and the WebViews it embeds, we can distinguish between first-party (`com.accuweather.android.ui.components.WebViewKt`) and third-party ones (`com.google.android.gms.ads.internal.zzs`). Then, using our attribution signals, we determine that the latter belongs to Google's GMS SDK.

Detecting WebView Customization. We statically study how app and SDK developers customize WebView properties at the Smali code level, focusing on cookie and DOM storage, JS runtime access to permission-protected data (e.g., geolocation) and other default parameters like User-Agent or Google's Safe Browsing API. To do this, we first create a list of APIs allowing WebView property customization using Android's official documentation [48] (Table 2), so that we can statically detect their use across apps, including the parameters set by app and SDK developers when invoking them.

4.2 Behavior Analysis Pipeline

We develop a dynamic analysis pipeline consisting of two elements: (1) a farm of nine instrumented devices for automatic dynamic app analysis using custom Frida scripts (§4.2.1); and (2) a JS dynamic analysis pipeline built on the dynamic analysis tool Fakeium [110] and the static analyzer Esprima [65] to detect the privacy risks of JS code execution on WebViews (§4.2.2). We use instrumented devices for our analysis instead of emulators to avoid anti-emulation, anti-fraud or anti-root methods implemented by apps and SDKs [32], as discussed in §4.3.

Due to the dynamic nature of JS code and the predominant role of probabilistic RTB processes in distributing privacy-intrusive JS code through WebViews, it is essential to follow a best-effort and automatic approach to capture as many JS scripts and behaviors as possible. For that, we use geographically distributed vantage points and app automatization tools as described below.

Geographic Diversity. We use a geographically distributed device farm to test the Android apps for 5 minutes each in six world regions: Europe, the US, India, Brazil, Japan, and Australia. The devices run Android 13 and the default WebView version 125.0.6422.165. We use Tinyproxy [143] to transparently route device traffic through vantage points in these regions. Informed by prior work [119], we opt for HTTP(S) proxies instead of VPNs, as they are harder for anti-fraud tools to detect. Additionally, we use Frida [75] to mock GPS geolocation data, injecting coordinates corresponding to each region to mitigate potential inconsistencies in our execution environment. The diversity and geographic relevance of ads loaded in WebViews support the hypothesis of our approach being effective.

Automatization. We customized the DroidBot UI exerciser to automate app execution [101] so that it supports Android 12 and above (API Level 32+).[2] To automate the execution and increase coverage, our monkey employs a depth-first strategy to explore apps' UIs and supports standard logins, including Single Sign-On (SSO) screens like Google SSO, but not complex registration or login forms requiring CAPTCHA or 2FA.

4.2.1 APK Analysis. We use Frida to dynamically analyze APKs that integrate WebViews and to collect JS code loaded on them, which is later analyzed with a JS-specific analysis pipeline (§4.2.2):

- We combine Frida hooks with traffic analysis techniques to identify in runtime loaded WebViews, attempts to access the PII data types reported in Table 13 (see Appendix B), and their subsequent dissemination to first- or third-party hostnames. During dynamic analysis, we also identify loaded WebViews by hooking their initialization methods (e.g., `init()`) using Frida. This allows us to attribute runtime observations to the actual WebView in runtime and to the activity in which it is initialized.
- We capture app traffic using *tcpdump* [136] and a customized fork of the friTap tool [72] to log TLS keys (SSLKEYLOGFILE) to decrypt TLS flows [141]. For attribution, we compare the reverse SLD name of the receiving hostnames with the organization responsible for the WebViews based on the package name. We detect hostnames associated with ATSes by comparing them against anti-tracking lists like EasyList [60], EasyPrivacy [61], and Fanboy [24], a technique validated in prior work [86, 99, 126, 131]. For instance, in the bwin Casino app (`es.bwin.casino`), we identify the first-party hostname *casino.bwin.es* and Google's third-party hostname *ad.doubleclick.net*.
- We enable WebView remote debugging—disabled by default—to monitor browser engine runtime events reported by Chromium in real-time, capture loaded URLs, and extract web content, JS code, and traffic generated from the WebViews. We use Frida to activate it by setting the `setWebContentsDebuggingEnabled` flag to true in the WebView settings and then enabling the Chrome DevTools Protocol (CDP) [54]. We leverage the *chrome-remote-interface* library [27] to communicate with WebViews and monitor events generated while loading a URL and JS and their traffic, including `Page.navigate`, `Page.frameNavigated`, `Network.requestWillBeSent`, `Debugger.scriptParsed`. The analysis of these events allows us to detect potential privacy risks and monitor traffic, JS execution and failures in script loading, among

[2] https://github.com/honeynet/droidbot/pull/152

others. Table 14 in Appendix C lists the most relevant instrumented methods and their purpose. Through this process, we capture 10,788,450 different JS programs loaded across all apps.

We complement dynamic analysis methods with static ones to detect (*i*) hard-coded URLs in WebViews by detecting calls to the `loadUrl` and `loadDataWithBaseURL` methods; and (*ii*) programmatic permission granting to WebViews in `WebChromeClient`, as explained in §4.2.2.

4.2.2 JS Analysis Pipeline. We complement our APK and WebView instrumentation with the JS dynamic analysis tool Fakeium [110], a lightweight open-source sandbox built on the V8 engine for dynamically testing untrusted JS code. We use it to analyze JS execution within WebViews, monitoring API calls and relevant arguments (e.g., string literals) to study runtime behavior and potential privacy risks. Because of the dynamic nature of Fakeium, we can detect calls even in obfuscated scripts that would otherwise go unnoticed by static methods. We complement our dynamic analysis pipeline with the Esprima [65] static analyzer to increase coverage and find JS API calls potentially missed by Fakeium: where Fakeium gives us the lower-bound of the JS API calls in a JS script, Esprima gives us the upper-bound. Specifically, we analyze:

- **JS API Calls.** We monitor runtime calls to sensitive JS APIs for: (i) accessing permission-protected information, including media devices, geolocation, and sensor data obtained using Navigator APIs [117]; (ii) web tracking (e.g., cookie management); and (iii) generating device and browser fingerprints. Our set of targeted methods is informed by prior research in desktop web browsers [1, 16, 23, 94, 98, 111]. To analyze and quantify the extent to which JS code accesses permission-protected device resources and data in WebViews, including media devices and geolocation, we cross-reference the permissions declared and granted to the host app in Java-land through the AndroidManifest file with the permissions explicitly enabled for the WebView executing the JS code. To avoid introducing false positives, we determine whether the JS code directly invokes the relevant `Navigator` APIs to access these protected resources and attribute such behavior to organizations by analyzing the attribution signals at the hostname level.
- **Detecting PII Leakage.** We analyze WebView traffic to identify leakage of PII (Table 13 in Appendix B), examining both cleartext and hashed transmissions.
- **Cookie Sync-ing.** We inspect WebView traffic to identify the use of advanced tracking techniques such as cookie sync-ing. For that, we build on prior work [1, 63, 64, 151] to identify *ID Cookies*, which are unique identifiers shared between different domains or subdomains. These cookies have long expiry times, maintain stability across requests, exhibit constant length, and contain high-entropy value strings sufficient to uniquely identify users. Then, we generate network graphs to model the flow of such cookies between online servers, WebViews, and apps.

Our JS analysis pipeline complements our Java-level WebView instrumentation by revealing opportunistic attempts to access sensitive JS APIs and their runtime behavior as discussed in §2.

4.3 Limitations

Any empirical measurement analysis of mobile apps cannot achieve full coverage and completeness due to the need to rely on black-box testing methods. These are exacerbated by the challenges introduced by non-deterministic and dynamic processes like RTB, and the use of anti-testing and obfuscation techniques by app developers and SDKs. However, these aspects do not invalidate the main contributions of this study: WebViews facilitate user tracking through RTB and the collaboration of advertising SDKs due to inappropriate system architectures and controls.

Anti-testing: We could successfully test 96% of the apps and dynamically analyze thousands of WebViews in the wild. Through code inspection, we could identify the presence of the following app hardening methods:

- *Anti-testing:* 2.6% of the tested apps leverage the Google Play Integrity API [45] to detect rooted devices and Frida. The remaining cases use custom anti-testing techniques based on native code to detect instrumented devices [34].
- *Code Obfuscation:* Although 61% of the third-party WebViews are obfuscated using tools like ProGuard [85], manual validation shows that our attribution method accurately maps the WebView implementation to their respective SDK in 94% of apps.

Harmful Apps: 24 apps were not analyzed as they were flagged and blocked by Google Play Protect as potentially harmful apps at the time of testing [43].

Code coverage: WebView analysis coverage is fundamentally influenced by the probabilistic nature of RTB processes or anti-testing capabilities. As a result, not all embedded SDKs may be invoked in runtime or only winning advertising SDKs may render advertisements in their WebViews. In fact, manual inspection shows that many WebViews detected statically are dead code introduced by large SDKs offering multiple features to developers, as our Frida instrumentation also confirms. Despite this, our methodology offers a new perspective into privacy abuses in WebViews and also significant coverage advantages over prior work [96], where the authors injected a set of popular desktop domains in WebViews, rather than analyzing the URLs and JS code organically loaded on them in the wild. To validate our dynamic analysis approach and assess its coverage, we manually analyzed a sample of 50 apps, comparing the total number of WebViews implemented by the apps—detected using static analysis—with those WebViews triggered by our testing monkey. On average, our instrumentation achieves 37% dynamic coverage, with full coverage in certain cases, and at least 50% WebView coverage in 26% of the apps.

5 WebViews in the Wild

Our hybrid analysis instrumentation allows us to capture a wide range of JS and privacy-intrusive practices across 1,724 WebViews embedded in 10,971 apps as summarized in Table 1. The vast majority of apps (88%) implement at least one WebView. This includes 11,277 unique WebViews, out of which 9,283 are classified as third-party ones. In fact, as Figure 2 shows, the average app implements at least 6 WebViews, out of which 5 belong to advertising SDKs.

By correlating the presence of these SDKs with Google Play metadata, we observe that certain app categories integrate more third-party SDK WebViews, and some SDK providers have greater

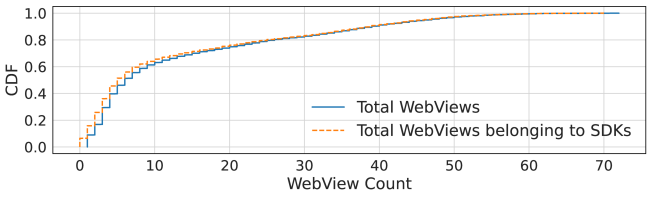

Figure 2: CDF of total WebViews per app (blue) and those belonging to 3rd-party SDK (orange).

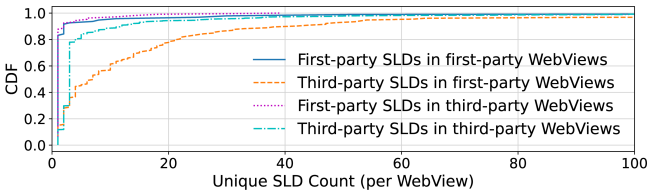

Figure 3: CDFs of the number of unique SLDs contacted per WebView. The x-axis is truncated at 100 to improve clarity.

Table 1: Dataset summary.

Type	# Total
Number of Apps Analyzed	10,971
Unique SDKs with WebViews	281
Unique WebViews Tested	1,724
Unique SLDs	4,129
Unique JS Code Samples	10,788,450

Table 2: Detected WebView customizations, reporting the percentage of apps involved and third-party SDKs involved.

Customization	% Apps	% SDK	# SDKs	Examples
setJavaScriptEnabled	98%	83%	172	Fyber, MS Clarity
setUserAgentString	88%	16%	26	ByteDance, Mintegral
setDomStorageEnabled	79%	39%	81	Tealium, PubNative
setAcceptThirdPartyCookies	54%	13%	13	Taboola, KIDOZ
setSafeBrowsingEnabled	22%	1%	5	CellRebel, Mintegral

market presence [69, 126]. Specifically, *GAME* apps integrate a median of 9 third-party SDK WebViews, while most other categories integrate only two. This aligns with the ad-driven business model of game apps [87, 127]. Figure 5 in Appendix E shows the percentage of apps integrating any of the top-10 SDKs per category. Google Mobile Services (GMS) and Facebook Ads dominate with 69% and 40% market presence, while SDKs like Fyber, Unity Ads, IronSource, and Inmobi are more common in games. While our findings align with prevalence values reported by Kuchhal *et al.* [96], our methodology reveals greater diversity and usage of WebViews due to our improved WebView detection and attribution techniques.

5.1 WebView Customization

As described in §2.3, developers can programmatically alter the default privacy properties and runtime behavior of WebViews. Through static analysis, we find that 99% of the tested apps customize WebViews. As summarized in Table 2, most apps allow JS execution (98%), enable DOM storage sharing (79%), or customize UA strings (88%). Advertising and tracking SDKs like Smart Instream [93], Yandex Ad [154], and TopOn [146] are responsible for 84% of WebView customizations. Interestingly, Mintegral and Byte Dance intentionally disable Google Safe Browsing features [82], which unnecessarily exposes mobile users to web threats by removing Google's capabilities to detect and block the execution of harmful web content. These third-party practices raise concerns about whether app developers integrating such SDKs are aware of the potential security and privacy issues of WebView customizations. As we study in §6, WebView customizations facilitate fingerprinting, cookie sync-ing, and PII dissemination.

5.2 Beyond Third Parties

As discussed in §3, RTB increases users' privacy risks by allowing any arbitrary party (henceforth referred to as a *fourth party*) to opportunistically execute JS code with tracking capabilities on third-party SDK WebViews used for advertising. Consequently, it is infeasible to determine which (and how many) hostnames could potentially load harmful JS code across WebViews. Yet, our best-effort and geo-diverse testing approach captures traffic between WebViews and 7,999 fully qualified domain names (FQDNs).

By reasoning about the SLDs of the hostnames, and the package names of both apps and SDKs as described in §4.2.1, we classify them in four categories: first-party SLDs in first-party WebViews (same organization), third-party SLDs in first-party WebViews, SLDs from the same organization responsible for the third-party WebViews, and third-party SLDs in third-party WebViews (i.e., fourth-party SLDs). The process of classifying hostnames as first-, third-, or fourth-parties is not always straightforward due to apps using code obfuscation or techniques such as CNAME tracking (see §6.1) that reduce the number of attribution signals in certain cases. In addition, 1% of the JS scripts are hosted in popular CDNs and cloud services such as *b-cdn.net*, *cloudfront.net*, or *akamaihd.net*. In some of these cases, we can manually identify the organization responsible for them by inspecting their third-level domain as proposed in prior work [108, 126]. For example *mappls.b-cdn.net* belongs to *mappls.com*, a navigation service provider [105].

Figure 3 plots the CDF of the number of unique first-, third- and fourth-party SLDs per WebView category (first- or third-party WebViews). This figure confirms that both first- and third-party SDK WebViews are more likely to load arbitrary exogenous third- and fourth-party content: 90% of first-party and third-party WebViews load at least 7 and 12 exogenous domains, respectively. This confirms the key role that third-party SDK WebViews play as JS distribution channels. Notably, during our execution, advertising SDK WebViews like GMS reaches 119 SLDs, Applovin 104, and Unity Ads SDK 51 due to RTB processes. This finding supports our initial hypothesis: WebViews significantly amplify the number of parties with potential for tracking users opportunistically and at scale through JS code executed within mobile apps, particularly when associated with RTB processes.

ATSes on WebViews. A visual inspection of the cloud services loading content on the WebViews suggests a prominent presence of

Table 3: Top-10 ATSes observed across WebViews.

FQDN	#Apps	#WVs	# SDKs	Examples
googleads.g.doubleclick.net	5,601	67	14	GMS, Applovin
csi.gstatic.com	3,052	8	1	GMS
tpc.googlesyndication.com	2,666	31	9	GMS, ByteDance
cdn.iads.unity3d.com	842	11	1	IronSource
ae.iads.unity3d.com	690	10	1	IronSource
s0.2mdn.net	642	20	7	GMS, Applovin
imasdk.googleapis .com	630	13	1	GMS
www.googletagmanager.com	603	121	15	Fyber, ByteDance
ad.doubleclick.net	533	36	10	GMS, Taboola
www.google-analytics.com	484	96	13	SafeDK, Applovin

ATS organizations according to well-known ad-blocking and anti-tracking blocklists [60, 61, 138]. In total, we find that 1,211 SLDs (29% of all domains) are associated with ATSes. Table 3 lists the top-10 ATS hostnames most frequently contacted by WebViews: Google and Unity/IronSource are among the most prominent ones. The wide presence of `Referer` headers allows analyzing the RTB chain involved in these processes. ATS eyeball reach is also significant, as 24% of the ATS SLDs are found in at least two different third-party WebViews despite the completeness limitations of our instrumentation. For example, the Google-owned FQDNs *csi.gstatic.com* and *tpc.googlesyndication.com*, which track the performance of advertising campaigns [89] and serve ad content [81], load their content on WebViews implemented by SDKs like Applovin, ByteDance, and Fyber, among many others.

Hardcoded Hostnames Loading on WebViews. We find 69 organizations hardcoding URLs to fetch JS code in 96 different WebViews. However, we observe noticeable differences in the use of some of those URLs, as they are only accessed in specific world regions. One frequent example is *fundingchoicesmessages.google.com*, linked to Google's Funding Choices Consent Management Platform (CMP) [95], which helps publishers manage user consent for GDPR [122] and CCPA [26] compliance [79]. Other hardcoded hostnames belong to SDKs providers like Fyber [148], Taboola [134], Amazon Ads [12], Bigo Ads [22], Adivery [2], and Yandex Ads [154], present in 16% of the total apps in our dataset. Table 15 in Appendix D provides examples of these URLs along with their associated SDKs and the regions where they were observed. This result corroborates the importance of having geographically distributed vantage points to capture a broader diversity of JS samples loaded on WebViews.

6 Tracking Behaviors on WebViews

In this section we study how WebViews' execution model and differences in Android's privacy enforcement mechanisms between Java and JS execution environments are abused in the wild for user tracking purposes. Specifically, we provide empirical evidence of WebView-based tracking using cookies and impression pixels (§6.1); web fingerprinting techniques (§6.2); silent dissemination of sensitive permission-protected data such as geolocation through JS API methods in WebViews (§6.3); abuse of the JS-Java bridging channels to leak unique identifiers and other PII (§6.4); and cookie sync-ing and ID bridging activities across WebViews and endpoints facilitating cross-platform user tracking (§6.5).

Table 4: RCs, SCs and impression pixels in WebViews.

Type	Count	% Apps	WebViews		SLDs	
			% Total	% SDKs	# Total	% ATS
RCs	36,307	53.9%	31%	44 (42%)	1,044	40%
SCs	55,599	52.6%	35%	38 (37%)	1,206	29%
IPs	85,875	53.4%	23.3%	46 (44%)	346	61%

6.1 Web Cookies and Impression Pixels

By parsing the `Set-Cookie` header within response data to extract relevant cookie details and identify the hostname associated with them, we find that most web content loaded on WebViews perform traditional web tracking methods like web cookies and impression pixels, both for session management and user tracking. To better characterize these behaviors, we define the following categories:

- Receiving Cookies (RCs) sent downstream from a server to the client in the `Set-Cookie` HTTP response header.
- Sending Cookies (SCs) sent upstream by the client in the `Cookie` HTTP request header.
- Impression Pixels (IPs) embedded in web pages for behavioral tracking across domains without user's explicit knowledge [18].

Additionally, we define a *session cookie* as one that does not have an expiration attribute, following the criteria established in Mozilla's Developer forums [114], and we use the detection methodology outlined in previous work for detecting impression pixels [18, 74]. We also cross-reference the domains that we find using these tracking methods with the information in the `Referer` header and the known ATS domain block lists listed §4.1 to distinguish between first- and third-party tracking activities.

Table 4 provides an overview of the RCs, SCs and IPs found loaded across WebView tests. Overall, 39.3% of WebViews run JS code setting at least one cookie or an impression pixel. Out of the 81,340 cookies detected across 288 WebViews, 49% are potential tracking cookies, and 37% are associated with ATSes. With respect to impression pixels, we find their use in 172 WebViews, 60.4% of which are associated with ATS hostnames.

By analyzing the hostnames responsible for these actions, we find that they are well-known ATSes accessing third-party SDK WebViews by Taboola, Applovin, Mintegral, Yandex Ad, Bigo Ads, ChartBoost, AppNexus, PubMatic, and ByteDance. For example, 58% of the RCs come from domains associated to identity graph solutions like *demdex.net* (Adobe), *id5-sync.com*, and *rubiconproject.com*. We further study these organizations in our cookie sync-ing analysis (§6.5). Similarly, 29% of the SLDs involved in SCs belong to ATSes such as *doubleclick.net*, *pubmatic.com*, and *taboola.com*, while 27% of the impression pixels come from ATSes like *google-analytics.com*, *facebook.com*, and *tealiumiq.com*.

Impact of WebView Behavior Customizations. The default same-origin policy enforced by the WebView's underlying web engine restricts cookie access to matching hostnames, potentially rendering third-party RCs ineffective unless developers programmatically relax these restrictions through WebView customizations (§5.1). To investigate this, we cross-reference the *Network.response*

Table 5: Top fingerprinting-related API calls by origin.

JS API	# FQDNs
navigator.userAgent.indexOf	6,897
navigator.userAgent.toLowerCase	6,753
OffscreenCanvas	4,877
navigator.userAgent.match	1,673
customElements.get	830
__core-js_shared__.state.has	660
__core-js_shared__.state.get	638
customElements.define	360
navigator.sendBeacon	294
HTMLElement	288

ReceivedExtraInfo CDP event log to determine which RCs are actively blocked by WebViews at runtime. Surprisingly, 99% of third-party RCs are programmatically allowed by third-party SDKs including GMS, Applovin, and Taboola.

CNAME Tracking. Certain first-party FQDNs exhibit behavioral characteristics attributable to CNAME tracking. ATSes use this sophisticated technique to bypass same-origin privacy protections by masquerading themselves as first-party hostnames through CNAME DNS records [59]. Consequently, when a WebView loads content from what appears to be a first-party subdomain, it allows the execution of code by a third-party tracker. To systematically detect this behavior, we follow the methodology outlined in prior research [33, 59], i.e., we derive the DNS CNAME chain to identify CNAMEs that align with known tracking entities. Additionally, we cross-reference the use of SCs and RCs with the domains found using CNAME tracking. We find that 90% of the hostnames involved in CNAME tracking use both SCs and RCs. Behind these CNAMEs we identify organizations like Ensighten, Adobe Experience Cloud, LexisNexis [100], Pardot [121], or Utiq [150].

6.2 Fingerprinting

WebViews enable Java applications to perform browser fingerprinting and generate unique, persistent device IDs without requesting special Android permissions. These fingerprints combine data from multiple JS APIs that reveal device-specific traits like hardware, browser engine, OS, and graphics drivers. Table 5 lists the top JS APIs used for fingerprinting, grouped by the number of FQDNs where at least one invocation was observed within a WebView. Three of these potentially sensitive APIs appear across nearly all contacted FQDNs, while others are limited to a few hundred. We present two illustrative case studies showing browser and canvas fingerprinting in WebViews, including both legitimate and secondary usages such as anti-fraud and advertising, respectively.

Browser Fingerprinting. We manually search for evidence of unique IDs generated and leaked from JS code. First, we look for references to popular fingerprinting libraries like FingerprintJS [71] in the collected JS samples, including library mentions and code snippets. We then identify the corresponding app and WebView that loaded the script, and extract all HTTP requests originating from that WebView. Finally, we manually inspect URLs, headers, and payloads for strings indicative of fingerprinting. We consider a case plausible if it (i) contains at least 10 hexadecimal characters and (ii) is specific to a WebView.

Table 6: Canvas fingerprinting captured in WebViews.

FQDN	App
movil.bbva.es	com.bbva.bbvacontigo
android-fanatics.frgapps.com	com.fanatics
consumercenter.mysynchrony.com	com.gpshopper.discounttire
www.synchrony.com	com.gpshopper.discounttire
fcid.fidelitycharitable.org	com.fidelity.android
www.fidelity.com	com.fidelity.android
www.jcpenney.com	com.jcp
www.michaels.com	com.michaels.michaelsstores

Using this process we find examples such as the following: the *com.kiwi.westbound* app [77] loads on a first-party WebView the landing page *https://popreach.com/privacy-policy/*. The WebView navigated from the initial URL to *https://www.bamboohr.com*, which embeds an external script[3] that creates the fingerprint. Once created, the ID is sent to three distinct FQDNs: a demand generation company [37], a tracking service [30], and a programmatic ad platform [21]. We detect analogous behavior for the *com.htwig.luvmehair* app, where the WebView accesses *https://shop.luvmehair.com*, which embeds a fingerprinting script from *https://tracking.server.bytecon.com/fp.js*. After obtaining the fingerprint, the ID is sent in the *visitor_id* query parameter to the tracking domain *tracking.server.bytecon.com*. Repeated testing on three devices shows that the script generates a stable and distinct ID on each device.

Overall, each captured browser fingerprint script accesses over 20 JS APIs that are classified by FPMON [70] as either *sensitive* (e.g., User Agent, CPU class) or *aggressive* (e.g., CPU concurrency), based on their prevalence and entropy across users. We note that our ability to capture fingerprint abuses varies across devices due to factors such as the probabilistic nature of RTB, (geo)location-dependent behavior, and differences in monkey interaction patterns.

Canvas Fingerprinting. Canvas fingerprinting is a special category of fingerprinting that leverages rendering inconsistencies in HTML5 Canvas elements to generate identifiers across different web contexts. This section reports instances of canvas fingerprinting observed in WebViews, as Table 6 summarizes. We detect three uses of canvas fingerprinting across apps: fraud detection (1 app), identification of bots (4 apps), and establishing persistent cross-platform user tracking (1 app). For example, the banking *com.bbva.bbvacontigo* app loads scripts from the FQDN *movil.bbva.es* to perform canvas fingerprinting likely to minimize fraud risks. This is evidenced by the integration of Akamai Bot Manager [11], which employs canvas fingerprinting scripts loaded from the */akam/* URL path. However, Table 6 also reports a case like *com.gpshopper.discounttire* in which there is no direct correlation between the app name and the FQDN loading the fingerprinting script. This suggests its potential user tracking purpose.

6.3 Accessing to Sensitive JS APIs

We apply the methodology outlined in §4.2.2 to detect JS code attempting access to permission-restricted geolocation and media devices through the *Navigator* APIs. We remind the reader that just calling *Navigator* APIs does not always guarantee access to the protected resources as the following requirements must all

[3] https://cdn.jsdelivr.net/npm/@fingerprintjs/fingerprintjs@3/dist/fp.min.js

Table 7: JS code successfully accessing permission-protected GPS data and media devices in WebViews.

Data Type	# Scripts	# 3rd-party SLDs	Success Rate	% Apps
Geolocation	290	74	57%	6%
Media	38	15	21%	<1%

Table 8: Examples of SLDs collecting sensitive data. (*) denotes development framework WebViews like React Native.

SLD	# Apps	SDKs	GPS	Media	AAID	And. ID	Email
unity3d.com	34	Tapjoy	✓		✓		
taboola.com	26	Taboola	✓		✓		
forter.com	9	*	✓	✓	✓		
rubiconproject.com	3	GMS		✓	✓		
aplicacionesyrecursos.com	1	*	✓	✓		✓	
getwarmly.com	1	*		✓			✓
borgataonline.com	1	*	✓		✓	✓	

be met: (i) the host app must request the corresponding Android permissions in its AndroidManifest.xml file to access geolocation or media devices; (ii) the WebView must be configured to run JS, and developers or SDK providers must grant access to the WebViews; and (iii) users must grant the runtime permission to the host app.

Table 7 summarizes our findings, highlighting the number of scripts opportunistically attempting access to these APIs and their success rate. Despite the opportunistic nature of this type of attack, the success rate is fairly high as most apps request access to these permissions. As we can see, 290 distinct JS scripts from over 74 third-party SLDs invoke the *navigator.geolocation* API in WebViews. These calls are successfully executed in 57% of cases across 6% of the tested apps. This is possible due to the fact that 22% of the apps in our dataset not only request access to the geolocation permission in their manifest files, but also programmatically grant their access to WebViews. As Table 8 exemplifies, ATS hostnames associated with Unity, Taboola, and Forter successfully collect GPS data from WebViews implemented by popular advertising SDKs like TapJoy and Taboola across dozens of apps, or their own anti-fraud solutions (e.g., Forter). Similarly, 38 scripts attempt to access the microphone and camera capabilities through the `navigator.mediaDevices.getUserMedia` API. The majority of these attempts also come from 15 ATSes such as Applovin, Mintegral, and GMS. Yet, as these permissions are more rarely requested by apps, they are successfully accessed in only 21 of the cases.

Our results demonstrate that the feasibility and success rate of opportunistic access to permission-protected resources through Android WebViews is fairly high, particularly for geolocation data. The dynamic nature of JS code execution in WebViews—often distributed through RTB without going through any vetting process—coupled with the lack of privilege separation between WebViews and their host apps, facilitates unauthorized and silent access to protected data. Allowing arbitrary hostnames to collect sensitive geolocation data along with user IDs—discussed in §6.4—facilitates persistent geo-tracking for commercial, advertising and surveillance purposes, including data brokerage. This risk is exacerbated by the fact that, unlike traditional web browsers, WebViews lack explicit user prompts to grant access to these permissions at runtime to JS code, allowing these practices to occur without user awareness as we show in §3.3.

Table 9: Java methods to leak PII to WebViews. * indicates that they are invoked in first-party WebViews only.

Method	# Apps	% SDKs	Example	SLDs	And. ID	AAID	GPS
evaluateJavascript	1,556	17%	IronSource	57	✓	✓	✓
loadDataWithBaseURL	1,055	29%	Applovin	58	✓	✓	
loadUrl	190	17%	Tapjoy	141	✓	✓	✓
loadData	27	1%	-	23	✓		
postUrl	3	*	-	5	✓		
setUserAgentString	7	*	-	55	✓	✓	✓

Unsupported JS APIs. We found 75 JS scripts from 16 first-party and 19 third-party SLDs attempting to invoke APIs not yet supported by the default Android WebView, some of which are experimental APIs according to Mozilla Developer Guides [112]. The anti-fraud service *forter.com* and a prominent Russian Internet corporation *mail.ru* load 11 distinct scripts that attempt to access the Web Bluetooth API via `navigator.bluetooth.getAvailability`, and, if successful, register event listeners using `navigator.bluetooth.addEventListener` to monitor Bluetooth activity. This information can be accessed by location services specialized in BLE beacons like Estimote [66]. Furthermore, two other services attempt to access APIs for GPU (`navigator.gpu`) and VR display access (`navigator.getVRDisplays`), whose usage allows device fingerprinting [97]. For reference, Table 12 in Appendix A reports other unsupported APIs invoked by JS code executed on WebViews. Although these JS APIs are not currently supported by WebViews yet, they may eventually become accessible and new risks may appear. In fact, as reported in the Chromium Issue Tracker [113], the Chromium development team is planning to support Bluetooth APIs for WebViews, which may facilitate user geo-tracking, cross-device tracking and profiling [78].

6.4 Exploiting Java ⇒ JS Channels

Our WebView instrumentation allows us to detect the dissemination of PII from Java code to WebViews, and from there to the cloud. This PII includes IDs accessible only in Java code like the AAID and Android ID, thus facilitating cross-platform tracking. Table 9 presents an overview of Java methods enabling these channels along statistics of their use across apps, SDKs and the ultimate endpoints receiving the PII:

- **evaluateJavascript.** 17% of the SDKs use this method to inject PII into WebViews. This practice is observed across 1,556 apps, resulting in the exfiltration of PII to 57 SLDs like *tealiumiq.com* and *appsflyer.com*. At the SDK level, IronSource injects the AAID into WebViews, which is then collected by subsidiary SLDs like *unity3d.com* and *supersonicads.com*.
- **Embedding PII on URLs and HTTP Headers.** A common technique used by SDKs to leak PII to WebViews is embedding it in URLs and HTTP headers using *loadDataWithBaseURL* (29% of SDKs) and *loadUrl* (17% of SDKs). We observe that sensitive

Listing 2: Android ID concatenation with the User-Agent.

```
"User-Agent": "platform/vogo_sdk platformVersion/1 baseName/base_vogo_android
      baseVersion/249 androidVersion/13 deviceBrand/google deviceId/
      5b9ac306cb5afdbc deviceName/Google-Pixel 6a"
```

Table 10: PII shared with cookie sync-ing organizations.

PII	% SLDs	% SDKs	# ATS SDKs	Examples
AAID	50%	82%	16	Mintegral, TopOn
Andr. ID	15%	41%	7	Smart Instream, Applovin
EICD	9%	32%	5	Smart Instream, MS Clarity
Email	1%	14%	2	MS Clarity, ByteDance
GPS	8%	41%	7	Taboola, ByteDance

IDs sent using these methods are subsequently transmitted to 141 SLDs, including prominent ATSes like *google-analytics.com* and *demdex.net*, the latter being associated with Adobe Experience Platform Identity Service. While less frequent, we also observe the use of the *loadData* and *postUrl* methods. For example, the Applovin SDK shares the AAID with 32 organizations like *appsflyer.com* and *adjust.com* by embedding it in URLs loaded in WebViews using the `loadUrl` method in 411 apps in our dataset. Another interesting case is JS code from *unity3d.com* loaded in the Tapjoy SDK, gaining access to both geolocation data and the AAID using the *loadDataWithBaseURL* method.

- **User-Agent (UA) Leakage.** The Indian car and scooter rental app *com.VoDrive* leverages the `Settings.Secure.getString` method to retrieve the device's Android ID. This value is subsequently concatenated into the UA field by the app developers (Vogo) [152] (see Listing 2). We observe the transmission of this value to *razorpay.com* and *loginwithamazon.com*. This concerning leak arises because, once a WebView user agent field is set either intentionally or by mistake, it is automatically included in **all** the HTTP requests generated by this WebView, regardless of the destination. Similarly, *forter.com*, an identity graph provider [73], successfully receives the AAID via the UA, then loads JS scripts in 9 apps to attach it to users' geolocation data.

Table 8 presents examples of SLDs and SDKs using these methods to collect user IDs and attribute permission-protected geolocation data to users. We note that we did not find any instances of IMEI or MAC address leakage using these methods.

6.5 Cookie Sync-ing and ID Bridging

We capture evidence of cookie sync-ing between 1,190 SLDs—1,059 origin and 1,082 destinations—, and ID bridging. Figure 4 shows cookie sync-ing events between SLDs with at least three occurrences. Around 44% of the SLDs involved in cookie sync-ing belong to ATSes. These practices are observed across WebViews by 19 third-party advertising SDKs like ByteDance, Tappx, IronSource, or Yandex. For example, we find 9 unique ID cookies synchronized between Taboola (*taboola.com*) [134], an ad platform, and Tealium iQ (*tealiumiq.com*) [137], a customer data management platform, on WebViews implemented in Taboola's SDK. We also observe interactions between Adobe (*adobe.com*) and Expedia (*expedia.com*) [68] through Adobe's Creative SDK [3] WebViews. Similarly, Unity (*unity3d.com*) and Taboola synchronized 11 cookies on IronSource's ad SDK WebViews. Google also performs cookie sync-ing across *adservice.google.com*, *id.google.com*, *play.google.com*, and *apis.google.com*. In fact, GMS' WebViews account for 31% of the total cookie-syncing instances.

PII Dissemination and Cookie Sync-ing. We cross-reference the SLDs receiving any of the PII types listed in Table 13—in addition to Adobe's Experience Cloud ID (EICD), a persistent ID that uniquely identifies visitors in Adobe's Experience Platform [7]—with those entities involved in cookie sync-ing. Table 10 reports the observed dissemination of PII to SLDs, and the SDKs facilitating it:

- **AAID:** 50% of cookie sync-ing SLDs also collect AAIDs, including *pangle.io*—a TikTok subsidiary—the tracker *tapad.com*, and *rubiconproject.com* through WebViews from SDKs like IronSource, Gadsme [76], and Bigo Ads [8].
- **Android IDs:** We find 15% of the SLDs receive the unresetable Android ID, including *doubleclick.net*, *tiktok.com*, and *omtrdc.net* (an Adobe service [5]) through WebViews from GMS, Taboola, and ByteDance.
- **EICD:** We find 9% of SLDs involved in cookie sync-ing receive Adobe's EICD, including hostnames by The Trade Desk [38], Google Analytics and Bing in WebViews belonging to SDKs like MS Clarity [109], Smart AdServer [93], and ByteDance.
- **Email:** We observe that 1% of cookie sync-ing SLDs contain hashed email addresses (HEMs). While this data type cannot be accessed programmatically, these emails correspond to the ones introduced by our monkey in registration forms while testing the apps. Notably, the email hashes are appended to Meta Tracking Pixels [107] sent to *facebook.com/tr* along with the Android ID, AAID and EICD in the same HTTP query as Listing 3 shows. However, hashing user data does not guarantee anonymity, as noted by the FTC [62, 133].
- **GPS:** Finally, 8% of SLDs receive geolocation data with synchronized cookies, AAID, Android ID, and EICD. This includes *snapchat.com* (a first-party WebView), *rubiconproject.com* on Mintegral, Taboola, Appodeal, and GMS SDKs.

All these cookie sharing and ID bridging practices pose severe privacy risks by allowing these entities not only to perform cross-app tracking but also to correlate individual users with their geolocation data over time. Unfortunately, this data is available for purchase (and misuse) through data brokers, as repeatedly demonstrated by investigative journalism and regulatory efforts [31, 123, 130, 140].

7 Discussion

Our empirical findings confirm significant privacy risks associated with the misuse of Android WebViews in the wild, particularly by third-party advertising SDKs. Particularly concerning is the dissemination of sensitive geolocation data from WebViews linked to unique user IDs, alongside widespread cookie sync-ing. These risks are amplified by WebView's relaxed permission model and the ability of third-party SDKs to disable default security guarantees programmatically, as discussed in §5. For instance, canvas fingerprinting can persist when combined with other device properties.

These practices facilitate cross-platform tracking and contribute to identity graph construction, potentially undermining existing privacy controls and privacy regulations like GDPR, CCPA and COPPA, especially in child-targeted apps that collect geolocation data [127].

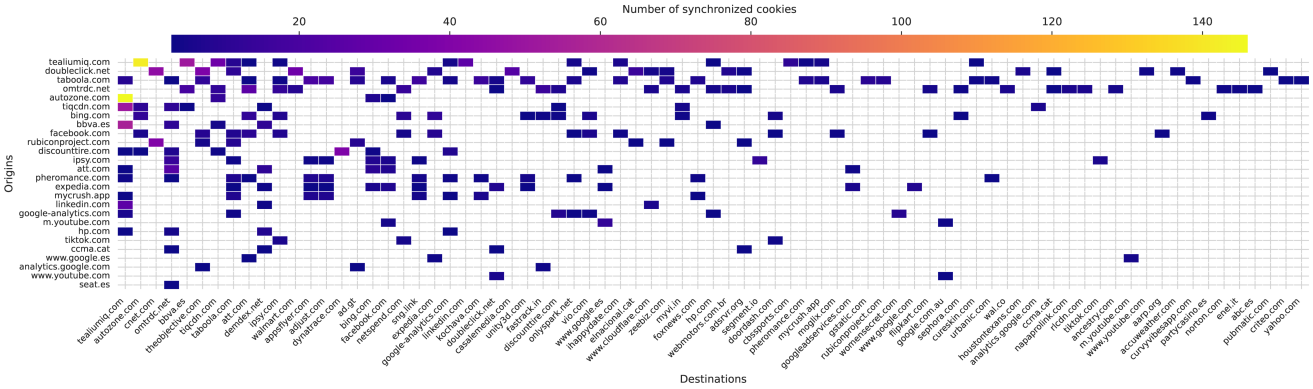

Figure 4: Cookie sync-ing between SLDs. Only pairs of SLDs exchanging at least 3 cookies are rendered for clarity.

Listing 3: Email hash (`udff[em]`) uploaded with Meta Tracking Pixels.

```
https://www.facebook.com/tr/?id=1468083936810812&ev=CompleteRegistration&dl=
      https%3A%2F%2Fsecure.aarp.org%2Fapplications%2Fuser%2Flogin%3Fclient_id%3
      D0oagxm3qk9E4GdgAL2p7%26redirect_uri%3Daarpst%3A%2F%2Fauthenticate%26
      response_type%3Dcode%26scope%3Doffline_access%2Bem%2Bbui%2Bbmi%26promo%3
      DCMTI%26state%3D5d336375-4704-4cbf-9d01-61492cde8c27&rl=&if=false&ts
      =1726177430240&cd[currency]=USD&cd[content_type]=product&cd[content_ids
      ]=%5B%22login%22%5D&sw=412&sh=915&udff[em]=
      658376e6824870dbf25b8f318843a5b88c114218c299683a19bd4325629c2&v=2.9.167&r=stable&ec
      =3&o=6174&fbp=fb.1.1726177389993.626457883380397359&cs_est=true&
      est_source=239921730312374&ler=empty&it=1726177388695&coo=false&es=
      automatic&tm=3&rqm=GET
```

In fact, gaming apps show the highest incidence of web cookies, impression pixels, PII leakage, cookie sync-ing, and browser fingerprinting, while canvas fingerprinting is more common in finance and shopping apps, likely for anti-fraud purposes.

Yet, the implications of hybrid privacy abuses in WebViews vary across stakeholders. Users face privacy loss and potential exploitation of their data by brokers or authorities in jurisdictions with weak human rights protections, unable to rely on existing controls. Developers remain liable for integrating SDKs that engage in such practices, often unknowingly. Ad networks' customization of default WebView privacy properties facilitates the execution of privacy-harmful JS via RTB processes. Platform checks like Google Play Protect, based on static analysis, may fail to detect or prevent these covert flows due to their dynamic nature [153], thus failing to safeguard users. Regulators also face difficulties capturing reproducible evidence, as the dynamic execution of JS complicates investigations into deceptive practices and enforcement of rules like GDPR [122] and CCPA [26].

7.1 Mitigations

Addressing the issues reported in this paper requires coordinated efforts across platforms, developers, ad networks, and regulators:

- Platform-Level Enhancements: Android's WebView architecture and permission model require a thorough redesign to better reconcile Java and Web paradigms and restore user control. It is essential to (i) restrict methods that allow WebView customizations weakening privacy defaults and enabling side-channels between Java and JS code, and (ii) introduce fine-grained, user-friendly controls to limit WebView tracking. Platforms could mandate clear, standardized disclosures of WebView data practices and provide real-time notifications when WebViews access sensitive data like geolocation, similar to AOSP permissions and iOS' approach. Unlike Android WebViews, iOS' WKWebView mitigates privacy issues like silent permission piggybacking by prompting users when web scripts attempt to access geolocation or media devices inside apps.
- App Store Policies: Platform operators should enforce stricter vetting processes and platform policies to control the abuse of WebViews and the dissemination and execution of privacy-intrusive JS code. Policies and industry standards could mandate justification for WebView customizations and require adherence to basic privacy principles and anti-tracking standards. Finally, enhanced detection and blocking of harmful dynamic JS code loading on WebViews at runtime could be introduced to limit abuse via Play Protect. Apple's stringent rules for WebView customization could inspire similar policies for Android [14]. However, the probabilistic nature of online ads makes abuse detection hard without the collaboration of ad networks.
- Ad Network Accountability: Ad networks should be responsible for vetting the JS code they distribute, limiting data collection, and ensuring compliance with privacy regulations. Google Ad Manager recommends CTs as an alternative for rendering ads, despite their current beta status [83] as it enforces stricter security controls such as limiting JS injection, which may help mitigate issues like permission piggybacking. While CTs may offer stronger safeguards than WebViews in paper, their usage is just recommended and further research is needed to assess their properties and correct use in the wild.
- User-Centric Features: Borrowing from Chrome's Privacy Sandbox or settings to limit ad tracking at the platform-level, WebViews should include configurable anti-tracking settings to empower users. Allowing users to choose third-party WebViews with certified anti-tracking features could provide additional safeguards to users while fostering competition, but at the risk of increasing platform fragmentation [124].
- Regulatory Enforcement: Regulators should push for transparency requirements in in-app browsing technologies and encourage accountability among all stakeholders and the RTB chain. Enforcement mechanisms could be bolstered by collaborating with

platforms and ad networks to detect and address covert data harvesting through WebViews at scale, and to investigate how data harvested through RTB processes is ultimately feeding into data markets or identity graph solutions.

Due to the sensitivity of the data collected through WebViews and the limitations in detecting these behaviors, we believe that developing mitigations to protect users' privacy rights should prevail over ensuring usability, legacy support, and compatibility.

8 Related Work

Several studies have explored the privacy-risks of in-app browsing in Android and iOS. While we use some of these papers to inform our analysis pipeline and our set of privacy threats, we are the first ones to provide evidence of multiple privacy-intrusive practices exploiting the hybrid nature of WebViews to bridge web and mobile tracking, involving popular SDKs and third-party ATSes.

Vulnerabilities and Attacks. Prior studies performed vulnerability analysis to detect security issues on in-app browsing. Luo et al. [103] introduced *Touchjacking attacks*, which exploit UI-based vulnerabilities in WebViews. Mutchler et al. [115] revealed that 28% of mobile apps have security vulnerabilities stemming from improper WebView configurations, like mishandling SSL certificates. Further, Thomas et al. [139] highlighted the JS-to-Java interface vulnerability in WebView, focusing on remote code execution. Recently, studies like those by Beer et al. [19, 20] have explored vulnerabilities in CTs, revealing how shared browser states can be exploited to leak sensitive user data. Zhang et al. [158] demonstrate how WebView APIs for local storage and cookie management can be abused for web manipulation, building a static analysis tool to detect their abuse in the wild. Choi et al. [29] analyzed phishing detection failures in WebViews, showing that user-agent-specific phishing sites evade blocking even after being reported to Google's Safe Browsing and built-in app reporting systems.

Privacy Concerns. Yue [155], Das et al. [35] and Zhang et al. [157] explored how WebViews allow tracking as JS accesses to motion sensors can produce user fingerprints without explicit consent. Tang et al. [135] proposed a dynamic tainting solution to detect privacy leaks in JS-Java bridges. We build on their findings to detect their abuse in the wild by real-world JS code by ATSes distributed through RTB on WebViews. Studies have also showed how sensitive data could be exfiltrated through cross-zone and UXSS attacks [132] Diamantaris et al. [57] hypothesize that ad networks can facilitate the distribution of malicious JS code to harvest personal data, finding issues on WebView's app isolation and access control mechanisms to sensors like GPS and Camera. However, they were unable to demonstrate the actual abuse and dissemination of PII to the cloud. Tiwari et al. [145] studied the risk of device fingerprinting in WebViews by combining cookie values, UAs and browser metadata. Our analysis pipeline allows us detecting more advanced fingerprinting techniques and their actual use. The most relevant paper is the recent publication by Kuchhal et al. [96], which investigates the use of WebViews and CTs across 146.5K popular Android apps through static analysis, attributing them to their respective SDKs. Their dynamic analysis focuses on security aspects of in-app browsing in controlled lab settings. In fact, their methodology involves the artificial injection of popular desktop websites into WebViews across a subset of 1K apps to study WebViews' properties, rather than analyzing their organic behavior and risks in-the-wild. Our research complements this work by investigating the content that developers, SDKs and ATSes load in WebViews, providing real-world evidence of intrusive practices—including fingerprinting, permission piggybacking, PII leakage, and cookie sync-ing—that originate from the architectural shortcomings in hybrid in-app browsing and unvetted RTB processes beyond the control of end-users, thereby presenting a fundamentally different and less explored threat model.

Tooling. Several studies have proposed static and dynamic analysis tools to detect privacy and security issues in WebViews. *Bifocals*, by Chin and Wagner [28], is a method to detect security issues in WebView implementations without executing JS code, like cross-zone scripting. Rizzo et al. [128] introduced *BabelView*, a tool for simulating code injection attacks to assess the impact of malicious JS in WebViews. Bai et al. [17] proposed *BridgeTaint*, a taint analysis technique to monitor real-time data flows between JS and native code. Finally, Hu et al. [90] proposed a test generation tool for detecting vulnerabilities in WebViews resulting from cross-language interactions, applying it on 74 apps embedding WebViews.

9 Conclusions

This paper empirically demonstrated the widespread abuse of WebViews to track users and leak sensitive data for advertising and cross-platform tracking, and the shortcomings of existing privacy controls. Specifically, we showed how WebViews' architecture and SDK practices facilitate web-based tracking by arbitrary JS code distributed within WebViews through RTB processes by downgrading their default privacy settings. We uncovered significant unreported privacy-invasive practices like fingerprinting, cookie sync-ing, permission piggybacking, and ID bridging in WebViews. Our findings highlight critical gaps in existing app vetting processes like Google Play Protect and the need for a redesigned WebView permission model, platform privacy controls and sandboxing, stricter app store policies, and enhanced accountability for advertising and tracking networks to control the distribution of privacy harmful JS code.

Acknowledgments

This research was partially supported by the MICIU/AEI/10.13039/501100011033/ through the grants PARASITE (PID2022-143304OB-I00, EU ERDF), CYCAD (PID2022-140126OB-I00, EU ERDF), PRODIGY (PID2022-142290OB-I00, NextGenerationEU/ PRTR), and EMACS (RED2024-154240-T), and Canada's NSERC (RGPIN/04237-2018). J. M. Moreno is funded by the MICIU with a FPI Predoctoral Grant (PRE2020-094224). Prof. N. Vallina-Rodriguez was appointed as 2019 Ramon y Cajal fellow (RYC2020-030316-I), funded by by the MICIU/AEI/10.13039/501100011033/ and the EU ESF Investing in your Future. S. Matic was partially funded by the Atracción de Talento grant (Ref. 2020-T2/TIC-20184), funded by Madrid Regional Government. We would like to thank P. Wijesekera, A. Feal and A. Girish for early contributions in this paper, particularly demonstrating the feasibility of user tracking on mobile apps through WebViews using canvas fingerprinting. We donated all our bug bounty rewards to Doctors Without Borders. We used Chat-GPT for improving the clarity of the abstract and §7.

References

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proc. of ACM CCS*. ACM.

[2] Adivery. 2024. Adivery. https://www.adivery.com/en/. Accessed on November 30, 2024.

[3] Adobe. 2024. Adobe Creative SDK. https://creativesdk.github.io/ Accessed on November 23, 2024.

[4] Adobe. 2024. Adobe Experience Cloud. https://business.adobe.com/. Accessed on November 30, 2024.

[5] Adobe. 2024. Adobe Experience League. https://experienceleague.adobe.com/en/docs/analytics/implementation/vars/config-vars/trackingserver Accessed on November 23, 2024.

[6] Adobe. 2024. Adobe Experience Platform. Identity Service. https://business.adobe.com/products/experience-platform/identity-service.html Accessed on November 2, 2024.

[7] Adobe. 2024. *Identity: Adobe Developer.* https://developer.adobe.com/client-sdks/home/base/mobile-core/identity/ Accessed on November 28, 2024.

[8] Bigo Ads. 2024. Bigo Ads. https://www.adsbigo.com/ Accessed on November 23, 2024.

[9] Google Ads. 2024. Integrate the WebView API for Ads. https://developers.google.com/ad-manager/mobile-ads-sdk/android/browser/webview/api-for-ads. Accessed on February 13, 2025.

[10] Cheq AI. 2024. Ensighten. https://cheq.ai/ensighten/. Accessed on November 30, 2024.

[11] Akamai. 2024. Akamai Bot Manager. https://www.akamai.com/products/bot-manager Accessed on November 23, 2024.

[12] Amazon. 2024. Amazon Ads. https://advertising.amazon.com/. Accessed on November 30, 2024.

[13] Nampoina Andriamilanto, Tristan Allard, Gaëtan Le Guelvouit, and Alexandre Garel. 2021. A Large-scale Empirical Analysis of Browser Fingerprints Properties for Web Authentication. *ACM Trans. Web* (2021).

[14] Apple. 2024. Using alternative browser engines in the European Union. https://developer.apple.com/support/alternative-browser-engines/. Accessed on November 30, 2024.

[15] Applovin. 2024. Integration | AppLovin / MAX SDK. https://developers.applovin.com/en/android/overview/integration/ Accessed on July 8, 2024.

[16] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. In *Proc. of PETS*. PETS.

[17] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. 2019. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *Proc. of IEEE Transactions on Information Forensics and Security* (2019).

[18] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. 2016. Tracing Information Flows Between Ad Exchanges Using Retargeted Ads. In *Proc. of USENIX Security*. USENIX Association.

[19] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. 2024. Tabbed Out: Subverting the Android Custom Tab Security Model. In *Proc. of IEEE S&P*. IEEE.

[20] Philipp Beer, Lorenzo Veronese, Marco Squarcina, Martina Lindorfer, and TU Wien. 2022. The Bridge between Web Applications and Mobile Platforms is Still Broken. In *Workshop of Designing Security for the Web (SecWeb)*.

[21] Beeswax. 2024. *Beeswax.* https://www.beeswax.com/ Accessed on November 28, 2024.

[22] Bigo. 2024. BIGO Ads. https://www.adsbigo.com/. Accessed on November 30, 2024.

[23] Soumaya Boussaha, Lukas Hock, Miguel Bermejo, Ruben Cuevas Rumin, Angel Cuevas Rumin, David Klein, Martin Johns, Luca Compagna, Daniele Antonioli, and Thomas Barber. 2024. FP-tracer: Fine-grained Browser Fingerprinting Detection via Taint-tracking and Entropy-based Thresholds. *Proc. of PETS* (2024).

[24] Ryan Brown. 2024. Fanboy Adblock Homepage. https://fanboy.co.nz/ Accessed on September 11, 2024.

[25] ByteDance. 2024. ByteDance - Inspire Creativity, Enrich Life. https://www.bytedance.com/en/. Accessed on November 30, 2024.

[26] California State Legislature. 2018. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa. Accessed on November 30, 2024.

[27] Andrea Cardaci. 2024. cyrus-and/chrome-remote-interface: Chrome Debugging Protocol interface for Node.js. https://github.com/cyrus-and/chrome-remote-interface Accessed on June 11, 2024.

[28] Erika Chin and David Wagner. 2014. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Information Security Applications*. Springer International Publishing.

[29] Yoonjung Choi, Woonghee Lee, and Junbeom Hur. 2024. PhishinWebView: Analysis of Anti-Phishing Entities in Mobile Apps with WebView Targeted Phishing. In *Proc. of WWW*. ACM.

[30] Contanuity. 2024. *Contanuity.* https://contanuity.com/ Accessed on November 28, 2024.

[31] Joseph Cox. 2021. *Location Data Firm Got GPS Data From Apps Even When People Opted Out.* https://www.vice.com/en/article/huq-location-data-opt-out-no-consent/ Accessed on November 28, 2024.

[32] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. MAdFraud: investigating ad fraud in android applications. In *Proc. of ACM MobiSys*. ACM.

[33] Ha Dao, Johan Mazel, and Kensuke Fukuda. 2020. Characterizing CNAME Cloaking-Based Tracking on the Web. *IEEE/IFIP Network Traffic Measurement and Analysis Conference (TMA)* (2020).

[34] Darvin. 2021. DetectFrida: Detect Frida for Android. https://github.com/darvincisec/DetectFrida. Accessed on February 17, 2025.

[35] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proc. of ACM CCS*. ACM.

[36] DataDome. 2023. What is canvas fingerprinting? https://datadome.co/learning-center/canvas-fingerprinting/. Accessed on May 20, 2025.

[37] Demandscience. 2024. *Demandscience.* https://demandscience.com/ Accessed on November 28, 2024.

[38] The Trade Desk. 2024. The Trade Desk. https://www.thetradedesk.com/uk Accessed on November 23, 2024.

[39] Android Developers. 2024. Build web apps in WebView. https://developer.android.com/develop/ui/views/layout/webapps/webview Accessed on June 24, 2024.

[40] Android Developers. 2024. *CookieManager.* https://developer.android.com/reference/android/webkit/CookieManager Accessed on November 28, 2024.

[41] Android Developers. 2024. CookieManager.setAcceptThirdPartyCookies. https://developer.android.com/reference/android/webkit/CookieManager#setAcceptThirdPartyCookies(android.webkit.WebView,%20boolean) Accessed on September 1, 2024.

[42] Android Developers. 2024. In-app browsing using Embedded Web. https://developer.android.com/develop/ui/views/layout/webapps/in-app-browsing-embedded-web. Accessed on February 16, 2025.

[43] Android Developers. 2024. On-Device Protections. https://developers.google.com/android/play-protect/client-protections Accessed on October 8, 2024.

[44] Android Developers. 2024. PermissionRequest. https://developer.android.com/reference/android/webkit/PermissionRequest Accessed on June 10, 2024.

[45] Android Developers. 2024. Play Integrity API. https://developer.android.com/google/play/integrity. Accessed on February 9, 2025.

[46] Android Developers. 2024. Privacy indicators. https://source.android.com/docs/core/permissions/privacy-indicators Accessed on October 8, 2024.

[47] Android Developers. 2024. WebChromeClient. https://developer.android.com/reference/android/webkit/WebChromeClient Accessed on June 10, 2024.

[48] Android Developers. 2024. WebSettings. https://developer.android.com/reference/android/webkit/WebSettings Accessed on October 1, 2024.

[49] Android Developers. 2024. WebSettings.setDomStorageEnabled. https://developer.android.com/reference/android/webkit/WebSettings#setDomStorageEnabled(boolean) Accessed on November 17, 2024.

[50] Android Developers. 2024. WebView. https://developer.android.com/reference/android/webkit/WebView. Accessed on October 8, 2024.

[51] Android Developers. 2024. *WebView.addJavascriptInterface.* https://developer.android.com/reference/android/webkit/WebView#addJavascriptInterface(java.lang.Object,%20java.lang.String) Accessed on November 28, 2024.

[52] Android Developers. 2024. *WebView.evaluateJavascript.* https://developer.android.com/reference/android/webkit/WebView#evaluateJavascript(java.lang.String,%20android.webkit.ValueCallback%3Cjava.lang.String%3E) Accessed on November 28, 2024.

[53] Chromium Developers. 2024. Chrome custom tabs smooth the transition between apps and the web. https://blog.chromium.org/2015/09/chrome-custom-tabs-smooth-transition_2.html Accessed on October 1, 2024.

[54] Chromium Developers. 2024. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/ Accessed on August 23, 2024.

[55] Chromium Developers. 2024. *WebView Java Bridge.* https://chromium.googlesource.com/chromium/src/+/master/android_webview/docs/java-bridge.md Accessed on November 28, 2024.

[56] Mozilla Developers. 2024. Geckoview. https://mozilla.github.io/geckoview/ Accessed on July 8, 2024.

[57] Michalis Diamantaris, Francesco Marcantoni, Sotiris Ioannidis, and Jason Polakis. 2020. The Seven Deadly Sins of the HTML5 WebAPI: A Large-scale Study on the Risks of Mobile Sensor-based Attacks. *ACM TOPS* (2020).

[58] Michalis Diamantaris, Serafeim Moustakas, Lichao Sun, Sotiris Ioannidis, and Jason Polakis. 2021. This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration. In *Proc. of ACM CCS*. ACM.

[59] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. 2021. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. In *Proc. of PETS*. PETS.

[60] EasyList. 2024. EasyList. https://easylist.to/ Accessed on September 11, 2024.

[61] EasyPrivacy. 2024. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt Accessed on September 11, 2024.

[62] Federal Trade Commission Ed Felten. 2012. Does Hashing Make Data "Anonymous"? https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2012/04/does-hashing-make-data-anonymous. Accessed on May 20, 2025.

[63] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proc. of ACM CCS*. ACM.

[64] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. 2015. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proc. of WWW*. ACM.

[65] Esprima. 2024. Esprima. https://esprima.org/ Accessed on November 17, 2024.

[66] Estimote. 2024. Estimote UWB Tags. https://estimote.com/. Accessed on November 30, 2024.

[67] Exodus. 2024. Exodus Privacy. Trackers. https://reports.exodus-privacy.eu.org/en/trackers/ Accessed on November 2, 2024.

[68] Expedia. 2024. TExpedia. https://www.expedia.com/ Accessed on November 23, 2024.

[69] Álvaro Feal, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez. 2021. Don't accept candy from strangers: An analysis of third-party mobile SDKs. *Data Protection and Privacy: Data Protection and Artificial Intelligence* (2021).

[70] Julian Fietkau, Kashyap Thimmaraju, Felix Kybranz, Sebastian Neef, and Jean-Pierre Seifert. 2021. The Elephant in the Background: A Quantitative Approachto Empower Users Against Web Browser Fingerprinting. In *WPES '21: Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society*. ACM.

[71] FingerprintJS. 2024. *FingerprintJS*. https://github.com/fingerprintjs/fingerprintjs Accessed on November 28, 2024.

[72] FKIE-CAD. 2024. fkie-cad/friTap. https://github.com/fkie-cad/friTap Accessed on June 11, 2024.

[73] Forter. 2024. Forter: Identity Intelligence Platform. https://www.forter.com/ Accessed on November 23, 2024.

[74] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. 2020. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. In *Proc. of PETS*. PETS.

[75] Frida. 2024. Frida. https://frida.re/docs/home/. Accessed on February 17, 2025.

[76] Gadsme. 2024. Gadsme. https://www.gadsme.com/ Accessed on November 23, 2024.

[77] Garden City Games. 2024. *Westbound:Perils Ranch*. https://play.google.com/store/apps/details?id=com.kiwi.westbound Accessed on November 28, 2024.

[78] Aniketh Girish, Joel Reardon, Juan Tapiador, Srdjan Matic, and Narseo Vallina-Rodriguez. 2025. Your Signal, Their Data: An Empirical Privacy Analysis of Wireless-scanning SDKs in Android. In *Proc. of PETS*. PETS.

[79] Google. 2020. Helping publishers manage consent with Funding Choices. https://blog.google/products/admanager/helping-publishers-manage-consent-funding-choices/ Accessed on July 3, 2024.

[80] Google. 2024. Android - Google Mobile Services. https://www.android.com/gms/ Accessed on August 11, 2024.

[81] Google. 2024. *Google Analytics Help*. https://support.google.com/analytics/answer/1011829 Accessed on November 28, 2024.

[82] Google. 2024. Google Safe Browsing. https://safebrowsing.google.com Accessed on October 1, 2024.

[83] Google. 2024. Optimize Custom Tabs (Beta). https://developers.google.com/ad-manager/mobile-ads-sdk/android/browser/custom-tabs. Accessed on April 3, 2025.

[84] W3C Community Group. 2024. WebView: Usage Scenarios and Challenges. https://webview-cg.github.io/usage-and-challenges/ Accessed on June 24, 2024.

[85] GuardSquare. 2024. Java Obfuscator and Android App Optimizer | ProGuard. https://www.guardsquare.com/proguard. Accessed on February 11, 2025.

[86] Kaspar Hageman, Álvaro Feal, Julien Gamba, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, and Narseo Vallina-Rodriguez. 2023. Mixed signals: Analyzing software attribution challenges in the android ecosystem. *IEEE Transactions on Software Engineering* (2023).

[87] Catherine Han, Irwin Reyes, Álvaro Feal, Joel Reardon, Primal Wijesekera, Narseo Vallina-Rodriguez, Amit Elazari, Kenneth A Bamberger, and Serge Egelman. 2020. The price is (not) right: Comparing privacy in free and paid apps. In *Proc. of PETS*. PETS.

[88] Dominique Hazaël-Massieux. 2022. Making WebViews work for the Web. https://www.w3.org/blog/2022/making-webviews-work-for-the-web/ Accessed on June 24, 2024.

[89] Hemant. 2021. *What is gstatic.com used for? All you need to know!* https://www.thewindowsclub.com/what-is-gstatic-com-used-for-all-you-need-to-know Accessed on November 28, 2024.

[90] Jiajun Hu, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2023. ωTest: WebView-Oriented Testing for Android Applications. In *Proc. of ACM ISSTA*. ACM.

[91] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A tale of two cities: how WebView induces bugs to Android applications. In *Proc. of ACM ASE*. ACM.

[92] ID5. 2024. ID5. https://id5.io Accessed on November 2, 2024.

[93] Smart Instream. 2024. Smart - Instream SDK Documentation. https://documentation.smartadserver.com/instreamSDK/. Accessed on November 30, 2024.

[94] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *Proc. of IEEE S&P*. IEEE.

[95] Vegard Johnsen. 2020. Helping publishers manage consent with Funding Choices. https://blog.google/products/admanager/helping-publishers-manage-consent-funding-choices/. Accessed on November 30, 2024.

[96] Dhruv Kuchhal, Karthik Ramakrishnan, and Frank Li. 2024. Whatcha Lookin' At: Investigating Third-Party Web Content in Popular Android Apps. In *Proc. of ACM IMC*. ACM.

[97] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. 2022. DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting. In *Proc. of NDSS*.

[98] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser Fingerprinting: A Survey. *ACM Transactions on the Web* (2020).

[99] Kiho Lee, Chaejin Lim, Beomjin Jin, Taeyoung Kim, and Hyoungshick Kim. 2024. AdFlush: A Real-World Deployable Machine Learning Solution for Effective Advertisement and Web Tracker Prevention. In *Proc. of WWW*. ACM.

[100] LexisNexis. 2024. LexisNexis Risk Solutions. https://risk.lexisnexis.com/global/en. Accessed on February 16, 2025.

[101] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Prof. of ICSE*. IEEE.

[102] Lotame. 2024. Lotame Panorama Identity. https://www.lotame.com/panorama-identity/ Accessed on November 2, 2024.

[103] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2013. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Foundations and Practice of Security*. Springer Berlin Heidelberg.

[104] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Prof. of ICSE*. IEEE.

[105] Mappls. 2024. *Mappls*. https://www.mappls.com/ Accessed on November 28, 2024.

[106] Meta. 2024. Meta Ads. https://www.facebook.com/business/ads. Accessed on November 30, 2024.

[107] Meta. 2024. Meta Pixel. https://www.facebook.com/business/tools/meta-pixel Accessed on November 23, 2024.

[108] Foivos Michclinakis, Hossein Doroud, Abbas Razaghpanah, Andra Lutu, Narseo Vallina-Rodriguez, Phillipa Gill, and Joerg Widmer. 2018. The cloud that runs the mobile internet: A measurement study of mobile cloud services. In *Proc. of INFOCOM*. IEEE.

[109] Microsoft. 2024. Microsoft Clarity. https://clarity.microsoft.com/. Accessed on February 16, 2025.

[110] José Miguel Moreno. 2024. Fakeium - Lightweight Chromium Sandbox. https://github.com/josemmo/fakeium Accessed on October 22, 2024.

[111] Keaton Mowery and Hovav Shacham. 2012. Pixel Perfect: Fingerprinting Canvas in HTML5. (2012).

[112] Mozilla. 2024. Navigator: gpu property. https://developer.mozilla.org/en-US/docs/Web/API/Navigator/gpu Accessed on November 23, 2024.

[113] Mozilla. 2024. Support Web Bluetooth API in Android Webview. https://issues.chromium.org/issues/40703318 Accessed on November 23, 2024.

[114] Mozilla. 2024. Using HTTP Cookies - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies Accessed on November 14, 2024.

[115] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. In *Proc. of MoST*. ACM.

[116] Mozilla Developer Network. 2024. *Document Cookie*. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie Accessed on November 28, 2024.

[117] Mozilla Developer Network. 2024. Navigator - Web APIs | MDN. https://developer.mozilla.org/en-US/docs/Web/API/Navigator Accessed on July 4, 2024.

[118] Facundo Olano. 2024. facundoolano/google-play-scraper. https://github.com/facundoolano/google-play-scraper Accessed on June 27, 2024.

[119] Mandeep Pannu, Bob Gill, Robert Bird, Kai Yang, and Ben Farrel. 2016. Exploring proxy detection methodology. In *Proc. of ICCCF*. IEEE.

[120] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos Markatos. 2019. Cookie Synchronization: Everything You Always Wanted to Know But Were Afraid to Ask. In *Proc. of WWW*. ACM.

[121] Pardot. 2024. Pardot. https://pi.pardot.com/. Accessed on February 16, 2025.

[122] European Parliament and Council of the European Union. 2016. General Data Protection Regulation (GDPR). https://gdpr-info.eu/. Accessed on November 30, 2024.

[123] Lisandro Perez-Rey. 2024. *We Tracked Every Visitor to Epstein Island.* https://www.wired.com/video/watch/we-tracked-every-visitor-to-epstein-island Accessed on November 28, 2024.

[124] Amogh Pradeep, Alvaro Feal, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, and David Choffnes. 2023. Not your average app: A large-scale privacy analysis of android browsers. In *Proc. of PETS*. PETS.

[125] Philip Raschke and Axel Küpper. 2018. Uncovering Canvas Fingerprinting in Real-Time and Analyzing ist Usage for Web-Tracking. In *Workshops der INFORMATIK 2018 - Architekturen, Prozesse, Sicherheit und Nachhaltigkeit*. Köllen Druck+Verlag GmbH.

[126] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. 2018. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proc. of NDSS*.

[127] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. 2018. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. In *Proc. of PETS*. PETS.

[128] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. 2018. BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews. In *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing.

[129] Nate Schloss. 2022. Launching a new Chromium-based WebView for Android. https://engineering.fb.com/2022/09/30/android/launching-a-new-chromium-based-webview-for-android/ Accessed on July 8, 2024.

[130] Justin Sherman. 2023. The Location Data Market, Data Brokers, and Threats to Americans' Freedoms, Privacy, and Safety. (2023).

[131] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proc. of WWW*. ACM.

[132] Wei Song, Qingqing Huang, and Jeff Huang. 2020. Understanding JavaScript Vulnerabilities in Large Real-World Android Applications. *IEEE Transactions on Dependable and Secure Computing* (2020).

[133] Federal Trade Commission Staff in the Office of Technology. 2024. No, hashing still doesn't make your data anonymous. https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2024/07/no-hashing-still-doesnt-make-your-data-anonymous. Accessed on May 20, 2025.

[134] Taboola. 2024. Taboola. https://www.taboola.com/ Accessed on November 23, 2024.

[135] Junwei Tang, Ruixuan Li, Zhiqiang Xiong, Hongmu Han, and Xiwu Gu. 2021. Detecting Privacy Leaks in Android Hybrid Applications Based on Dynamic Taint Tracking. In *Proc. of EUC*. IEEE.

[136] TCPDump. 2024. TCPDUMP. https://www.tcpdump.org/ Accessed on June 11, 2024.

[137] Tealium. 2024. Tealium iQ. https://tealium.com/resource/datasheet/tealium-iq/ Accessed on November 23, 2024.

[138] Adguard Team. 2024. Adguard Filters. https://github.com/AdguardTeam/AdguardFilters Accessed on September 11, 2024.

[139] Daniel R. Thomas, Alastair R. Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. 2015. The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface. In *Security Protocols XXIII*. Springer International Publishing.

[140] Stuart A. Thompson and Charlie Warzel. 2019. *Twelve Million Phones, One Dataset, Zero Privacy.* https://www.nytimes.com/interactive/2019/12/19/opinion/location-tracking-cell-phone.html Accessed on November 28, 2024.

[141] Martin Thomson. 2022. *The SSLKEYLOGFILE Format for TLS*. Internet Draft draft-thomson-tls-keylogfile-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-thomson-tls-keylogfile-00 Accessed on June 11, 2024.

[142] Deyu Tian, Yun Ma, Aruna Balasubramanian, Yunxin Liu, Gang Huang, and Xuanzhe Liu. 2021. Characterizing embedded web browsing in mobile apps. *IEEE Transactions on Mobile Computing* (2021).

[143] Tinyproxy. 2024. Tinyproxy. https://tinyproxy.github.io/ Accessed on June 11, 2024.

[144] Abhishek Tiwari, Jyoti Prakash, Sascha Groß, and Christian Hammer. 2019. LUDroid: A Large Scale Analysis of Android – Web Hybridization. In *Proc. of IEEE SCAM*. IEEE.

[145] Abhishek Tiwari, Jyoti Prakash, Alimerdan Rahimov, and Christian Hammer. 2022. Our fingerprints don't fade from the Apps we touch: Fingerprinting the Android WebView. *ArXiv e-prints* (2022).

[146] TopOn. 2024. TopOn. https://www.toponad.com/en. Accessed on November 30, 2024.

[147] Connor Tumbleson. 2024. https://apktool.org/ Accessed on November 2, 2024.

[148] Digital Turbine. 2024. Digital Turbine — digitalturbine.com. https://www.digitalturbine.com/. Accessed on November 30, 2024.

[149] Unity3d. 2024. Unity Ads: Mobile Game Ad Network Platform & Analytics. https://unity.com/products/unity-ads. Accessed on November 30, 2024.

[150] Utiq. 2024. Utiq enables responsible digital marketing. https://utiq.com/. Accessed on February 16, 2025.

[151] Pelayo Vallina, Álvaro Feal, Julien Gamba, Narseo Vallina-Rodriguez, and Antonio Fernández Anta. 2019. Tales from the Porn: A Comprehensive Privacy Analysis of the Web Porn Ecosystem. In *Proc. of ACM IMC*. ACM.

[152] Vogo. 2024. Vogo. https://app.vogo.in/ Accessed on November 23, 2024.

[153] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. 2018. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *Proc. of ACM IMC*. ACM.

[154] Yandex. 2024. Yandex. https://ads.yandex.com/welcome. Accessed on November 30, 2024.

[155] Chuan Yue. 2016. Sensor-Based Mobile Web Fingerprinting and Cross-Site Input Inference Attacks. In *Proc. of SPW*. IEEE.

[156] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *Proc. of ISSTA*. ACM.

[157] Jiexin Zhang, Alastair R. Beresford, and Ian Sheret. 2019. SensorID: Sensor Calibration Fingerprinting for Smartphones. In *Proc. of IEEE S&P*. IEEE.

[158] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, Xiaofeng Wang, Long Lu, and Haixin Duan. 2018. An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications. In *Proc. of USENIX Security*. USENIX Association.

A JavaScript Web Permissions

Table 11 provides details on JavaScript permissions, their APIs and the corresponding Android Permissions. Table 12 provides details about the unsupported JS APIs in WebViews.

Table 11: JS permissions, their relevant JS APIs and the corresponding Android permissions.

JavaScript Permission	JavaScript API	Android Permission
navigator.permissions.query({ name: "geolocation" })	navigator.geolocation	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION
navigator.permissions.query({ name: "camera" })	navigator.mediaDevices.getUserMedia({ video: true })	CAMERA
navigator.permissions.query({ name: "microphone" })	navigator.mediaDevices.getUserMedia({ audio: true })	RECORD_AUDIO

Table 12: JS permissions and their relevant APIs that are unsupported in Android WebViews.

JavaScript Permission	JavaScript API
Background Synchronization API (background-sync)	navigator.serviceWorker
Compute Pressure API (compute-pressure)	PressureObserver
Local Font Access API (local-fonts)	window.queryLocalFonts
Notifications API (notifications)	Notification
Payment Handler API (payment-handler)	PaymentRequest
Push API (push)	PushManager
Sensor APIs (magnetometer, ambient-light-sensor)	Magnetometer, AmbientLightSensor
Storage Access API (storage-access, top-level-storage-access)	document.requestStorageAccess
Web Bluetooth API (bluetooth)	navigator.bluetooth

B Detecting PII Leaks

Table 13 provides the details about the PII types considered in our PII leakage analysis.

Table 13: Description of PII types considered.

PII Type	Description
AAID/GID	The Android Advertising ID (AAID) or Google Advertising ID (GID) is a globally unique and resettable ID for advertising provided by Google Play services. Can be reset through Android's limit tracking settings.
Android ID	A globally unique and resettable ID tied to the combination of app-signing keys, users, and devices. Can be reset through factory resets.
IMEI	The IMEI (International Mobile Equipment Identity) is a unique 15-digit serial number that uniquely identifying any device with a SIM card.
Email	Email addresses are closely related to user identities. Emails are often collected after hashing them with MD5, SHA1, or SHA256 hash functions.
Geolocation	Geographical coordinates, either lat/long pairs or geohashes.
MAC Address	Local Bluetooth Low-Energy (BLE) and Wifi MAC addresses.

C WebView Instrumentation

Table 14 lists the WebView methods instrumented with Frida and their purpose.

Table 14: List of WebView instrumented methods with Frida.

WebView Method	Purpose
`WebChromeClient.onPermissionRequest`	Tracks access to sensitive permissions.
`navigator.geolocation.getCurrentPosition`	Access to geolocation data.
`GeolocationPermissions.allow`	Access to geolocation data.
`HTMLCanvasElement.prototype.toDataURL`	Fingerprinting techniques.
`CanvasRenderingContext2D.prototype.fillText`	Fingerprinting techniques.
`WebView.evaluateJavascript`	Log JS API calls, including arguments passed and returned values.
`WebView.loadUrl`, `WebView.loadData`	Tracks URL loading events.
`WebView.addJavascriptInterface`	Tracks interactions between JS and Java code.
`CookieManager.setCookie`	Tracks cookie storage events.

D Hardcoded Hostnames Loading on WebViews

Table 15 presents instances of hardcoded hostnames along with their associated SDKs and the regions where they were observed.

E SDKs Across App Categories

Figure 5 shows the distribution of top-10 SDKs across app categories.

Table 15: Examples of hardcoded hostnames accessed by WebViews in various apps, along with their associated SDKs and the regions from which they were accessed.

Hostname	EU	IN	US	BR	JP	AU	# Apps	SDK	Organization
fundingchoicesmessages.google.com	✓	✓	✓	✓	✓	✓	1035	GMS SDK	Google
sf16-static.i18n-pglstatp.com				✓	✓	✓	36	ByteDance SDK	ByteDance
c.amazon-adsystem.com	✓	✓	✓	✓	✓	✓	24	Amazon Ads	Amazon
cdn.taboola.com	✓	✓	✓	✓	✓	✓	11	Taboola SDK	Taboola
trc.taboola.com	✓	✓	✓	✓	✓	✓	11	Taboola SDK	Taboola
yandex.ru	✓	✓	✓	✓	✓	✓	9	Yandex Ads SDK	Yandex
mraid.bigo.sg	✓		✓	✓	✓		8	Bigo Ads	Guangzhou HuanGuo
cdn2.inner-active.mobi	✓						3	Fyber	Fyber
www.dailymotion.com		✓					1	Dailymotion SDK	Dailymotion
api.adivery.com	✓						1	Adivery	Imazh Rayvar, Inc.
static-web.likeevideo.com			✓				1	Bigo Ads	Guangzhou HuanGuo

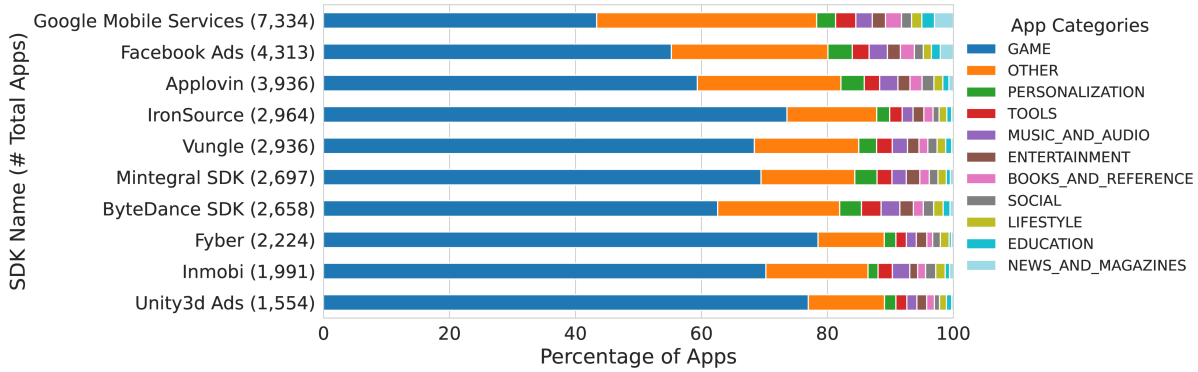

Figure 5: Market share of the top-10 SDKs per app category. Horizontal bars represent the fraction of apps per category.