Whatcha Lookin' At: Investigating Third-Party Web Content in Popular Android Apps

Dhruv Kuchhal
dkuchhal@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Karthik Ramakrishnan
rkarthik@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Frank Li
frankli@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Abstract

Over 65% of web traffic originates from mobile devices. However, much of this traffic is not from mobile web browsers but rather from mobile apps displaying web content. Android's WebView has been a common way for apps to display web content, but it entails security and privacy concerns, especially for third-party content. Custom Tabs (CTs) are a more recent and recommended alternative.

In this paper, we conduct a large-scale empirical study to examine if the top ~146.5K Android apps use WebViews and CTs in a manner that aligns with user security and privacy considerations. Our measurements reveal that most apps still use WebViews, particularly to display ads, with only ~20% using CTs. We also find that while some popular SDKs have migrated to CTs, others (e.g., financial services) benefiting from CT's properties have not yet done so. Through semi-manual analysis of the top 1K apps, we uncover a handful of apps that use WebViews to show arbitrary web content within their app while modifying the web content behavior. Ultimately, our work seeks to improve our understanding of how mobile apps interact with third-party web content and shed light on real-world security and privacy implications.

CCS Concepts

• **Security and privacy → Web application security**; • **Information systems → Browsers**.

Keywords

Android, WebView, CustomTabs, In-App Browser, Web, Privacy

ACM Reference Format:
Dhruv Kuchhal, Karthik Ramakrishnan, and Frank Li. 2024. Whatcha Lookin' At: Investigating Third-Party Web Content in Popular Android Apps. In *Proceedings of the 2024 ACM Internet Measurement Conference (IMC '24), November 4–6, 2024, Madrid, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3646547.3688405

1 Introduction

Mobile devices are ubiquitous in today's digital world. According to a recent estimate, more than 65% of web traffic originates from these devices [25]. This traffic includes users browsing the web via mobile browsers as well as various mobile apps that display web content. Examples include hybrid mobile apps that render a web page in full-screen mode within an app, in-app advertisements that show free content alongside ads, and web links that users follow from within an app.

Traditionally, such functionality has been achieved using the `android.webkit.WebView` class, or simply 'WebView'. WebView is a component of Android's View class that allows developers to embed web pages into their app's interface [12]. Android suggests using it only for trusted first-party content, as it gives the app more control over the webpage, such as executing Javascript (JS) code, intercepting requests, and enabling interaction between the webpage and the app's native code [11]. These properties support hybrid app experiences that can be useful in many contexts [88, 89]. However, nothing prevents an app from loading untrusted third-party content in a WebView, which exposes a large attack surface as previously identified (see Table 1).

On the other hand, Custom Tabs (CT) is a feature designed for securely and efficiently displaying third-party web content [60]. It has been available in Android devices since 2015 [1]. With CTs, developers can provide a customized browser experience within their app using Android browsers such as Chrome. It loads the webpage with access to the user's default browser cookies but does not manipulate the page itself [11, 28]. CT offers the latest browser experience, with a secure user interface (e.g., TLS lock icon), without requiring extra development effort. It has also been found to load pages up to twice as fast as WebView, as shown in Figure 7 in the Appendix [2]. We summarize the advantages of using CTs over WebViews for displaying third-party web content in Table 1. However, there is limited empirical research on whether developers leverage CT's advantages in the real world.

In this paper, we conduct large-scale measurements of popular Android apps to understand how they interact with web content. We analyze ~146.5K apps, each with over 100K users, and find ~55.7% apps using WebViews, ~20% using CTs, and ~15% using both. To infer use cases, we detect 125 popular SDKs using WebViews, 45 using CTs, and 34 using both. Our results indicate that WebViews were mainly used for advertising purposes, with 46 SDKs supporting this use-case in ~39K apps. In contrast, CTs were predominantly used for social media integration, with 6 SDKs enabling this use-case in nearly 24K apps. We combine our measurements with the extensive prior work on the insecurity of WebViews to propose evidence-based recommendations for both app and SDK developers. Overall, we find that there are still many widely-used SDKs that would benefit from migrating from WebViews to CTs.

In addition, we conduct a semi-manual analysis of the top 1K apps, identifying cases where apps override Android's default behavior of opening third-party URLs in a browser. We found 11 apps launched an In-App Browser (IAB), defined as any non-browser

IMC '24, November 4–6, 2024, Madrid, Spain

ACM ISBN 979-8-4007-0592-2/24/11
https://doi.org/10.1145/3646547.3688405

Comparison Attribute	WebViews	CustomTabs
Attack vectors from third-party web content or malicious app code	✗ Vulnerable due to bidirectional access between web and app contexts [42, 42, 61, 69, 72, 76, 82, 82, 95, 97, 97].	✓ Untrusted web content loads in browser context isolated from app context (no bidirectional access).
Phishing	✗ Vulnerable to cookie/credential stealing [54, 96].	✓ Supports passkeys (phishing resistant). ✓ Secure UI (e.g., TLS icon). ✗ Side-channel attacks exist [43].
Browser Fingerprinting	✗ WebViews are significantly more vulnerable [90, 91].	✓ Same default web browser used across multiple apps.
Page Load Time	✗ Slower, doesn't allow pre-initialization.	✓ Faster, allows pre-initialization [2].
User Experience	✗ User needs to authenticate repeatedly.	✓ User's active sessions are restored using existing browser cookies.

Table 1: Advantages of using CustomTabs over WebViews for displaying third-party web content.

Activity that can navigate to an arbitrary URL. Of these 11 apps, 10 apps implemented a WebView-based IAB, which we experimented with further. We uncover unique behaviors of these apps (with millions of downloads) modifying the web content, such as facilitating payments, ad injection, and crowdsourcing network measurement from end-user devices.

Ultimately, this work serves as a key step in better understanding how mobile apps interact with web content, particularly third-party content. Through our findings, we identify real-world security and privacy implications of mobile app behaviors and provide recommendations for mobile SDK and app developers.

2 Related Work

While WebView-based mobile apps offer a seamless and enriched user experience with relatively low development demands, they bear the risk of potential security and privacy concerns. Apps utilizing WebViews can inject Javascript (JS) code into webpages, enable bidirectional communication between native Java objects and webpages through JS Bridges, and control network requests. In 2011, Luo et al. introduced the threat model for WebViews, illustrating how they could serve as attack vectors for webpages within malicious apps or for malicious webpages embedded in benign apps [72]. Since then, numerous static analysis approaches have been proposed to detect data leaks and code injection attacks through JS Bridges [42, 82, 97]. Additionally, vulnerabilities and bugs introduced by interactions with the JS Bridge [61, 69, 76] and native app event handlers [95] have been identified. WebViews have also been manipulated to expand traditional web-based attacks, such as phishing [96] and browser fingerprinting [90, 91]. Moreover, malware has been discovered exploiting the capabilities of WebViews to evade detection [87]. As apps have become more complex, the threat landscape has also evolved. Recently, Zhang et al. examined how identity confusion in WebView-based apps can expose privileged APIs of one app component to an unrelated component [98, 102].

WebViews possess such expansive access rights primarily because they are intended to present trusted first-party content. However, no restriction prevents them from loading third-party content. Tuncay et al. suggested a policy-based strategy modeled after the Same-Origin-Policy, permitting granular control over web origins within WebViews [92]. Nevertheless, in practice, Zhang et al. identified instances of inter-party manipulation of web resources in WebViews [100]. The majority of these manipulations served benign purposes like customizing web services or enabling hybrid functionality. Yet, there were a handful of malicious instances aimed at stealing cookies or user credentials. It is worth noting here that Zhang et al.'s approach uses static analysis techniques to generate the URLs passed into the WebView. This means their analysis is constrained to URLs that already exist in the app in some form prior to its execution. Therefore, their findings do not extend to IABs implemented with WebViews. In other words, if a user clicks a third-party link seen in a social media app and that link opens in a WebView, this behavior is not included in their study because the URL was not originally present in the app.

In order to mitigate these risks, Android encourages employing CT for any non-first-party or untrusted web content [12]. Beer et al. recently proposed that the callback mechanism of CTs could be exploited as a cross-site oracle [43, 44], which highlights the need for developers to follow secure practices even when implementing CTs. That said, the attacks demonstrated on CTs so far are far less impactful than the ones discovered existing in the wild for WebViews. To date, there has been limited empirical investigation into the real-world adoption of CT. Zhang et al. studied IAB implementations in 25 popular apps from a usable security perspective [101]. Their findings indicate that IABs leveraging CT generally exhibit secure behaviors, whereas every single IAB utilizing WebViews presented at least one insecure behavior. Through our research, we offer an extensive characterization of WebView and CT adoption, drawing from a large-scale evaluation of popular Android apps. Furthermore, we contribute to a comprehensive understanding of privacy-invasive behavior within WebView-based IABs found in a smaller sample of widely used apps.

3 Method

In this section, we describe our method for assessing the usage of WebViews and Custom Tabs (CTs) within Android apps. First, we outline our systematic approach to a large-scale analysis using static techniques aimed at discerning patterns and deriving insights at an ecosystem level. Subsequently, we describe our usage of dynamic analysis methods, applied to a smaller, select subset of popular apps, which enables us to delve into the specifics of their behavior.

3.1 Static Analysis

Here, we delve into the details of our dataset and the methodology we use to apply static analysis techniques to characterize the usage of WebViews and CTs in apps. Figure 1 helps illustrates each of the steps in our static analysis pipeline.

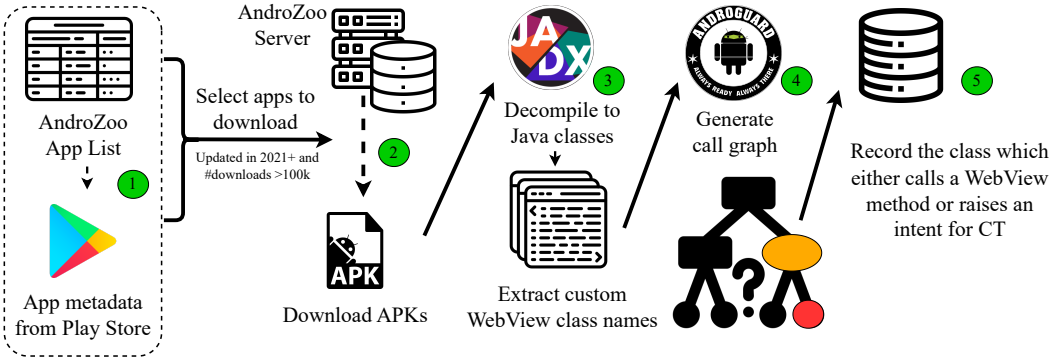


Figure 1: Our static analysis pipeline. (1) We fetch the list of Android apps available in AndroZoo [39] and for each app, we fetch its metadata from Google Play Store. (2) For apps that have been updated at least once in the last two years, and have been downloaded more than 100K times, we download their most recent APK from AndroZoo. (3) Next, we decompile each APK (using JADX [57]) and extract the names of classes that extend the WebView class. (4) Finally, we generate call graphs for each APK (using Androguard [10]) and (5) record the instances where a WebView method is called or a CT is initialized.

3.1.1 Dataset. We use Androzoo, a widely used repository of Android apps, which periodically fetches Android Package Kit (APK) files from multiple app stores, including the Google Play Store [39]. Using the January 13, 2023 snapshot of Androzoo, we fetch the list of ~6.5M apps that have appeared in the Google Play Store. Since our aim was to gain insights about popular apps that are currently being used by a large population of users, we collected app metadata from the Google Play Store, which included information such as the number of times an app was installed and the category of the app, among other information [20]. Next, after filtering for apps that had at least 100K downloads and were being actively maintained, i.e., had been updated at least once after January 1, 2021, we narrowed it down to a set of 146.8K apps. For each of these 146.8K apps, we downloaded their most recent APK from AndroZoo for further analysis.

Dataset	No. of apps
Play Store apps in Androzoo	6,507,222
Apps found on Play Store	2,454,488
Apps with 100k+ downloads	198,324
Apps with 100k+ downloads and updated after 2021	146,800
Apps successfully analyzed	**146,558**

Table 2: Statistics for apps that we statically analyze.

3.1.2 Preprocessing APKs. For apps to display web content using WebViews, they need to instantiate objects of the Android class `android.webkit.WebView` as part of their activity layout [12]. The `WebView` class offers extensibility in terms of customizing the UI and other advanced configuration options, so it is only natural that developers extend the `WebView` class to build their own custom WebView experiences. To detect such custom WebView class implementations, our approach is two-fold. Firstly, we decompile the downloaded APK files into Java source code using JADX, a state-of-the-art tool with the lowest failure rate compared to other tools [57, 74]. Secondly, for each class file in the source code that imports `android.webkit.WebView`, we use an open-source parser to parse the Java source code [80] and extract the names of classes that extend the WebView class. With this information, we can accurately pinpoint the entities in the source code that instantiate and interact with WebViews, as we will describe in detail shortly.

The behavior of a program, precisely its control flow relating to the sequence and dependencies of function calls, can be effectively comprehended through static analysis techniques like call graphs. As a crucial part of our preparatory process for analysis, we generated call graphs for all 146.8K APKs we had procured. This was achieved using Androguard [10], an open-source reverse engineering tool specifically designed to analyze Android apps, and a component of well-established security and privacy toolkits like Cuckoo Sandbox and Exodus, respectively. Our preference for Androguard, over other tools such as FlowDroid [40], was informed by several factors. FlowDroid, while widely cited in the literature for smaller datasets, is primarily designed for taint-tracking and leads to comparatively higher failure rates and increased resource overheads [84]. In contrast, Androguard enabled us to generate call graphs for 99.8% of the apps in our dataset. The remaining 242 APKs were discovered to be broken. A summary of our dataset is available in Table 2.

3.1.3 Measuring WebView and CT Usage. An Android app is comprised of various components, specifically Activity, Service, Content Provider, and Broadcast Receiver. Any of these elements can serve as the initial point of interaction or "entry point" which leads to a WebView or CT [49]. Multiple entry points exist within a single activity, facilitated by lifecycle methods like `onCreate()` or callbacks tied to system or GUI events, each of which could follow a different control flow. Unlike other apps built with Java, an Android app lacks a 'main' function, which could be considered the primary entry point. Therefore, in order to exhaustively identify the usage of WebViews and CTs in an app, we traversed the app's entire call graph via all entry points, recording every call to WebViews and CTs. For WebView-related activity, we recorded the names of the WebView class methods that were called, along with the names of

the classes in which the respective call was made. For CT-related activity, we similarly recorded calls made for the `CustomTabsIntent` intent class of `androidx.browser.customtabs` [16].

Next, to filter out app activities that are likely to host first-party web content, we identified activities that can handle deep links to app content [15] and excluded them from further consideration. Specifically, we used the app manifest to check if an activity has the flag `exported` set to `true`, and contains an intent-filter of the category `android.intent.category.BROWSABLE`, which accepts data with a scheme of either `http` or `https`.

3.1.4 Identifying and Labelling SDKs. To understand the reasons why an app might utilize a WebView or CT, we explored whether the Java package responsible for invoking the methods that populate content into these components is part of an SDK with a defined function. In the case of WebViews, we searched for calls to one of the following methods: `loadUrl`, `loadData`, or `loadDataWithBaseURL`. For CTs, we searched for method calls to `launchUrl`. These methods would need to be invoked to populate content. Since we had recorded which classes called these methods, we were able to extract the package names from those classes, assuming that package names adhere to the proper Java conventions [26]. In aggregate, we identified 141 packages, each of which was used by more than 100 apps. We then used data from the Google Play SDK Index [21] and supplementary Google Search to label these packages into SDK categories if they were known to be part of certain SDKs. Excluding Google's Android SDK (`com.google.android`) – due to its multiple essential functions – we successfully categorized 126 out of 140 packages. The remaining 14 packages either had obfuscated labels (4), or they could not be associated with any known SDK (10).

We summarize statistics for the various types of SDKs found using WebViews and CTs in Table 3. We also illustrate examples of the most popular SDKs we identified as using WebViews and CTs in Tables 4 and 5 respectively.

Type of SDK	No. of SDKs		
	Use WebViews	Use CT	Use both
Advertising	46	3	3
Payments	15	6	5
Development Tools	11	7	5
Engagement	12	0	0
Social	10	6	4
Authentication	7	10	6
Unknown	10	4	4
Hybrid Functionality	6	7	5
Utility	4	2	2
User Support	4	0	0
Total	**125**	**45**	**34**

Table 3: Statistics for use of WebViews and CTs in SDKs.

Type of SDK	Total #apps	SDK Name	#apps
Advertising	39,163	AppLovin	27,397
		ironSource	16,326
		ByteDance	13,080
		InMobi	10,066
		Digital Turbine	8,654
Engagement	21,040	Open Measurement	11,333
		SafeDK	7,427
		Airship	652
		Branch	514
Development Tools	7,020	Flutter	5,568
		InAppWebView	1,868
		Corona	449
		AdvancedWebView	386
Payments	3,212	Stripe	1,171
		RazorPay	484
		PayTM	400
User Support	1,692	Zendesk	1,000
		Freshchat	438
		LicensesDialog	129
Social	1,686	VK	456
		NAVER	406
		Kakao	347
Utility	362	NAVER Maps	130
		Barcode Scanner	129
		Ticketmaster	64
Authentication	342	Gigya	120
		NAVER	90
		Amazon Identity	37
Hybrid Functionality	256	Baby Panda World	194
		SoftCraft	15
		Cube Storm	14

Table 4: Popular SDKs which use WebViews.

Type of SDK	Total #apps	SDK Name	#apps
Social	23,807	Facebook	23,234
		NAVER	157
		Kakao	54
Authentication	7,802	Google Firebase	7,565
		NAVER	81
		AdobePass	55
Advertising	1,953	HyprMX	1,257
		Linkvertise	383
		Taboola	317
Payments	208	Juspay	77
		Ticketmaster Checkout	47
Development Tools	172	android-customtabs	53
		GoodBarber	48
		Mobiroller	27
Hybrid Functionality	87	Cube Storm	14
		Scripps News	13
Utility	71	Ticketmaster	55
		MyChart	16

Table 5: Popular SDKs which use CTs.

3.1.5 Limitations. Our approach enables us to conduct a large-scale study of the usage of WebViews and CTs. However, it is worth noting some important limitations to our methodology.

- A recognized limitation of static analysis methods is their tendency to produce false positives. False positives may occur for various reasons. For instance, in our case, specific logic within an application might prevent initiating a code block that utilizes WebViews or Custom Tabs (CTs), often due to user interactions. Despite this, static analysis has been effectively employed in previous studies for large-scale measurements [45, 70, 94].
- Our investigation targets the identification of method calls based on their characterized behavior as detailed in Android's documentation. Consequently, our method may fall short in detecting obfuscated method calls exhibiting similar behaviors. It is worth noting, however, that obfuscation is relatively uncommon in apps available on the Google Play Store [52].
- Our approach assumes that a third-party SDK would serve third-party web content, however that might not necessarily always be true. For precise identification of the web content loaded, static modeling of control or data flow instigated by user interactions with the app would be required. However, this has been identified as a limitation to static analysis methods [45, 94].

3.2 Semi-Manual Dynamic Analysis

Building upon the insights garnered from our large-scale static analysis of WebView and CT utilization in apps, we meticulously carry out a semi-manual analysis of the top 1K apps with the goal of understanding how apps implement WebView-based IABs to handle third-party web links. In this case, we specifically consider WebViews, which display third-party web content a user navigates to by clicking on an external link, rather than web content belonging to a third-party SDK, as we did with large-scale static analysis.

3.2.1 Dataset. From the ~146.8K apps we had selected in Section 3.1.1, we further select 1K apps with the highest number of downloads. We programmatically download each app from the Google Play Store and install it on a Pixel device. Dummy accounts are manually created where necessary to access content within the app. Next, we manually investigate areas of the app that potentially contain user-generated web links. Intuitively, we focus on sections that display user-generated content, for example, social media feeds with user posts and comments, direct message or chat interfaces, and profile biographies where users may link affiliated web pages. This methodology echoes that of Zhang et al.'s approach to studying the security design of IABs [101]. Our analysis uncovers 38 apps, incidentally belonging solely to the social and communication categories, where users can post links. We find that 905 of the 1K apps we analyze do not contain user-generated content. These are predominantly utility apps such as media players, entertainment, stock, and gaming apps. Notably, nine apps are browsers themselves, and the remaining 48 apps are unclassifiable due to various factors, as presented in Table 6.

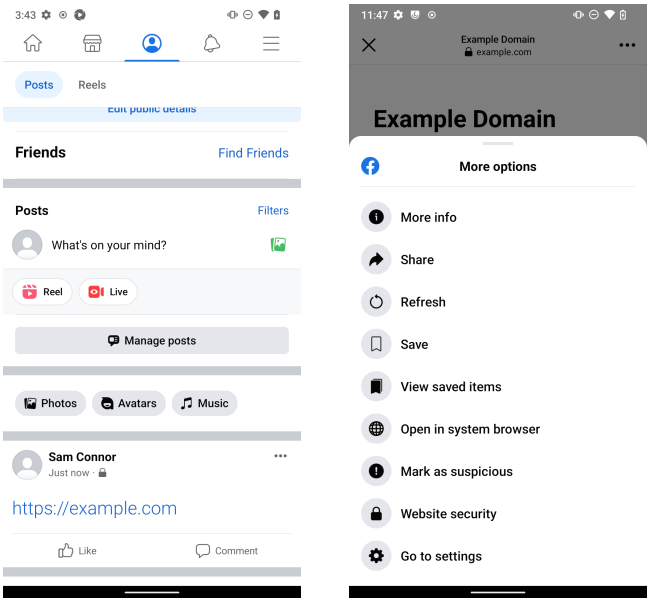
For the 38 apps where users can post links, we manually submit a link to `https://example.com` and follow it. We observe that approximately 71% (27 of 38) of the apps open the links in a browser, which is the intended and default behavior for web links in apps, thus these apps are not studied further. Interestingly, we discovered around 26% (10 of 38) of the apps open this third-party link in a WebView-based IAB. For instance, if a Facebook user clicks on a URL they see on their feed, it opens inside a WebView-based IAB. Discord is the only app that opens the link in a CT. For reference, Figure 2 differentiates the UI flow between WebViews and CTs, and Table 6 summarizes these statistics.

Classification of apps	#apps
Users can post links.	38
Link opens in browser.	27
Link opens in a WebView.	10
Link opens in CT.	1
Users can not post links.	905
Browser Apps.	9
Could not classify app.	48
Required a phone number.	24
App incompatibility error.	22
Required paid account.	2

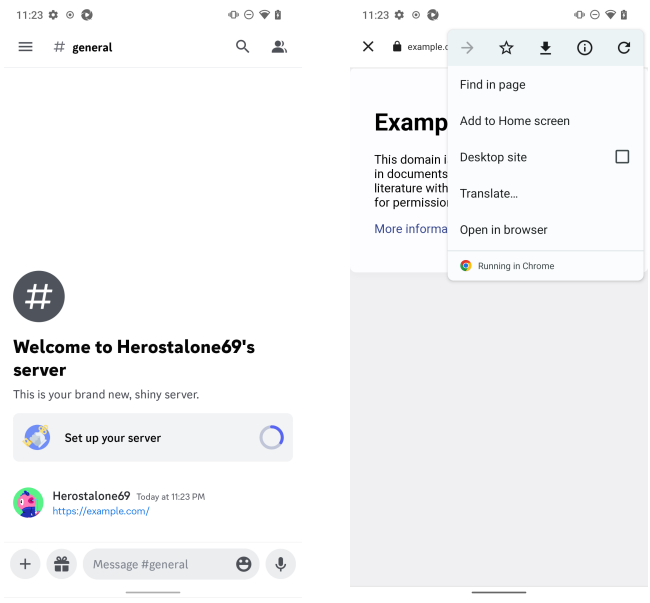
Table 6: Statistics for our manual classification of hyperlink clicking behavior in the top 1K Android apps on Play Store.

3.2.2 Measurement Setup. We previously established that CTs are the most efficient and secure approach to implementing IABs. To investigate the motives of apps using WebViews to implement IABs, we navigate each of the 10 WebView-based IABs to a controlled web page hosted on our server and record the following measurements:

- *App-WebView Interactions*: Using Frida [19], a widely-used dynamic instrumentation tool [38, 78, 99], we dynamically override all methods of `android.webkit.WebView` at run-time in order to record the WebView APIs used by the app, along with the arguments passed. Specifically, when an app interacts with WebView beyond mere loading of the URL, this information provides detailed insight for further analysis.
- *JS Code Injection*: An app can inject JS code into a WebView primarily via methods `evaluateJavascript` and `loadUrl`. `evaluateJavascript` allows executing JS code in the WebView and retrieving the result asynchronously, while `loadUrl` can be used to execute JS code once the page has finished loading by prepending `javascript:` (as the scheme) to the code. In cases where such injection is detected, we further record:
  - *Web API usage*: We host the HTML5 test page (composed of common HTML elements) developed by Bracco et al. [46] as our controlled web page. The only JS script used on the page is meant to override all methods of all Web APIs as listed in MDN Web Docs [75] and submit the intercepted requests with parameters back to our server [64]. Thus, when the injected JS code uses any Web APIs, our server records it.
  - *Network Logs*: We use a rooted Pixel 3 mobile device running LineageOS 19 [23] for our measurements. LineageOS is a custom `userdebug` image of Android, which enables us to record the network logs directly from Chrome's network



(a) WebView-based IAB in Facebook.



(b) CT-based IAB in Discord.

Figure 2: Screenshots showing WebView and CT-based IAB implementations in Facebook and Discord, respectively.

stack [59]. As opposed to capturing device-wide network traffic via `mitmproxy`, we are able to collect detailed network logs for each web page visit via a specific WebView instance. Previous work has found such logs reliable for website-specific network measurements [67].

With this comprehensive set of measurements, we can gain a nuanced understanding of the user's privacy posture when employing WebView-based IABs. Finally, we use our measurement setup to investigate if the behavior of WebViews varies based on the website a user visits. To this end, we systematically crawl the landing pages of 100 randomly selected top sites, taken from Google Chrome's top 1K most visited origins in the snapshot of February 2023 (CrUX) [83], using the ten different WebViews previously identified. In addition to the WebViews found on apps, we also crawl each site using the Android's System WebView Shell App which gives us a baseline for the network requests expected to be made from a WebView without any injections [32]. These crawls were executed with a Pixel 3 device, which was connected to an academic ISP via WiFi. For each app, a distinct crawler was crafted using Android Debug Bridge (ADB) commands to traverse the unique user interface of the app. During each website visit, the script utilizes Android Debug Bridge (ADB) commands to (i) launch the app, (ii) navigate to the intended activity by simulating screen taps at predetermined coordinates, (iii) insert the desired crawl URL, (iv) tap on the URL to instigate a visit within a WebView, and (v) swipe upwards to scroll through to the end of the webpage. Following a 20-second wait to allow the page to load fully, we gather the device's network log. To ready the system for the next crawl, we also purge the logs on the device, terminate the app, and wait for 1 minute.

3.2.3 Limitations. Our approach enables us to conduct comprehensive measurements for WebView-based IABs in real-world Android apps. However, it is worth noting some important limitations to our methodology.

- Our methodology inherently necessitates significant manual effort, thus constraining our scope to a limited number of apps. We focus on the top 1K apps due to their wide usage and significant influence on the user base. In all 10 apps where we discovered WebView-based IABs, the creation of dummy accounts was a prerequisite to accessing the app content. Automating account creation is challenging to fully automate, given the broad diversity in app UI designs and authentication workflows. For the same reason, Android's Monkey [50] - despite its efficacy in other studies - may also not be effective in our context.
- During our manual analysis, we observed only 1 WebView-based IAB per app. However, it is possible that there exist other WebView-based IABs, embedded deeper within the app or activated by specific user behavior we did not replicate. While our observations might not fully encapsulate all potential privacy-invasive actions apps could be executing in the wild, they contribute an initial, meaningful characterization of why apps may still employ WebView-based IABs.
- Our crawl, although automated, was limited to the landing pages of the top 100 sites. The restriction was due to rate-limiting we experienced with the Facebook app, which restricted our account twice during the measurements, necessitating manual intervention and the creation of new dummy accounts. WebViews might exhibit different behaviors on other web pages that we did not crawl.
- During our crawls, we attempt to load the entire page by scrolling down through the end of the page and allowing a 20-second pause for resources to load. However, we did not emulate any additional user behaviors. WebViews could exhibit additional

behavior when the user interacts with the page (e.g., form filling). Our measurements are a first step towards understanding the dynamics of WebView-based IABs in Android apps.
- The `addJavascriptInterface` method of the `WebView` class facilitates an interface between the app native code and the JS Virtual Machine inside the WebView. Our measurements successfully record instances when such a bridge is exposed. However, our methodology lacks the ability to monitor the communication between the JS VM and the Java methods.

We would be happy to share the code and data used to derive the results in this paper privately with researchers interested in reproducing and extending our work.

4 Findings

In this section, we examine the data from our measurements described in Section 3, aiming to understand the diverse ways that web content is embedded in Android apps. We mainly investigate the different use cases for which apps use WebViews and CTs. Considering the security, privacy, and performance issues related to WebViews, we hypothesize that WebViews should only be used in situations where CTs are insufficient – specifically, in cases where the app needs to interact with the web content, such as in a hybrid app. Furthermore, we explore the use cases where CTs could offer better security and privacy protection than WebViews. We also identify the apps that use WebViews to display IABs and analyze the reasons and implications of this choice, particularly if it exposes users to any security or privacy risks.

4.1 App Use of WebViews vs. CTs

In Section 3.1, we describe our method for analyzing the usage of WebViews and CTs in the 146.5K most popular apps using static analysis techniques. Our analysis reveals that 55.7% of the apps use WebViews, 20% incorporate CTs, and 15% make use of both. Cumulatively, we detected 125 SDKs using WebViews, 45 SDKs using CTs, and 34 SDKs using both. The popular SDKs we identified are used in approximately 67% of the apps integrating WebViews, 96% of the apps adopting CTs, and 76% of the apps utilizing both. These statistics are summarized in Table 7. It is worth noting that an app could be making use of more than one SDK, and it could also have multiple instances of WebViews or CTs, some of which might not be facilitated through the top SDKs. We take a holistic approach to examining the use cases of WebViews and CTs through prevalent SDKs as it allows us to identify patterns and extract insights.

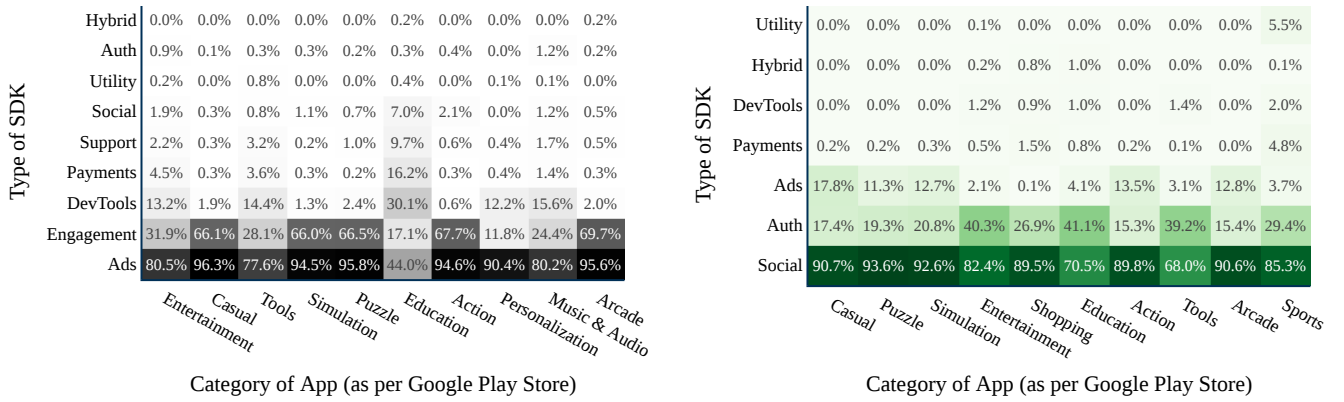
We analyzed the use cases of WebViews and CTs in different app categories, as shown in Figure 3. We found that WebViews were mainly used for advertising purposes, with 46 SDKs supporting this use case in about 39K apps. In contrast, CTs were predominantly used for social media integration, with 6 SDKs enabling this use case in nearly 23.8K apps. We observed that gaming apps (Puzzle, Simulation, Action, and Arcade) frequently used CT-based social media SDKs, while education apps used a lower proportion of WebView-based Ad SDKs (44%) and a higher proportion of WebView-based Payment SDKs (~16.2%). We summarize the statistics for use of WebViews and CTs in SDKs in Table 3, and statistics for the top SDKs using WebViews and CTs in each categories in Table 4 and Table 5 respectively. Next, we will comprehensively discuss each individual SDK use case to acquire an exhaustive understanding.

Dataset	Total #apps	#apps using top SDKs
Apps using WebViews	81,720	54,833
loadUrl	77,930	50,984
addJavascriptInterface	36,899	23,087
loadDataWithBaseURL	35,680	27,474
evaluateJavascript	26,891	18,716
removeJavascriptInterface	19,684	15,034
loadData	8,275	918
postUrl	5,028	2,678
Apps using CTs	29,130	27,891
Apps using both WebViews and CTs	21,938	16,810

Table 7: Statistics of the apps using WebViews and CTs. For the apps using WebViews, we consider the usage of WebView API methods that can be used to load and modify (by injecting JS) the web content requested by the user.

4.1.1 Advertising (Ads). In-app Ads are a major source of revenue for free Android apps and are essential to the mobile app economy. These ads are usually displayed in banner or interstitial formats, showcasing rich media, such as HTML5 content, or interactive ads, which are often streamed directly from the internet [37]. Google's Mobile Ads SDK empowers app developers to monetize their app content by showing Google's third-party ads adjacent to it. At the time of our study in 2023, the SDK primarily utilized WebViews, while supporting CustomTabs minimally [22]. However, starting March 2024, the SDK has released support for monetization and anti-fraud protections when implemented via CTs (in Beta) [36], which indicates a positive shift towards increased CT adoption. Similar to Google Ads, there are several other ad networks implementing mobile ad SDKs, many of which have been studied by previous work [41, 47, 71, 77, 86].

Notwithstanding the benign nature of most ads served by these networks, numerous instances exist of malicious ads exploiting WebViews' capabilities. Liu et al. discovered instances where popular mobile ad networks manipulated WebViews to present deceptive click ads, execute malevolent JavaScript code on the device (such as cryptojacking), redirect users to harmful websites, and deceive users into downloading malicious apps [71]. Browsers, which CTs can utilize, are constantly evolving to incorporate robust defenses against such web threats. For instance, features such as Google's Safe Browsing offer real-time threat intelligence [58], which could potentially decrease users' exposure to harmful content. Since WebViews are customizable, Ad SDKs can choose to disable SafeBrowsing, whereas Ad SDKs using CTs would be subject to SafeBrowsing unless the user has explicitly disabled it in their browser. Research conducted by Son et al. has highlighted instances where malicious mobile ads have exploited the access granted by WebViews to extract sensitive information from the device's external storage, and in certain cases, even read the user's local files [86]. Furthermore, there have been proposals to enforce stricter separation of privileges between the ad SDK environment and the host app [77, 85].



(a) For apps which use WebViews via SDKs.

(b) For apps which use CTs via SDKs.

Figure 3: Distribution of use-cases (SDKs) per app category for the top-10 app categories using WebViews and CTs respectively.

Android's CTs practically implements those goals by securely segregating the execution context of the mobile ad from the host app.

Nevertheless, in our analyses, we found that the usage of CTs by Ad SDKs is minimal. Only 3 Ad SDKs were observed using CTs in ~2K apps, all of which also utilized WebViews. As seen in Figure 4, more than 45% of apps displaying Ads via WebViews expose a JS Bridge to the WebView using the `addJavascriptInterface` method, while more than 30% apps inject JS via the `evaluateJavascript` method. This exposes an attack surface that might not always be necessary to the intended functionality. As shown in Table 4, the prevalent practice among Ad SDKs is still to use WebViews, with 46 SDKs implemented in ~40K apps.

> **Takeaway:** Ad SDKs should consider implementing CTs where possible, and app developers should carefully weigh the functionality offered by WebView-based Ad SDKs against the attack surface they expose. Our measurements show that ad networks primarily rely on WebViews, which have previously been shown to be exploited by malicious ads, and CTs offer a promising replacement.

4.1.2 Engagement. App developers frequently employ engagement measurement or analytics SDKs to determine which content is most engaging, intriguing, or valuable to the user. Our data indicate that ~21K apps use a WebView via such an SDK. The Open Measurement (OM) SDK is the most prevalently used, found in an estimated ~11.3K apps. The OM SDK facilitates third-party access to engagement measurement data. Its primary use is to measure and verify ad performance [68]. This involves checking whether an ad was visible, the duration of its visibility, its compliance with the set performance metrics, and whether it engaged in legitimate interactions as opposed to fraudulent views. Our manual analysis verifies that the OM SDK is predominantly used to support the ad networks AdColony and Ogury, which are found in ~10.6K and ~1.4K apps respectively, for performance measurement. We did not detect Engagement or Measurement SDKs using CTs. Even though CTs natively measure similar user engagement signals [51], we regard the use of SDKs to measure user engagement as a legitimate use case for WebViews as they are currently more capable of providing custom measurements.

4.1.3 Development Tools. We label SDKs which provide a set of software development tools and libraries to help streamline the process of creating apps as 'Development Tools' (Dev Tools). These SDKs enable developers to cost-efficiently build high-quality apps. For instance, Google's Flutter is a cross-platform framework that enables developers to build apps compatible with both Android and iOS using a single codebase [17].

In our measurements, ~7K apps utilize WebView components provided by such Dev Tool SDKs. Specifically, we found ~5.5K apps leveraging `url_launcher`, a Flutter plugin used by apps to open a URL in a WebView [33]. Interestingly, about ~2K apps make use of 'InAppWebView', an actively maintained third-party Flutter plugin used to embed WebView-based IABs in apps [18]. This plugin, which has earned over 2.6K stars on GitHub, boasts functionality for visiting third-party web content in an IAB, while also offering the ability to inject JS code into the WebView-based IAB. We found a limited number of 172 apps using CTs via such SDKs. Notably among them, 'android-customtabs' is an SDK developed to default apps to WebViews if no browser on the user's device supports CTs [55]. As the adoption of CTs increases, we believe that support for Dev Tools SDKs using CTs will also grow. Our discussion of the security and privacy implications of Dev Tools SDKs utilizing either WebViews or CTs is limited, as their actual use case depends on the app developers incorporating them – they alone specifically do not constitute a use case.

4.1.4 Payments. Payment processing SDKs offer APIs, UI components, and other tools that make it easier to integrate payment processing into an app. They can be used by e-commerce apps, either directly (i.e., selling products on a store) or indirectly (i.e., displaying ads with buy buttons next to other content) – or by freemium apps that generate revenue from in-app purchases of premium features. Mahmud et al.'s security analysis of Android Payment SDKs uncovered 26 SDKs that used WebViews to enable payments functionality such as checkout [73]. They reported that none of the SDKs configured WebViews securely enough to comply

with the OWASP Mobile Application Security Verification Standard [56]. In particular, 20 of those SDKs breached the PLAT4 platform requirement by exposing sensitive payment data (e.g., credit card numbers) from the WebView to the app. As seen in Figure 4, our measurements too indicate that 48.5% apps using WebViews for payments expose a JS Bridge to the WebView. Plaid Link, a financial services SDK that enables app users to connect their bank accounts, addressed these issues by switching to CTs [5]. They emphasized that besides the interception of sensitive credentials, the absence of security UI built into browsers (e.g., SSL verification icon) could allow a malicious app phishing the the user's credentials.

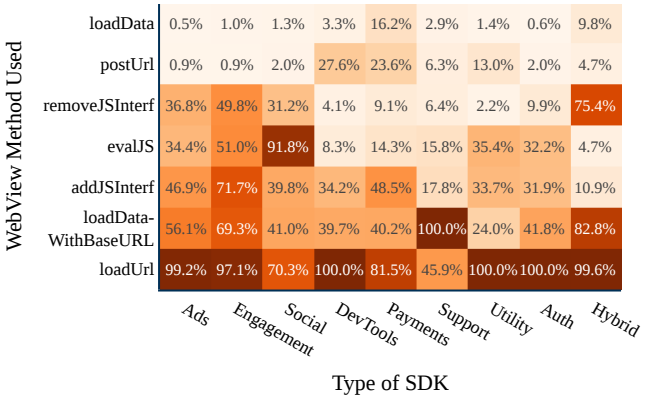


Figure 4: Heatmap of the distribution of WebView API method calls made by apps via SDKs.

However, our own analysis showed that WebViews remain the prevailing choice for Payment SDKs. We identified their use in ~3.2K apps, facilitated by 15 distinct Payment SDKs. The most prominent SDKs that utilize WebViews include Stripe (~1.1K apps), RazorPay (484 apps), and PayTM (400 apps), all of which do so to facilitate checkouts. CTs, on the other hand, were found in 6 SDKs and were implemented by a mere 208 apps. Juspay (77 apps) and Ticketmaster Checkout (47 apps) - SDKs utilizing CTs, both also facilitate checkouts. Juspay's documentation outlines their use of CTs to enable payments via Amazon Pay [63]. Intriguingly, we observed that 5 out of the 6 SDKs supporting CTs also support WebViews. We hypothesize that this is because these SDKs default to WebViews when a browser with CT support is not available.

Takeaway: Payment providers should use CTs instead of WebViews for integrating payment processing in their SDKs, as CTs provide more secure and user-friendly payment experiences. We find that most payment SDKs still use WebViews, which can leak sensitive payment data or expose users to phishing attacks.

4.1.5 User Support. User support SDKs allow developers to offer customer service options within their app, rather than directing users to call or email. We found that ~1.7K apps used WebViews to enable in-app customer service. Of these, ~85% relied on Zendesk and Freshcat SDKs to provide live chat support. This is a suitable use case for WebViews, as the customer service web component may need to access information from the app. For example, if a user requests support for an order they made, the customer service web component may need to retrieve order status details from the app. This is supported by our analysis of WebView APIs by User Support SDKs. In particular, as seen in Figure 4, all apps using WebViews for user support load local data into the WebView using the `loadDataWithBaseURL` method, while only 45.9% apps use `loadUrl`. Not surprisingly, we did not find any apps that used CTs for in-app customer service.

4.1.6 Social. Social Media SDKs enable developers to access social media platforms' functionalities such as authentication, sharing, and posting. We analyzed 6 SDKs that use CTs and 10 SDKs that use WebViews, primarily to enable authentication. We found that ~23.8K apps use CTs, while only ~1.7K apps use WebViews. Among the apps that use CTs, ~98% of them rely on Facebook's SDK, which deprecated WebViews in October 2021 due to an increase in phishing attempts via WebViews [7]. When users visit Facebook's site via a WebView, Facebook shows a "Log in Disabled" error, as shown in Figure 5. With CTs, users can stay logged in to Facebook across sessions, which increases conversion rates for apps authenticating via Facebook. Similarly, NAVER, a South Korean online service, has also deprecated WebViews in favor of CTs for OAuth authorization [27]. We identified 406 apps that use WebViews and 156 apps that use CTs via NAVER's SDK. Using CTs for authorization requests is also in line with the best practices set out in the IETF RFC 8252 for 'OAuth 2.0 for Native Apps' [48].

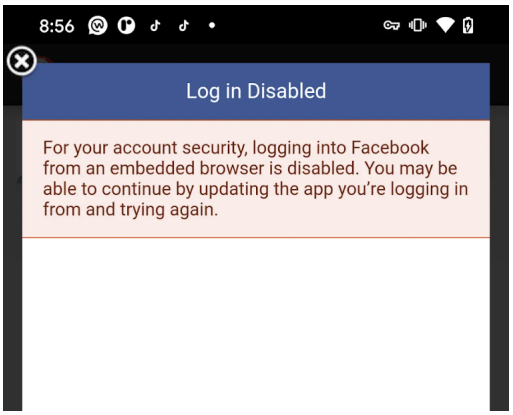


Figure 5: Facebook does not allow its users to authenticate on its website if Facebook's site is visited via a WebView [6].

However, not all social media SDKs have adopted CTs for authentication yet. For example, the SDK for VK, a Russian social media platform, uses WebViews and is used by 456 apps to authenticate via VK. Kakao's SDK, another South Korean platform, also uses WebViews in 347 apps for OAuth. Additionally, 54 apps use CTs via Kakao's utilities SDK, but their purpose is unclear.

Takeaway: Social media SDKs should prefer CTs, especially for authentication. Widely-used SDKs such as Facebook and NAVER have already switched to CTs as it provides better usable security, however there still exist SDKs such as VK and Kakao which use WebViews.

4.1.7 Utility. We define 'Utility' SDKs as those that integrate useful features and utilities from other online services. We measured 4 Utility SDKs that use WebViews and 2 that use CTs in our study. For example, NAVER Maps SDK, which is used by 130 apps, provides map-related functionality to apps. Ticketmaster's SDK allows app developers to access Ticketmaster features such as ticketing and booking management. It uses both WebViews and CTs for authentication and checkout processes. We also detected 16 apps that use CTs through the MyChart SDK, an online service that enables patients to schedule appointments and access their medical records. We argue that WebViews are appropriate for utilities such as Maps, where they can leverage the capabilities of WebViews, but CTs are preferable for utilities such as MyChart, where sensitive information is involved.

4.1.8 Authentication. Authentication or identity SDKs help developers to add authentication and authorization features with identity providers (IDPs). They offer methods and classes to perform tasks such as signing in, signing up, requesting permissions, verifying credentials, and managing sessions. The threats discussed in Section 4.1.6 for social media SDKs also apply to these SDKs. Authentication SDKs should avoid using WebViews because they handle sensitive information such as credentials, which can be compromised. User experience (UX) would also improve with CTs, as the users' active sessions will persist, and they can also use password managers without having to manually enter credentials. Moreover, WebAuthn, an upcoming web standard for secure authentication via FIDO2-based security keys or passkeys, is supported in CTs, but not in WebViews [8].

We measured 7 identity SDKs that use WebViews in our study. They include Gigya (120 apps), a customer identity and access management platform used by SAP, NAVER corporate identity SDK (90 apps), and Amazon's IDP (37 apps). However, a vast majority of apps (~7.8K) used CTs for authentication. Among them, ~97% of the apps used Google Firebase's Authentication SDK, which allows apps to authenticate into custom IDPs and perform reCAPTCHA checks, among other functionality [4].

Takeaway: IDPs should prefer using CTs in their SDKs to protect sensitive information and improve the user experience (via password managers and passkeys). Popular SDKs, such as Google Firebase, use CTs, while some others, such as Gigya and Amazon, still use WebViews.

4.1.9 Hybrid Functionality. Hybrid Functionality refers to the ability to combine web technologies (HTML, CSS, and JS) with native features of the device in an app. As such, hybrid apps are an ideal use-case for WebViews. Our measurements also find apps that use both WebViews and CTs to create hybrid gaming (Baby Panda World, Cube Storm) and news reading (Scripps News) apps.

Recommendations:

- Not all third-party web content requires delivery via CTs. SDK developers should carefully consider the nuances we discuss prior to making the transition. SDKs that facilitate engagement measurement, developer tools, user support, utility, and hybrid functionalities may not necessarily benefit from the transition to CTs, as they utilize WebView functionality legitimately.
- For use-cases that involve handling sensitive information, we advise developers transition to CTs. Secure practices implemented by SDKs can have an outsized positive impact on the security and privacy of the app ecosystem, as several thousand apps rely on them.

4.2 WebView-Based IABs

Android launches a Web URI Intent when a user clicks on an HTTP(S) URL within an app. As per Android's documentation, the default browser handles the Web URI intent on Android 12 and later versions, unless there is an app installed that can handle URLs from that specific domain [35]. For instance, a `maps.google.com` URL clicked from a social media app will launch the Google Maps app if it is present. Otherwise, it will open in the default browser. However, our semi-manual analysis of the top 1K apps, as detailed in Section 3.2.1, revealed 11 popular social media apps that do not launch a Web URI intent, but instead, open the URL using an In-App Browser (IAB). Among them, 10 apps used WebView-based IABs to open HTTP(S) URLs. This implementation is problematic for several reasons. First, WebViews are designed to display first-party content, not third-party content. Second, WebViews are insecure and vulnerable to various attacks, as discussed in Section 2. Third, WebViews have extensive functionalities that can potentially be abused by apps. In this section, we will examine our comprehensive measurement datasets, as collected in Section 3.2.2, to investigate whether such exploitation is occurring in practice.

We begin by examining App-WebView Interactions, which indicate whether and how apps use privileged WebView API methods. We observed that Snapchat, Twitter, and Reddit did not inject any JS (via `evaluateJavascript`) or JS Bridge (via `addJavascriptInterface`) into the WebView. Pinterest injected a JS Bridge into the WebView-based IAB that was triggered when a URL in the DM activity was clicked, but the name of the exposed Java class was obfuscated, so we could not determine the purpose of the injection. For the remaining 6 out of 10 apps, we detected both JS and JS Bridge injections. Facebook and Instagram exhibited identical behavior, as did Moj and Chingari. LinkedIn and Kik showed their own unique behaviors. We explore these 4 behaviors in more depth by analyzing our measurements collected in Section 3.2.2. To gain more insight into the reasons behind a certain measurement, we also manually investigated using Android logs collected by Logcat [34], and by using the remote GUI debugging tool for Android [31]. They facilitated our manual investigation with fine-grained and real-time information, that helped us comprehend activity on the device. We summarize our findings of the inferred intents for all 10 apps in Table 8.

No. of Downloads	App Name	WebView Via	Inferred Intent for Injected Content	
			HTML/JS Injected	JS Bridge Injected
8.4B	**Facebook**	Post	Returns DOM Tag Counts. Returns simHash for page to detect cloaking[53].	Meta Checkout. AutoFillExtensions. Facebook Pay.
4.6B	**Instagram**	DM	Logs performance metrics. Insert FB Autofill SDK JS script.	
2.34B	**Snapchat**	Story	No injection.	No injection.
1.38B	**Twitter**	DM	No injection.	No injection.
1.2B	**LinkedIn**	Post	Calls to Cedexis traffic management API.	No injection.
840M	**Pinterest**	DM	No injection.	(Obfuscated)
289M	**Moj**	Profile Bio	Insert and manage a video Ad via Google Ads SDK.	Google Ads.
97.5M	**Chingari**			
124M	**Reddit**	DM	No injection.	No injection.
176.5M	**Kik**		Insert ads via Ad Networks: Google Ads, MoPub and InMobi.	Google Ads.

Table 8: Summary of WebView injection and its inferred intents, as studied in the WebView-based IABs.To infer the intent, we manually inspect the arguments the app passes to the WebView via WebView methods such as `loadUrl`, `evaluateJavascript`, `loadData` and `loadDataWithBaseURL`, when we visit our controlled web page.

4.2.1 Facebook and Instagram. Facebook and Instagram are both popular social media apps operated by Meta, with 8.4B+ and 4.6B+ downloads respectively. We manually analyzed the logcat logs when a user clicks on a URL in a Facebook news feed post or an Instagram direct message. We found that no intent was raised and instead, a WebView opened to load the URL. Using remote debugging, we discovered that the URL was implemented as a button that triggered the app logic to open the WebView. Our App-WebView interaction measurements further revealed that the app injected several JS scripts and JS Bridges into the web content after loading. The code was well documented and we could infer the purpose of each injection. The following JS injections were performed:

- A JS script that inserted a Facebook Autofill script element into the page, as shown in Listing 1. We believe that the script was used to populate merchant checkouts with user information such as name, address, and phone number from the user's Facebook profile.

```
(function(d, s, id){
    var sdkURL = "//connect.facebook.net/en_US/iab.
     autofill.enhanced.js";
    var js, fjs = d.getElementsByTagName(s)[0];
    if (d.getElementById(id)) {
        return;
    }
    js = d.createElement(s);
    js.id = id;
    js.src = sdkURL;
    fjs.parentNode.insertBefore(js, fjs);
}(document, 'script', 'instagram-autofill-sdk'));
```

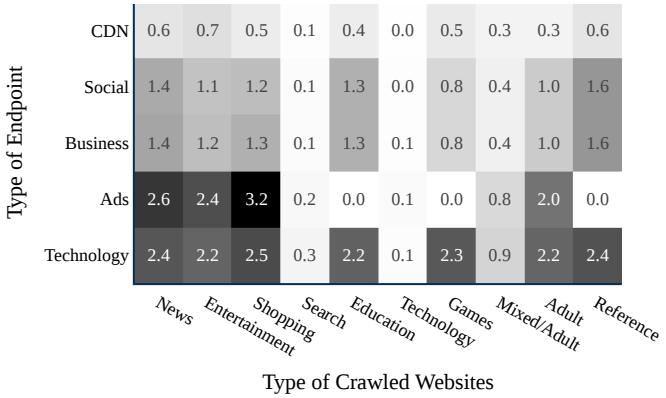
Listing 1: JS executed by Facebook and Instagram's WebView-based IAB to embed a JS SDK that populates merchant checkouts with user information from their Facebook profile.

- A JS script that returned a frequency dictionary with the DOM tag counts.
- A JS script that returned locality sensitive hashes for (i) text and DOM elements, (ii) text elements, and (iii) DOM elements. In-line comments indicated that this information was collected to detect client-side cloaking based on Cloaker Catcher by Duan et al. [53].
- A JS script that logged performance metrics to the console. It recorded the time it took to load the DOM content and whether the page was an Accelerated Mobile Pages (AMP) supported page.

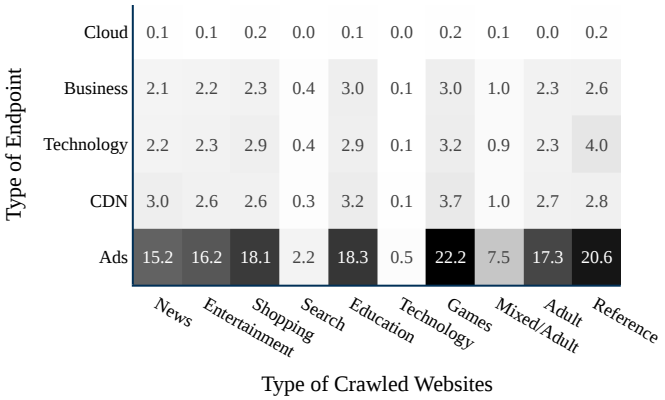
In the visit to our controlled web page via Facebook and Instagram's WebView-based IABs, we also recorded the methods of the Document and Element Web APIs being called, as summarized in Appendix Table 9. This confirmed that the injected JS code was not just injected, but also executed.

Besides JS code, the WebViews also had access to JS bridges, namely `fbpayIAWBridge`, `metaCheckoutIAWBridge`, and `_AutofillExtensions`. We believe that the first two enable payments via Facebook [24] and checkouts via Meta [9], if the user visits a website that supports these features (e.g., by clicking on an ad). The third one provides the Java interface that the autofill JS code can use to retrieve user information from their Facebook profile. The network logs revealed that Facebook and Instagram's WebViews used redirectors hosted at `lm.facebook.com/l.php` and `l.instagram.com` respectively. They passed the intended URL and a random identifier to the redirector in the GET request, which could be exploited for tracking the user [66] (a similar redirect via `t.co` was also observed for Twitter). In the crawl of the top 100 websites, the WebViews did not make any network requests other than those to the intended website and its resources.

4.2.2 LinkedIn. LinkedIn is a social network for professionals, and its Android app has more than 1.2 billion downloads. It uses a WebView-based IAB to display links found in their news feed posts. Other than visiting the intended webpage, we found that the IAB also loads resources from `radar.cedexis .com` and runs JS code to interact with the Cedexis Radar API at `cedexis-radar.net`. Cedexis Radar, currently part of Citrix's NetScaler Intelligent Traffic Management, aims to 'collect network performance data to enable smart routing decisions' [30]. It offers its customers, such as LinkedIn, access to a large set of network performance data



(a) Distribution of number of unique endpoints contacted by LinkedIn's IAB. It most frequently contacts external trackers, such as Cedexis (`cedexis-radar.net`) and LinkedIn's own services, such as CDN (`licdn.com`), ad engagement APIs (`px.ads.linkedin.com`) and performance monitoring (`perf.linkedin.com`).



(b) Distribution of number of unique endpoints contacted by Kik's IAB. It most frequently contacts external ad networks, such as MoPub (`ads.mopub.com`) and InMobi (`supply.inmobicdn.net`), in addition to CDNs such as Amazon CloudFront (`cloudfront.net`).

Figure 6: We crawl 100 top sites via WebView-based IABs of the 10 apps listed in Appendix Table 8, and show the distribution of endpoints contacted specifically by an app's WebView-based IAB. In this figure, we show the average number of distinct endpoints of a certain kind (y-axis) contacted by the IABs of LinkedIn and Kik, when they access a certain kind of site (x-axis).

collected from real users worldwide, in return for conducting and sharing network measurements from their own end-users. Radar's documentation states that it measures availability, response time, and throughput across different Cloud and CDN providers on an end-user's device. Radar's Android SDK called AndroidRadar, outlines how developers can integrate their network measurement library into their app WebViews [13].

Figure 6a illustrates the distribution of distinct endpoints contacted by LinkedIn's IAB, per different types of websites crawled via the IAB. These endpoints were specific to LinkedIn's IAB and were not contacted by any other app's IAB. We classified the endpoint types using Symantec Sitereview [93]. We observed that for websites with rich content, such as News, Entertainment, and Shopping, LinkedIn's IAB contacted more than 2 trackers on average (e.g., Cedexis trackers), as well as its own services, such as CDN (`licdn.com`), ad engagement APIs (`px.ads.linkedin .com`) and performance monitoring (`perf.linkedin.com`). We observed that the average number of endpoints contacted was smaller for Search or Technology websites, presumably because they contained less content to interact with. Our results broadly corroborate with the functionality of Radar as described in its documentation.

4.2.3 Moj and Chingari. Moj and Chingari are short video-based social media apps, developed by two different developers for the Indian audience, and they had more than 289M and 97M downloads, respectively. We discovered that both apps exhibit identical behavior in terms of WebView injections. The injected JS code was highly obfuscated, making it difficult to understand its functionality. However, we manually inspected the names of the JS Bridges exposed and the variable names used by the JS code and inferred that the injections aimed to insert and manage video ads via the Google Ads SDK. Our inference was supported by the fact that one of the JS bridges injected was named `googleAdsJsInterface`, and the injections included JSON data fields with Ad specifications of ads from `doubleclick.net` – which is known to serve Google Ads [81]. Despite the numerous injections, we did not observe any ads on our test page, nor did our server record any Web API usage. Upon further inspection of the JSON objects containing the ad details for both Moj and Chingari, we observed that although they had populated parameters about which ad to display and where to fetch it from, the width and height were fixed to 0 – and a field 'notVisibleReason' was set to 'noAdView'. Based on this, we believe that the injected code did not execute to show any ad because there was no compatible ad view on the page. If there were an ad view on the page, the app's injection could potentially help Google Ads display an ad more relevant in the context of the app. However, to understand ad injection, we are limited to measurements from our controlled web page, as measurements from our crawl dataset could be contaminated with activity from ad networks used by the crawled web pages themselves.

4.2.4 Kik. Kik, an instant messaging application with 176.5M+ downloads, utilizes a WebView-based IAB to open URLs that users open in private Direct Messaging windows. We discovered that Kik's IAB also injects the `googleAdsJsInterface` bridge that we found injected in Moj and Chingari. However, the JS code injected by Kik was different and markedly more obfuscated than Moj and Chingari. Due to the complex nature of the JS code, we were unable to parse the ad payload, even after manual inspection. Furthermore, as highlighted in Appendix Table 9, our web server logged the IAB's use of only read-only Web APIs, indicating that there was no actual modification to the DOM of our test page. However, our network logs did highlight that the IAB communicated with ad networks. For a more comprehensive understanding, we analyzed the distribution of endpoints contacted by Kik's IAB during our top site crawl, as displayed in Figure 6b. The plot revealed that when

visiting websites rich in content, Kik's IAB communicates with, on average, over 15 ad network endpoints. MoPub (ads.mopub.com) and InMobi (supply.inmobicdn.net) are two prominent ad networks contacted by the IAB. From these findings, we hypothesize that the JS code injected by Kik communicates with a multitude of ad networks, and could be selectively inserting ads, but its logic framework remains unclear to us.

Takeaway: Popular apps are misusing WebViews to show third-party web content within their app, while monitoring and manipulating the web content without explicit consent from the users.

5 Concluding Remarks

In this study, we conducted extensive measurements on popular Android apps with the goal of providing novel insights into the state of web security in mobile apps. Our large-scale investigation illuminated the adoption rate of CTs, a secure and user-friendly alternative to WebViews. Furthermore, our detailed examination of WebView-based IABs in the top 1K apps, each downloaded at least 86M times, revealed practices potentially harmful to the user's security and privacy.

We analyzed ~146.5K apps, each boasting over 100K users, and highlighted the diverse use cases of non-browser mobile apps presenting web content via WebViews and CTs. We contend that for use cases like Authentication and Authorization, using WebViews violates the least privilege principle. CTs, with `CustomTabsCallback` are well-privileged to handle such use cases, while WebViews with functionality such as `evaluateJavascript`, are over-privileged. Their privilege exposes the user's credentials to potential abuse. CTs provided by browsers such as Chrome not only bolster security, but also enhance user experience through the shared state with the user's default browser, a consistent and secure UI, and faster load times. Our study is a first step in quantifying the adoption of CTs in widely used apps. Building upon previous work, we take an evidence-based approach in highlighting the risks for SDKs persisting in using WebViews. We note that many popular SDKs, such as Facebook Login and Google Firebase, have already migrated to CTs; however, a considerable number of extensively used SDKs lag behind. We recognize SDKs that measure user engagement of first-party web content, provide in-app support, and offer other utilities as legitimate use cases of WebViews. Future research could focus on adapting Ad SDKs, the most common WebView application, to CTs, taking advantage of innovations like Partial CTs, which enable developers to launch resizable inline CTs in response to native ads, as showcased by Google in 2023 [29]. This would mitigate the risk posed by WebViews, which has previously been exploited by malicious ads to compromise the user's security and privacy. We note that secure practices implemented by SDKs can have an outsized positive impact on the security and privacy of the app ecosystem, as several thousand apps rely on them.

To further characterize the usage of WebViews to display third-party content, our semi-manual analysis of the top 1K apps uncovers 11 apps that disguised URLs as buttons with hyperlink-like text. These buttons, upon clicking launched an IAB, instead of raising a Web URI intent. Typically, a Web URI intent is handled either by an appropriate app or the default browser, however, these apps force the URL to open within the app itself. A deeper probe revealed a variety of unique behaviors. Apps such as Facebook and Instagram used this method to facilitate payments and checkouts, despite the potential for misuse by malicious websites to pilfer personal user information. Other apps like Moj, Chingari, and Kik did so for potential ad injection, while LinkedIn incorporated a network measurement SDK to source performance metric data from user devices. Despite the seemingly benign behaviors found in WebView-based IABs, we suggest that the extensive attack surface they expose renders the trade-off inadequate. Although some apps offer an option to disable in-app browsers for privacy-focused users [3], we recommend apps make it an opt-in feature and implement IABs using CTs, for improved user safety. Alternatively, websites can also take proactive steps to handle web sessions from WebViews differently. Every request that comes from a WebView has a `X-Requested-With` header field with the app's APK name as its value [79]. The steps could vary from showing the user a prompt to inform and consent to the risks involved in performing an action (such as logging in or making a transaction), to completely blocking access to sessions from WebViews, as Facebook did [6]. Interestingly, Facebook prevents its users from logging in on its website via a WebView (as shown in Figure 5), but opens third-party links in a WebView itself, ignoring the users' security and privacy on other websites.

Our community has consistently observed that adopting new, secure technologies necessitates not only the education of developers but also the alignment of incentives. We hope that our empirical findings will contribute to achieving these objectives. Our research lays the foundation for improving transparency regarding how mobile apps display third-party web content. Future research could consider including WebView usage for third-party content as a metric in the 'privacy nutrition labels' as displayed on the app store. Android could also explore measures such as Apple's App Tracking Transparency, which enables users to choose whether an app's WebView can track their activity across other companies' websites for the purposes of advertising or sharing with data brokers [14]. Although cross-party tracking on the web remains an arms race, such measures enable users to make educated decisions about their online safety [62], and incentivize the ecosystem of apps and SDKs to adopt safer practices [65].

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We would like to express our gratitude to Dr. Paul Pearce, Dr. Brendan Saltaformaggio, and Dr. Vijay Madisetti for loaning their Android mobile devices and enabling our experimental work.

References

[1] 2015. *Android Developers Blog: Chrome custom tabs smooth the transition between apps and the web.* https://android-developers.googleblog.com/2015/09/chrome-custom-tabs-smooth-transition.html.

[2] 2015. Chrome custom tabs smooth the transition between apps and the web. https://android-developers.googleblog.com/2015/09/chrome-custom-tabs-smooth-transition.html.

[3] 2019. *How to Disable In-App Browser for Android Apps.* https://gadgetstouse.com/blog/2019/12/21/how-to-disable-in-app-browser-for-android-apps/

[4] 2020. *Firebase Android SDK Release Notes.* https://firebase.google.com/support/release-notes/android#2020-11-12

[5] 2020. *Securing WebViews with Chrome Custom Tabs.* https://plaid.com/blog/securing-webviews-with-chrome-custom-tabs.

[6] 2021. *Deprecating Facebook Login support on Android WebViews.* https://developers.facebook.com/docs/facebook-login/android/deprecating-webviews/.

[7] 2021. *Deprecating support for FB Login authentication on Android embedded browsers.* https://developers.facebook.com/blog/post/2021/06/28/deprecating-support-fb-login-authentication-android-embedded-browsers/.

[8] 2022. *Google Groups: Clarifying WebView and Chrome Custom Tabs support for Android's implementation of Passkeys.* https://groups.google.com/a/fidoalliance.org/g/fido-dev/c/SWuq7ORnnLQ.

[9] 2023. *About checkout on Facebook and Instagram Shops.* https://www.facebook.com/business/help/2509359009104717?id=533228987210412

[10] 2023. *androguard/androguard.* https://github.com/androguard/androguard

[11] 2023. *Android Developers: Web-based content.* https://developer.android.com/develop/ui/views/layout/webapps.

[12] 2023. *Android Developers: WebView.* https://developer.android.com/reference/android/webkit/WebView

[13] 2023. *AndroidRadar.* https://github.com/cedexis/androidradar

[14] 2023. *App Tracking Transparency.* https://developer.apple.com/documentation/apptrackingtransparency

[15] 2023. *Create Deep Links to App Content.* https://developer.android.com/training/app-links/deep-linking.

[16] 2023. *Custom Tabs: Getting started.* https://developer.chrome.com/docs/android/custom-tabs/guide-get-started.

[17] 2023. *Flutter.* https://flutter.dev

[18] 2023. *flutter_inappwebview.* https://pub.dev/packages/flutter_inappwebview

[19] 2023. Frida. https://frida.re.

[20] 2023. *google-play-scraper.* https://pypi.org/project/google-play-scraper/

[21] 2023. *Google Play SDK Index.* https://play.google.com/sdks.

[22] 2023. *Integrate the WebView API for Ads.* https://developers.google.com/admob/android/webview

[23] 2023. *LineageOS Android Distribution.* https://lineageos.org/.

[24] 2023. *Meta Pay.* https://pay.facebook.com

[25] 2023. Mobile vs. Desktop vs. Tablet Traffic Market Share. https://www.similarweb.com/platforms/

[26] 2023. *Naming a Package.* https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

[27] 2023. *NidOAuthBehavior.kt.* https://github.com/naver/naveridlogin-sdk-android/blob/master/Nid-OAuth/src/main/java/com/navercorp/nid/oauth/NidOAuthBehavior.kt.

[28] 2023. *Open a Custom Tab for links in a WebView.* https://developer.chrome.com/docs/android/custom-tabs/howto-custom-tab-from-webview

[29] 2023. *Partial Custom Tabs.* https://developer.chrome.com/blog/whats-new-in-web-on-android-io2023/#partial-custom-tabs

[30] 2023. *Radar.* https://docs.netscaler.com/en-us/citrix-intelligent-traffic-management/radar.html

[31] 2023. *Remote debug Android devices.* https://developer.chrome.com/docs/devtools/remote-debugging/

[32] 2023. *System WebView Shell.* https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/webview-shell.md

[33] 2023. *url_launcher.* https://pub.dev/packages/url_launcher

[34] 2023. *View logs with Logcat.* https://developer.android.com/studio/debug/logcat

[35] 2023. *Web links.* https://developer.android.com/training/app-links#web-links.

[36] 2024. *Optimize Custom Tabs (Beta).* https://developers.google.com/admob/android/browser/custom-tabs

[37] Google AdMob. 2023. *Ad units, ad formats, & ad types.* https://support.google.com/admob/answer/6128738

[38] Abdulla Aldoseri and David Oswald. 2022. insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers. In *Workshop on Measurements, Attacks, and Defenses for the Web.*

[39] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *International Conference on Mining Software Repositories.*

[40] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* (2014).

[41] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *ACM Conference on Computer and Communications Security.*

[42] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. 2018. BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications. *IEEE Transactions on Information Forensics and Security* (2018).

[43] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. 2024. Tabbed Out: Subverting the Android Custom Tab Security Model. In *IEEE Symposium on Security and Privacy.*

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

[45] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation. In *USENIX Security Symposium.*

[46] Chris Bracco. 2023. HTML5 Test Page. https://github.com/cbracco/html5-test-page

[47] Gong Chen, Wei Meng, and John Copeland. 2019. Revisiting Mobile Advertising Threats with MAdLife. In *The World Wide Web Conference.*

[48] W. Denniss and J. Bradley. 2017. *OAuth 2.0 for Native Apps.* https://datatracker.ietf.org/doc/html/rfc8252

[49] Android Developers. 2023. *App components.* https://developer.android.com/guide/components/fundamentals.html

[50] Android Developers. 2023. *UI/Application Exerciser Monkey.* https://developer.android.com/studio/test/other-testing-tools/monkey

[51] Chrome Developers. 2023. Measuring User Engagement. https://developer.chrome.com/docs/android/custom-tabs/guide-engagement-signals

[52] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Security and Privacy in Communication Networks.*

[53] Ruian Duan, Weiren Wang, and Wenke Lee. 2017. Cloaker Catcher: A Client-based Cloaking Detection System. *arXiv preprint arXiv:1710.01387* (2017).

[54] Ankit Gangwal, Shubham Singh, and Abhijeet Srivastava. 2023. AutoSpill: Credential Leakage from Mobile Password Managers. In *ACM Conference on Data and Application Security and Privacy.*

[55] GitHub. 2023. *android-customtabs.* https://github.com/saschpe/android-customtabs

[56] GitHub. 2023. owasp-masvs. https://github.com/OWASP/owasp-masvs

[57] GitHub. 2023. *skylot/jadx.* https://github.com/skylot/jadx

[58] Google. 2023. Google Safe Browsing. https://safebrowsing.google.com

[59] Google. 2023. *NetLog: Chrome's network logging system.* https://www.chromium.org/developers/design-documents/network-stack/netlog/

[60] Google. 2024. Web on Android. https://web.dev/articles/web-on-android.

[61] 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 *ACM/IEEE International Conference on Automated Software Engineering.*

[62] Hannah J Hutton and David A Ellis. 2023. Exploring User Motivations Behind iOS App Tracking Transparency Decisions. In *CHI Conference on Human Factors in Computing Systems.*

[63] Juspay. 2023. AmazonPay S2S Tokenised Flow. https://developer.juspay.in/v5.1/docs/amazonpay-s2s-tokenised-flow

[64] Nico Kokonas. 2023. Trace.js. https://gist.github.com/nicoandmee/62ecd1829d761fbed779dc3a3ba35c64

[65] Konrad Kollnig, Anastasia Shuba, Max Van Kleek, Reuben Binns, and Nigel Shadbolt. 2022. Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels. In *ACM Conference on Fairness, Accountability, and Transparency.*

[66] Martin Koop, Erik Tews, and Stefan Katzenbeisser. 2020. In-Depth Evaluation of Redirect Tracking and Link Usage. *Proceedings on Privacy Enhancing Technologies* (2020).

[67] Dhruv Kuchhal and Frank Li. 2021. Knock and Talk: Investigating Local Network Communications on Websites. In *ACM Internet Measurement Conference.*

[68] IAB Tech Lab. 2023. Open Measurement SDK. https://iabtechlab.com/standards/open-measurement-sdk/

[69] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: Static analysis framework for Android hybrid applications. In *IEEE/ACM International Conference on Automated Software Engineering.*

[70] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).

[71] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé Bissyandé, and Jacques Klein. 2020. MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps. In *The Web Conference.*

[72] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Annual Computer Security Applications Conference.*

[73] Samin Yaseer Mahmud, K Virgil English, Seaver Thorn, William Enck, Adam Oest, and Muhammad Saad. 2022. Analysis of Payment Service Provider SDKs in Android. In *Annual Computer Security Applications Conference.*

[74] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. 2021. A Large-Scale Empirical Study of Android App Decompilation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering.*

[75] Mozilla. 2023. Web APIs: MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API.
[76] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A Large-Scale Study of Mobile Web App Security. In *Mobile Security Technologies Workshop.*
[77] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security.*
[78] Sajjad Pourali, Nayanamana Samarasinghe, and Mohammad Mannan. 2022. Hidden in Plain Sight: Exploring Encrypted Channels in Android apps. In *ACM Conference on Computer and Communications Security.*
[79] Amogh Pradeep, Álvaro Feal, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, David Choffnes, et al. 2023. Not Your Average App: A Large-scale Privacy Analysis of Android Browsers. In *Privacy Enhancing Technologies Symposium.*
[80] PyPI. 2023. javalang. https://pypi.org/project/javalang/
[81] Hina Qayyum, Muhammad Salman, I Wayan Budi Sentana, Duc Linh Giang Nguyen, Muhammad Ikram, Gareth Tyson, and Mohamed Ali Kaafar. 2022. A First Look at Android Apps' Third-Party Resources Loading. In *Network and System Security.*
[82] 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.*
[83] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. 2022. Toppling Top Lists: Evaluating the Accuracy of Popular Website Lists. In *ACM Internet Measurement Conference.*
[84] secure-software engineering/FlowDroid. 2018. Analysis does not finish. https://github.com/secure-software-engineering/FlowDroid/issues/27
[85] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium.*
[86] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What Mobile Ads Know About Mobile Users. In *Network and Distributed System Security Symposium.*
[87] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-Force: Understanding WebView Malware via Cross-Language Forced Execution. In *ACM/IEEE International Conference on Automated Software Engineering.*
[88] Abhishek Tiwari, Jyoti Prakash, Sascha Groß, and Christian Hammer. 2019. LUDroid: A Large Scale Analysis of Android – Web Hybridization. In *International Working Conference on Source Code Analysis and Manipulation.*
[89] Abhishek Tiwari, Jyoti Prakash, Sascha Gross, and Christian Hammer. 2020. A Large Scale Analysis of Android — Web Hybridization. *Journal of Systems and Software* (2020).
[90] 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 preprint arXiv:2208.01968* (2022).
[91] Abhishek Tiwari, Jyoti Prakash, Alimerdan Rahimov, and Christian Hammer. 2023. Understanding the Impact of Fingerprinting in Android Hybrid Apps. In *International Conference on Mobile Software Engineering and Systems.*
[92] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. 2016. Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android. In *ACM Conference on Computer and Communications Security.*
[93] Pelayo Vallina, Victor Le Pochat, Álvaro Feal, Marius Paraschiv, Julien Gamba, Tim Burke, Oliver Hohlfeld, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. Mis-shapes, Mistakes, Misfits: An Analysis of Domain Classification Services. In *ACM Internet Measurement Conference.*
[94] Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the unsoundness of static analysis for Android GUIs. In *ACM International Workshop on State Of the Art in Program Analysis.*
[95] Guangliang Yang and Jeff Huang. 2018. Automated Generation of Event-Oriented Exploits in Android Hybrid Apps. In *Network and Distributed System Security Symposium.*
[96] Guangliang Yang, Jeff Huang, and Guofei Gu. 2019. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *USENIX Security Symposium.*
[97] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. 2017. Precisely and Scalably Vetting JavaScript Bridge In Android Hybrid Apps. In *Research in Attacks, Intrusions, and Defenses.*
[98] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. 2022. Identity Confusion in WebView-based Mobile App-in-app Ecosystems. In *USENIX Security Symposium.*
[99] Xiaohan Zhang, Haoqi Ye, Ziqi Huang, Xiao Ye, Yinzhi Cao, Yuan Zhang, and Min Yang. 2023. Understanding the (In)Security of Cross-side Face Verification Systems in Mobile Apps: A System Perspective. In *IEEE Symposium on Security and Privacy.*
[100] 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 *USENIX Security Symposium.*
[101] Zicheng Zhang. 2021. On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps. In *International Symposium on Research in Attacks, Intrusions and Defenses.*
[102] Zhibo Zhang, Zhangyue Zhang, Keke Lian, Guangliang Yang, Lei Zhang, Yuan Zhang, and Min Yang. 2023. TrustedDomain Compromise Attack in App-in-app Ecosystems. In *ACM Workshop on Secure and Trustworthy Superapps.*

A Appendix

App Name	Web API used	
	Interface	**Method**
Facebook & Instagram	Document	getElementById
		createElement
		querySelectorAll
		getElementsByTagName
		addEventListener
		removeEventListener
	Element	insertBefore
		hasAttribute
		getElementsByTagName
	HTMLBodyElement	insertBefore
	HTMLCollection	item
	NodeList	item
	HTMLMetaElement	getAttribute
Kik	HTMLDocument	querySelectorAll
	HTMLMetaElement	getAttribute
	Document	querySelectorAll

Table 9: Web APIs accessed by apps, as recorded by our controlled web page server.

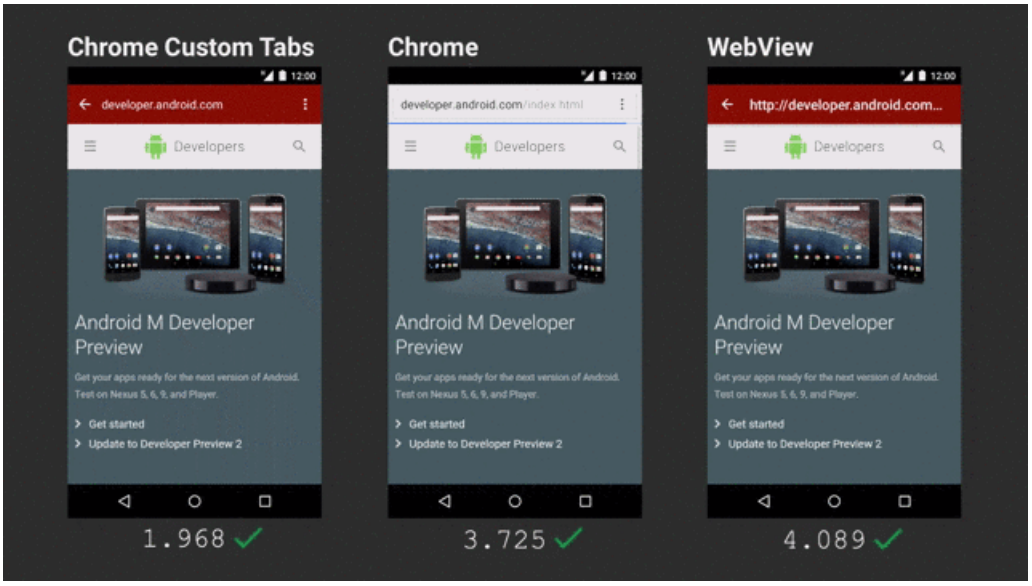


Figure 7: (L-R) Screenshots from a test in which the same page loaded in Custom Tabs (CT) within an app, Chrome, an external browser, and a WebView within an app [2]. The CT was fastest at loading the page – twice as fast as a WebView.

B Ethics

In conducting our manual analysis of the top 1K apps, we ensured that our experimentation methodology did not induce a high load on online services, as we manually navigated the apps and tested how they implement IABs using only one benign URL. When carrying out our automated crawls, we recognized the possibility that some of the URLs could potentially contain sensitive or malicious content. As a proactive measure to safeguard other users, we ensured that none of the URLs were posted publicly, thereby eliminating any potential harm to other users on the platform.

Following ethical research practices, we reported our findings to Google's Trust and Safety team via the Abuse Vulnerability Reward Program's responsible disclosure process, and are actively participating in the discussion, as of September 2024.