Understanding the Impact of Fingerprinting in Android Hybrid Apps

Abhishek Tiwari*
University of Passau
Passau, Germany
abhishek.tiwari@uni-passau.de

Jyoti Prakash*
University of Passau
Passau, Germany
jyoti.prakash@uni-passau.de

Alimerdan Rahimov
University of Passau
Passau, Germany
rahimo01@ads.uni-passau.de

Christian Hammer
University of Passau
Passau, Germany
christian.hammer@uni-passau.de

Abstract—**Numerous studies demonstrate that browser fingerprinting is detrimental to users' security and privacy. However, little is known about the effects of browser fingerprinting on Android hybrid apps – where a stripped-down Chromium browser is integrated into an app. These apps expand the attack surface by permitting two-way communication between native apps and the web. This paper studies the impact of browser fingerprinting on these embedded browsers. To this end, we instrument the Android framework to record and extract information leveraged for fingerprinting. We study over 60,000 apps, including the most popular apps from the Google play store. We exemplify security flaws and severe information leaks in popular apps like Instagram. Our study reveals that fingerprints in hybrid apps potentially contain account-specific and device-specific information that identifies users across multiple devices uniquely. Besides, our results show that the hybrid app browser does not always adhere to standard browser-specific privacy policies.**

Index Terms—**Hybrid Apps, Android Webview, Privacy**

I. INTRODUCTION

Browser fingerprinting is an effective method to identify individuals based on information accessible through browser settings while eliminating local information, e.g., in cookies. Web pages capture distinguishable information about the user and the environment, such as the timezone and locale. Several websites leverage browser fingerprinting to detect botnets and other harmful activities, such as an account accessed from an unusual location or device. On the flip side, online entities exploit fingerprinting to develop targeted advertisements, price inflation for identified individuals, and targeted malware for particular browser/operating system versions.

Multiple studies [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] acknowledged the privacy and security implication of this topic in the last decade. The majority of these studies targeted desktop browsers; however, recent years have seen a technological shift towards mobile devices rather than desktop PCs for internet browsing. A recent study [11] explored fingerprinting on mobile browsers and demonstrated fingerprinting to be quite effective on mobile browsers. However, to the best of our knowledge, there are no studies to understand the impact of fingerprinting on hybrid apps.

Hybrid mobile apps integrate native and web components into a single mobile application. Hybrid apps, on the surface, are native applications combined with web technologies such as JavaScript. Hybrid apps offer advantages to developers as they facilitate reusability across multiple platforms: Existing web apps, e.g., login pages, may effortlessly be integrated into multiple mobile platforms (e.g., iOS and Android) to save time and development costs. In this work, we explore the implications of browser fingerprinting on Android hybrid apps. The Android framework provides the WebView [12] class to embed web applications into a view of the Android app displaying webpages in a Chromeless browser [13].

WebView also provides an active communication channel between the native Android app and JavaScript in the browser. JavaScript can access the Android app's functionality through shared objects, which grant web components strong capabilities to access native Android APIs without the need to ask for Android permissions individually. In contrast to Android's permission system, where users can authorize permissions just once (perhaps in a completely different context), on the web, users must approve sensitive access (e.g., location access) or grant it for one day. However, a hybrid app's inbuilt browser inherits this permission (if the shared Android component has this permission) without further user interference. There have been multiple studies [14], [15], [16], [17] to understand the security and privacy implications of hybrid apps in Android. These studies demonstrate multiple scenarios where hybrid apps are insecure with respect to users' security and privacy. Many hybrid apps use insecure protocols and send private information to third-parties. Unfortunately, the impact of fingerprinting the hybrid app's inbuilt browser is still unknown.

In this work, we bridge the gap in understanding the impact of hybrid apps' browser fingerprinting. We perform a large-scale study of fingerprints generated by hybrid Android apps. In particular, we are interested in information leakage, user tracking, and security implications arising from the bridge communication capabilities of hybrid apps. The bridge communication provides (potentially untrusted) web components of hybrid apps access to the trusted native app's data and functionality. In this work, we explore how the web counterparts of a hybrid app exploit these capabilities to expose information via fingerprinting. Besides, we identify the differences in fingerprinting between the stand-alone and the browser in hybrid apps. To this end, we study over 60,000 apps, including the most popular apps from the Google play

---

*joint first author

store. To obtain the fingerprint of the hybrid app's browser, we employ dynamic instrumentation of WebView using the Frida instrumentation framework [18]. Frida provides a dynamic instrumentation toolkit to inject code into the Android Framework programmatically. In particular, Frida supports overloading of existing methods of the Android Framework. We develop a tool, CHARLIE, based on Frida to identify and collect the browser fingerprints. CHARLIE instruments the Android framework to overload the *loadUrl*, *postUrl* methods of the WebView class, and the *onLoadResource* method of *WebViewClient*. In particular, the instrumentation is targeted to collect three key pieces of information; User Agents string, custom headers, and URLs. URLs help identify the unencrypted traffic originating from *loadUrl*. Custom headers and the User Agents string help identify privacy leaks and unique identifiers associated with the web request. Finally, we exemplify the security flaws and information leaks on popular apps like Instagram. In summary, our study reveals that some apps' fingerprints contain account-specific and device-specific information that can be used to identify and link their users over multiple devices uniquely. Besides, our results show that the hybrid app browser does not always adhere to standard browser-specific privacy policies.

To summarize, this study contributes the following:

- *A Large-scale analysis of Hybrid app's browser fingerprinting* We perform a large-scale analysis of the Hybrid app's browser fingerprinting. Our analysis helps to understand the privacy and security implications of fingerprinting on Android hybrid apps. We explore that the hybrid app browser does not adhere to standard browser-specific privacy policies due to customization inability. Besides, many popular apps' fingerprints contain account-specific and device-specific information that can be used to identify users over multiple devices uniquely.
- CHARLIE We develop a tool, CHARLIE, based on Frida to identify and collect the browser fingerprints. We make our tool public [19] for researchers to reuse and build upon it.
- *Dataset* We open-source the datasets [19] used in our study to help researchers and developers reproduce and understand the implication of fingerprinting on hybrid Android apps.

II. MOTIVATION AND BACKGROUND

Before delving into the details of our core framework and the implications of browser fingerprinting in Android hybrid apps, we provide a brief background of the techniques utilized in our study.

A. Hybrid Apps

Android hybrid applications embody native Android parts alongside web components. These apps enable developers to reuse their existing web applications in their Android apps. To enable hybrid apps, Android provides a set of APIs to facilitate the communication among Android native app components (primarily written in Java or Kotlin) and web components. These APIs are composed via the Android *WebView* class, which allows the developer to display web pages as a part of the app's activity (e.g., login screen).

```
 1  
 2 //Android side: exposing methods to JavaScript
 3 public class BridgedClass {
 4   public String name;

 6   @JavascriptInterface
 7     public void setValue(String x) {
 8       this.name = x;
 9     }

11     public String getValue(){
12        return this.name;
13     }
14 }
15 //Activity implementing WebView
16 @Override
17 protected void onCreate(Bundle savedInstanceState)
        {
18 //some  code
19   WebView wv = (WebView)
        findViewById(R.id.webview);
20   WebSettings webSettings =
        wv.getSettings().setUserAgentString("My User
        agent");
21   webSettings.setJavaScriptEnabled(true);
22   BridgedClass bClass = new BridgedClass();
23 //share the bridge object to JavaScript
24   wv.addJavascriptInterface(bClass,
        "sharedJavaObject");
25 // JavaScript invoking Android via the shared
        object
26   wv.loadUrl("javascript:" +
        "sharedJavaObject.setValue(\"Hello World\")");
27 // Invoking JavaScript methods
28   wv.loadUrl("javascript:set()");
29 //Loading a url
30   wv.loadUrl("http://www.dummy.com");
31  
32 //JavaScript side
33 set() {
34   x = new Object();
35   const str = new String();
36   x.f = str.concat("x", "y");
37   v = x.f;
38   sharedJavaObject.setValue(v)
39 }
```

Listing 1: Android Hybrid app communication

WebView provides two styles of communication channels between Android and the web. In the first type, an app can invoke a webpage/script without sharing any Android functionality with them. In the second, more interesting *two-way* communication channel, an app actively communicates with a webpage/script by sharing Android-side functionality to the WebView. The example in Listing 1 contains both of these cases. Line 22 and Line 24 present the code (using the `addJavascriptInterface` API) to share an Android object to JavaScript. In our example, Line 3 to Line 12 describe a class `BridgeClass` shared with JavaScript. By default, none of the methods in a class are exposed to JavaScript. The Android framework provides the `@JavascriptInterface` annotation to specify the shared methods of a bridge class. For example, `BridgeClass` does *not* share the `getValue` method to

JavaScript. Line 17 to Line 30 present an Android activity code that creates a WebView. Line 19 and Line 20 provide a general configuration for creating a WebView. By default, the execution of JavaScript is disabled in a WebView. Developers need to manually enable JavaScript by utilizing `setJavaScriptEnabled(true)` (e.g., Line 21). Once enabled, the JavaScript can be invoked using the `loadUrl` method. Line 26 to Line 28 describe two ways to achieve this. Finally, `loadUrl` can also be used to invoke normal URLs, e.g., Line 30.

WebView APIs: WebView provides the following APIs to fetch URLs and execute *JavaScript* scripts.

- *loadUrl(Url)*: It loads the specified Url in the WebView. *loadUrl* can also execute JavaScript code. JavaScript script strings are prepended with `javascript:`.
- *loadUrl(Url, HttpHeaders)*: It has the same functionality as *loadUrl* with additional HTTP headers. Developers can specify the HTTP headers they want to bundle with the request.
- *postUrl(Url, postData)*: It loads the specified network *Url* using the POST method along with the post data.
- *WebViewClient.onLoadResource(webView, Url)* It notifies the host application that WebView *webView* will load the specified *Url*.

WebView User Agent Settings: WebView provides an API to set custom user-agent settings for the WebView browser. Developers can override the user-agent settings, which can be intercepted by the loaded URL. For example, Listing 1 sets the user agent settings to “My User agent” (Line 20).

User-agent settings are useful for user’s security, as well as notorious for breaking it. However, the user agent settings in WebView are a bit different from those on browsers. Recently, desktop and mobile browsers, such as Chrome, Mozilla, and others, allow users to hide sensitive information to evade fingerprinting. However, this provision is lacking in the case of WebView browsers. Here, the control is directly in the hands of the developer. This makes WebView browsers a lucrative option for fingerprinting since these may inherit privacy-sensitive data with the shared native Android app’s functionality. Our study shows that developers have leveraged these features to collect users’ device fingerprints.

B. Browser Fingerprinting

Browser fingerprinting is a technique to profile users to uniquely identify them based on passive information, known as a *browser fingerprint*, obtained from the browser. Browser fingerprint uses the information collected from browsers, such as HTTP headers (e.g., User Agent and Accept and Content Language), Flash plugins, JavaScript cookies, and many others. Recent web advances, such as browser extensions, canvas elements, and WebGL components are also known to be sources of fingerprints [20], [3]. We explain three approaches here: (1) User Agents, (2) Accept and Content-Language, and (3) browser extensions, to aid the understanding of this paper. Interested readers may refer to Laperdrix *et al.* [20] for a detailed survey of browser fingerprinting.

The HTTP protocol is meant to be platform-independent, and therefore, browsers rely on the information from HTTP headers to identify the browser of an incoming request. The information is encoded in the standard HTTP semantics (RFC 9110 [21]) called *User-Agent request headers* or User Agent strings. User-Agent strings specify the system characteristics such as browser, operating system, architecture, and many others, and are used by web servers to identify the client information. As of now, User-Agent strings are complex and add a plethora of information other than the browser. Developers can override the existing user-agent headers and inject information into these headers. For example, developers can modify these strings via JavaScript and add more information, such as the timezone, screen-specific attributes (such as resolution, depth), platform, and many others. This information is a source of fingerprinting as shown by earlier works [20], [1].

Accept headers, specifying the file types accepted by the browser, are another source of fingerprinting [20], [1]. They come as a comma-separated list of content types and their subtypes. For example, a browser can set the accept headers to `text/html, application/xhtml+xml`, which indicates the browser can accept the type `text` of sub-type `html`. Content-Language attribute specifies the localization information of the browsers, such as `de-DE, en-US, en-IN`. Content-language is also a source of localization information for fingerprinting [22].

Browser extensions are browser-based applications that enhance the browsing experience. Although these improve the browser experience, such as reducing ads, they are also a source of fingerprinting information. Starov and Nikiforakis [7] identified 14.10% of users via fingerprints obtained from their browser extensions. They used the changes in the DOM model introduced by the browsers to detect extensions. A similar study from Sanchez-Rola et al. [8] showed the possibility of extension enumeration attacks on browsers, thus identifying 56.28% from 204 users. To this end, they measure the timing difference between querying resources of fake and benign extensions.

a) Large scale studies on browser fingerprinting: Browser fingerprints can compromise users’ privacy. It was first demonstrated in the experiment *Panoptclick* [1] by Peter Eckersley from the Electronic Frontier Foundation, where he fetched around 470,000 fingerprints, of which around 84% were unique. His experiment shows the gravity of the problem, i.e., browser fingerprints can uniquely determine a majority subset of the users on the web. Following up on these experiments, researchers revealed many other sources of browser fingerprinting generation techniques to profile users and break their privacy. We list these techniques in the related work.

The evolution of the Web from desktop to mobile browsers has affected users’ privacy in terms of browser fingerprinting. Earlier research [11] shows that fingerprints from mobile browsers reveal a lot more sensitive information than from desktop browsers. To tackle the problem of fingerprinting, web browsers have started introducing policies to minimize browser fingerprints. Unfortunately, these policies do not apply to the

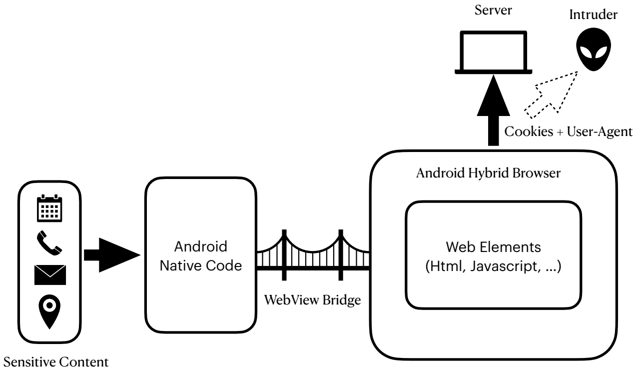


Figure 1: Threat Model

hybrid app's in-built browser, leaving the control in the hands of the developers. We study this aspect in this paper.

b) Uniqueness of a fingerprint: To compare the strength of the information revealed by the fingerprints obtained in our study, we compare it against a larger dataset of *Cover your tracks* [23]. It shows the bits of unique information [24] revealed by the fingerprint, which matches with the fingerprint obtained in our database. *Cover your tracks* shows this information in terms of the number of browsers having the same fingerprint. In this paper, we refer to it as *uniqueness*.

III. SYSTEM DESIGN

A. Threat Model

Figure 1 details the threat model concerning fingerprinting in hybrid Android apps. In this work, we are interested in sensitive information release via fingerprints of hybrid apps' browser. The source of sensitive information comes from the native side of the Android apps. Our threat model assumes that the Android native side is trusted as the sensitive data access is controlled via the Android permission model. The web elements of the hybrid app can access (selected) Android's sensitive sources using the WebView bridge. At the same time, web elements may release more information via fingerprints due to the hybrid browser's privacy policies. Finally, a threat may arise from the web elements' vulnerabilities that cause unintentional information release to an intruder.

B. Workflow

CHARLIE solves numerous technical challenges required to identify and collect fingerprints from Android WebView. With traditional browsers, it is feasible to attach scripts/plugins to a web page and rely on cookies to gather information, which is, unfortunately, not possible with hybrid apps. The hybrid app browser is provided as a part of the Android Framework, and it displays web pages as a part of the app's activity. In this work, we perform runtime instrumentation of the *WebView* class to intercept the fingerprinting data. Generally, network analysis tools such as *Wireshark* could also obtain parts of the required data. However, for a large scale analysis, instrumenting the *WebView* class gives us more control over the data we collect, e.g., *Wireshark* does not associate the apps' identifier to the network traffic containing fingerprinting data. Besides, instrumenting *WebView* enables us to capture the direct traffic from the particular app, while *Wireshark* captures all traffic, including noise from other apps and Android Framework.

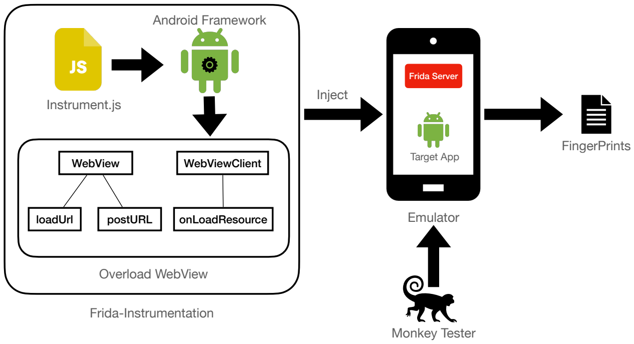


Figure 2: Workflow of CHARLIE

Figure 2 provides an overview of CHARLIE. There are two potential ways to instrument WebView. First, modifying the Android framework by integrating the required code changes directly into the Android Open Source project, and then running the apps on this custom Android OS. Second, achieving the desired modifications with the help of dynamic instrumentation. In this work, we opt for the latter path; we leverage an existing Android dynamic instrumentation framework, Frida [18]. Frida provides a dynamic instrumentation toolkit to inject code into the Android Framework programmatically. In particular, Frida supports overloading the existing methods of the Android Framework. CHARLIE instruments the Android framework to overload the *loadUrl*, and *postUrl* methods of the *WebView* class, and *onLoadResource* of *WebViewClient*. In particular, the instrumentation is targeted to collect three key pieces of information; User Agent strings, custom headers, and URLs. URLs help to identify the unencrypted traffic originated from *loadUrl*. Custom headers and the User Agent help to identify privacy leaks and unique identifiers associated with the transmission. To navigate through various Android activities, we leverage the Android automated tester *Monkey* [25]. Monkey produces pseudo-random streams of user events such as mouse movements and gestures and generates various system-level events for automatic navigation of Android apps. We configured Monkey with 500 pseudo-random events.

Algorithm 1 presents pseudocode for instrumenting the *loadUrl* method with a single parameter and collecting the corresponding fingerprints during the app execution. Line 1 creates an instance object *WebView* pointing to the Android Framework's *WebView* class. In the next line, the *loadUrl* method is overloaded to extract browser fingerprints. In particular, the *loadUrl* method is instrumented to extracts app's unique identifier (package name), custom headers, the user-agent string, and URL. Finally, the app is run via Monkey tester on the instrumented Android runtime to collect these fingerprints for each explored instance of the *loadURL* invo-

Algorithm 1: Instrument loadURL(url)

```
Input: APP $\mathcal{A}$, Android Runtime ART, FridaServer $\mathcal{F}$
Output: Instrumented ART', Fingerprints $\mathcal{B}_{\mathcal{F}}$
WebView ← $\mathcal{F}$.getWebViewClassInstance(ART)
ART' ← $WebView.loadURL$.overload.implement($URL$) ↪
                        { context ← getCurrentAPPContext()
                          packageName ← context.packageName()
                          LOG(packageName, URL, header, userAgent) }
foreach WebView in launchMonkeyTester($\mathcal{A}$, ART') do
    $\mathcal{B}_{\mathcal{F}}$ ← $WebView.loadURL$.collectLog(URL)
```

cation. Similarly, other WebView APIs are instrumented, and corresponding fingerprints are extracted at runtime.

IV. DATASET

To study the impact of browser fingerprinting on hybrid apps, we conducted our study on a diverse set of Android apps. In particular, we curated apps from the following datasets:

- **AndroZoo Dataset** We downloaded over 60,000 apps (as of December 2022) from the AndroZoo dataset [26]. All downloaded apps were selected to be 5 MB or greater in size. AndroZoo contains a compilation of Android apps from several marketplaces, including the Google Play store. The AndroZoo dataset is updated daily with new apps from Google Play, every three days for F-Droid, and twice a month for the Anzhi play store. We explicitly confirmed this with AndroZoo's maintainers. Accumulating apps from the AndroZoo dataset provides a wide variety of apps since they belong to three different marketplaces. In our study, we are interested in hybrid apps containing at least one instance of WebView. Thus, to filter for hybrid apps, we first decompiled the apps in the dataset and examined the decompiled code for WebView-related method signatures. To further validate that these apps are hybrid, we applied our instrumentation framework to them and logged WebView-related method calls. We ended up with 37,623 apps that use at least one instance of WebView's APIs. The pie chart in Figure 3a provides the distribution of app categories.
- **Google Play Store** We downloaded 10,000 recent applications from the Google play store. Of these, 5,000 apps are accumulated as the top 500 apps each from the top 10 app categories. The remaining 5,000 apps were downloaded randomly. We applied the same strategy to filter the hybrid apps and removed apps already present in the Androzoo dataset, ending up with 3422 additional apps that use at least one instance of WebView. The pie chart in Figure 3b shows the categorization of these apps based on the categories.

  Apps for Manual Analysis: Among the popular apps, we selected *ten* apps for automatic as well as manual analysis. In particular, for the manual analysis, we created multiple (fake) accounts and observed HTTP headers like cookies, user-agent strings, and URLs for these accounts. The manual analysis aims to determine the information that can help identify a user uniquely over multiple devices or platforms. Table I lists these apps, along with the sensitive information they expose in their user agent, cookies, and custom headers.

Figure 3c and 3d provide the distribution of apps based on their size and release date. Apps in our dataset range from 5 MB to 600 MB in size, with an average size of 25.22 MB. Besides, these apps were released between Feb 2022 and Dec 2022. All of these applications were subsequently instrumented as described in Section III to collect the user agent strings, custom headers, and URLs. We further created scripts to automate the data collection process: All of our scripts are publicly available to researchers for replication purposes.

V. EVALUATION

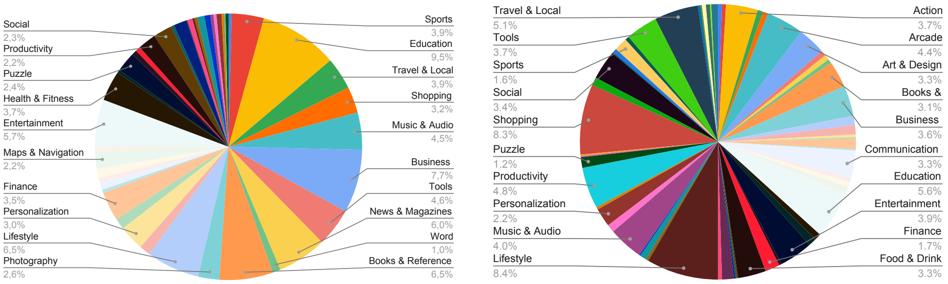
Multiple studies have been proposed for browser fingerprinting [11], [1], [20], [2] and Android hybrid app analysis [16], [14], [15], [17]. The most relevant recent work [11] performed a preliminary investigation on fingerprinting of mobile browsers. However, their work focused on full-fledged mobile browsers. In contrast, we aim to perform a large-scale study of fingerprints generated by hybrid Android apps. In particular, we are interested in information leakage, user tracking, and security implications arising from the bridge communication capabilities of hybrid apps. The bridge communication provides access from (potentially) untrusted web components of a hybrid app to the trusted native app's data and functionality. In this work, we explore how the web component of a hybrid app exploits these capabilities to expose information via fingerprinting. Besides, we identify the differences in fingerprinting between the stand-alone and the hybrid apps' browser. In summary, we find that hybrid apps reveal more information about the user than traditional browsers. Our experiments were designed to answer the following research questions:

RQ1 Does the hybrid browser release more information than other browsers?
RQ2 Is the hybrid browser susceptible to passive fingerprinting?
RQ3 What is the impact of the combination of Cookies and user-agents in information release?
RQ4 Can the hybrid browser's fingerprint infringe on the native app's security policies?
RQ5 What is the impact of unencrypted communication on the hybrid browser?

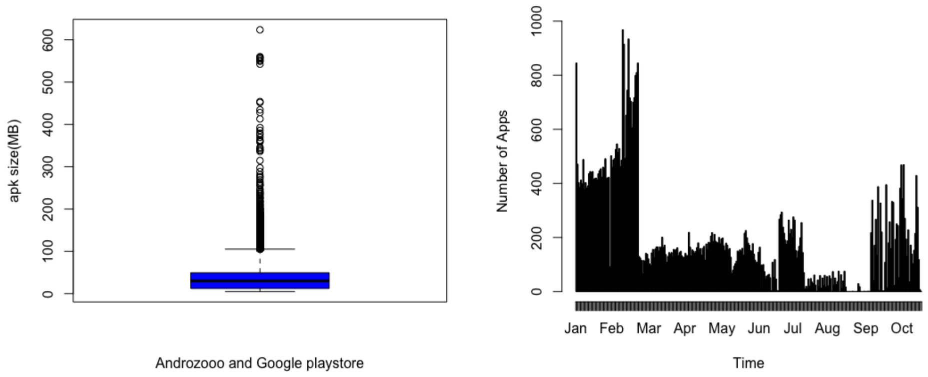
Our experiments were performed on a personal laptop with 16 GB RAM and a fourth-gen Intel Core i7-4500U processor running Windows 10.

a) ***RQ1: Privacy leakage unique to hybrid apps' browser.:*** Fingerprints in WebView are a good source of (potentially) privacy-sensitive information. For example, the hybrid app browser's fingerprint contains sensitive information such as the phone model and build number. The latter is sensitive information that can be leveraged to determine vulnerable devices and craft operating-system-specific attacks as observed by security analysts [27] and acknowledged by Google [28]. The desktop Chrome browser removed the build number in 2018 whereas the hybrid apps' browser includes this information in the user agent string up to this date.

To further improve user privacy, Chrome contains a privacy sandbox since version 93 (released on August 31, 2021). It



(a) AndroZoo Apps by Categories (1k apps) (b) Google Play Store Apps Categories (1k apps)



(c) Size distribution of apps in dataset (d) Time distribution of apps in dataset

Figure 3: App Categorization

allows the user to manually limit[1] leaking of sensitive information to protect against passive fingerprinting. However, no such configuration can be activated in hybrid apps' in-built browser. Table II shows the uniqueness of the fingerprints obtained on hybrid apps' in-built browser, the standalone Chrome browser, and the Chrome browser with sandboxing. The *uniqueness* brought by the privacy sandbox is 259 times lower than the unmasked fingerprint: The higher the uniqueness number, the worse it is for users' privacy.

To obtain the *uniqueness* of a browser fingerprint, we leverage *Cover Your TRACKS* [23], a research project to understand the uniqueness of browser fingerprints. It provides a uniqueness score to a fingerprint based on a large fingerprint database. We observed that fingerprints including the build number are highly unique; the uniqueness decreases significantly when removing the build number, and again drastically when limiting the phone model information.

Finding 1: *Hybrid apps' built-in browser permits more sensitive information leakage than the stand-alone browser. All hybrid apps in our dataset expose the build number and phone model in their fingerprints. This permissiveness stems from the inability to configure system-wide privacy policies.*

RQ2: Information leak via passive fingerprinting.*:* Like traditional browsers, Android allows WebView to transmit a user-agent HTTP header to the server, which can derive information from it. It is the app developers' responsibility to control the information they want to share with the server. As is, the web components (WebView) of hybrid apps indirectly inherit the same level of permissions as the shared components of the native side of the apps. Thus, by using the shared APIs, they potentially have access to sensitive device/user-specific information. During our manual analysis of the most popular apps from the Google play store, we observed an interesting mechanism to profile users based on the HTTP headers in the social media apps (with chat functionality) that use WebView to open in-app URLs. We exemplify our attack on the well-known social media app Instagram. Instagram's Android app leverages WebView to open an in-app URL/link, i.e., a link shared in a chat. We crafted a scenario where a curious (or malicious) user, Bob, wants to get some personal information such as the phone model, language, or ethnicity of a user Alice. Bob owns a server that can create account-specific links (e.g., server.com/Alice) and sends this link to Alice, and once Alice clicks on this link, it is displayed in the built-in WebView browser. Figure 4 shows the fingerprint and the sensitive information shared with Bob's server; Bob is able to obtain Alice's personal information, such as phone

---

[1] Via chrome://flags/#reduce-user-agent

Table I: Manually Analyzed Apps

App Name	Version	Category	Cookie	User agent	Custom headers
Instagram	229.0.0.17.118	Social	no	Phone model, build number, localization info, SDK, Android version, processor	no
Facebook	359.0.0.30.118	Social	no	Phone model, build number	no
Alibaba	7.48.1	Shopping	yes	Phone model, build number	unique user ID
Twitter	9.31.1	Social	no	Phone model, build number	no
LinkedIn	4.1.629.1	Social	no	Phone model, build number	no
Uber	4.361.10001	Maps and Navigation	no	Phone model, build number	no
QuuBe - Wholesale	6.5.1	Shopping	yes	Phone model, build number, UUID in the user agent	unique user ID
Flipboard	4.2.97	News & Magazines	no	Phone model, build number	no
Youtube	17.08.32	Video Players & Editors	no	Phone model, build number	no
DW Learn German	1.0.1	Education	no	Phone model, build number	no

Table II: Fingerprints from Various Browsers

Platform	Fingerprint	Uniqueness (1/X)
Hybrid apps' Browser	{Mozilla/5.0 (Linux; Android 9; SM-A505FN Build/PPR1.180610.011; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/99.0.4844.88 Mobile Safari/537.36}	X= 218256
Chrome Browser	{Mozilla/5.0 (Linux; Android 9; SM-A505FN) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/98.0.4758.87 Mobile Safari/537.36}	X = 218112
Chrome Browser with sandboxing	{Mozilla/5.0 (Linux; Android 10; K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.0.0 Mobile Safari/537.36}	X=838.98

model and language preferences.

As discussed in case study 1, the Instagram app, by default, sends the phone's model and build number, already providing more uniquely identifiable information than the stand-alone Chrome browser. On top of that, it also reveals the Android version (both OS and SDK), phone resolution, processor name, and localization information. Localization information is very sensitive for profiling users. We observed that the uniqueness of this information is very high (217923), which is detrimental to users' privacy.

This fine-grained information in the user-agent header renders the app vulnerable to passive fingerprinting, where an attacker can infer these user-agent headers by simply observing the traffic coming from a malicious URL shared through the chat. To mitigate the problem of passive fingerprinting, RFC9110 [21, ch. 10.1.5] disallows "advertising or other nonessential information within the product identifier". Instagram adds personally identifiable information to the contrary. In contrast to the stand-alone browser where the user can choose to hide this information, the user has no control over which information is shared once certain permissions are given to the Instagram app.

Finding 2: *Hybrid apps are susceptible to passive fingerprinting and often violate standard privacy policies. Famous apps like Instagram provide less to no control to their users over the amount of sensitive information released via web components.*

b) ***RQ3: Profiling Users via a combination of cookies and user-agent.*** *:* In the previous case studies, we demonstrated how users could be profiled based on user-agent strings. The situation becomes more severe when this information is combined with other mediums such as cookies; the combined information helps obtain a fine-grained profile of the user. For example, in the Alibaba app, the user's account ID (unique over multiple devices) is added to the cookies; thus, one can intercept the user ID and the phone model information obtained from the user-agent string to profile users' phone buying behavior. Note that the user's account ID stays the same over various devices/browsers, i.e., users can be uniquely identified over different service providers. Besides, the server can concretely infer sensitive information on the user, e.g., how many devices a user owns, how frequently users change their phone, and what the financial situation of a user is.

User profiling is also possible through HTTP ACCEPT-language headers. ACCEPT-language headers are used to determine the language preferences of the client. Generally, these headers are derived from the language preference of the user. For example, a user located in Switzerland and speaking German would have the accept language *CH-de*. Unfortunately, a user can be profiled based on her language preferences, e.g., identifying the user's origin, ethnicity, or nationality. Worse, if the user speaks more languages, with the combination of other fingerprintable information, the user can be uniquely identified. For example, a user speaking a combination of Russian and Turkmen languages could be profiled as Turkmenistan origin. However, users can hide this information on regular browsers through their settings or, better, use a privacy-compliant browser. Unfortunately, this is not possible for the hybrid browser as users cannot control the

```
Mozilla/5.0 (Linux; Android 9; SM-A505FN Build/PPR1.180610.011; wv)
AppleWebKit/537.36(KHTML, like Gecko) Version/4.0 Chrome/99.0.4844.88
Mobile Safari/537.36 Instagram 229.0.0.17.118 Android (28/9; 420dpi;
1080x2131; samsung; SM-A505FN; a50; exynos9610;en_DE; 360889116)
```

(a) Fingerprints

```
Instagram Version (Instagram 229.0.0.17.118, 360889116), Platform (Android)
Android SDK (28) and version (9), Phone model (samsung; SM-A505FN;),
Proccessor name (exynos9610) DPI and Resolution (420dpi; 1080x2131),
Locale (en_DE)
```

(b) Identifying Information

Figure 4: Fingerprint from Instagram

Table III: Apps including unique IDs into user-agent string

Package Name	App Name	Category
com.oddm.adpick	Adpick	Office
net.giosis.shopping.id	Qoo10 Indonesia	Shopping
net.giosis.shopping.cn.nonepush	Qoo10 APK 3.2.7	Shopping
Net.giosis.shopping.sg	Qoo10 - Online Shopping 6.5.1	Shopping
xyz.quube.mobile	QuuBe - Wholesale by Qoo10	Shopping
xyz.quube.shopping.tablet	QuuBe for Tablet	Shopping
mobile.qoo10.qpostpro	Qpost Pro 1.4.1	Shopping
mobile.qoo10.qstl20	Style Club 6.4.0	Shopping
com.alibaba.intl.android.apps.poseidon	Alibaba	Shopping
Com.accelainc.ihou.fr.droid	Illegal Romance 1.0.2	Adventure

settings of this browser.

Furthermore, we observed that various applications attach unique device IDs to the user-agent string, resulting in the direct identification of a user. To observe this behavior, we logged into the apps with multiple user accounts and observed the differences in the fingerprints. This manual analysis confirms this misconduct [21, ch. 10.1.5] in at least ten apps in our dataset. Table III presents the list of these apps alongside their categories. Apps with a similar name, e.g., Qoo10 Indonesia and Qoo10 APK 3.2.7, are from the same manufacturer but belong to different countries and have different privacy policies. Owing to the sheer volume of the dataset, it was not feasible to create multiple accounts for all the apps and relate fingerprints for this unique information. Table IV shows a sample of the fingerprints obtained from the devices containing unique device IDs. As is, the unique IDs are attached to the devices; they remain unchanged after even reinstalling the apps. Along with the unique device ID, these devices contain fine-grained information about the device attributes, such as build number, phone model, and Android version. Thus, one can directly relate a device to its attributes, and also build a temporal profile of the particular device, in case the device is used by another user.

Finding 3: *The combination of cookies and user agents links sensitive device and user-specific information. This information can be exploited to profile a user uniquely, such as identifying the origin and estimating the personal financial status. Besides, a few apps in our dataset attach their users' account IDs (unique for a user) to the cookies making their users uniquely identified over different devices.*

```
JavaScript:if(window.Application)
{
  Application.setDeviceUid(""APA91bG956w4WPzLIh
  DCHdcnIdbigwApzJzX-WFCkrKRcpJMr9Xw0kbAAxjBYj-
  f6UnVrfeMWRhuPlQIiv8np8733GgHzHm6QHLMeK1
  -InIkhWvxq9yjGb_i2a5WdxIQmaAl-QP3aHHIqK9XTGJiiPpJo
  _dXqkVNzQ"");
}
```

Listing 2: Setting device IDs through JavaScript

*c) **RQ4: JavaScript modifying Android objects.**:* As a part of our instrumentation framework, we instrument the *loadUrl* method to extract the originating URLs. On top of loading URLs *loadUrl* also provides functionality to load/execute a JavaScript code snippet directly. We also intercepted many cases where JavaScript modifies Java objects using bridge objects. A recent study [16] exposed instances of potentially untrusted JavaScript code interfering with Android objects. However, in several cases, the aim of such interference was unknown in that study. In this work, we identify a number of patterns where JavaScript transmits unique IDs to native Android objects. These unique IDs can be used as fingerprints for devices. For example, an app *com.a2stacks.apps.app57191abb7ab09* sets the user ID of the user as shown in listing 2, violating multiple security policies. First, the (potentially) unsafe web component violates the integrity of the native app by modifying its object, i.e., writing the device UID into a field. Second, the app may violate the Android privacy policies by assigning a unique device identifier without having asked for permissions.

Finding 4: *(Potentially) Unsafe web components infringe the integrity of a native app's object. Hybrid app web components (JavaScript) assign unique identifiers to the device for (potential) fingerprinting purposes via the Android bridge communication.*

*d) **RQ5: Unencrypted communication.**:* During our analysis of extracted URLs, we find various instances where unencrypted protocols such as HTTP are used to communicate secret information such as device IDs, IP addresses, Google

Table IV: Fingerprint showing unique ID

App	Category	Fingerprint
com.oddm.adpick	Office	`Mozilla/5.0 (Linux; Android 10; Android SDK built for x86 Build/QSR1.210802.001; wv)AppleWebKit/537.36 (KHTML, like Gecko)Version/4.0 Chrome/74.0.3729.185 Mobile Safari/537.36 AdpickEncrypted:GDPViCyiXnbcQgWnvAmIBusjAV43FvgPeawc/Xc5ayQW0rBy/oA8BUz4Vdmy9ITgwRQDnaI7BmZB#nXG5+MzNecK3HyqXv7P5/2u9yqMmkwrA/leTfsNeUZbmjvzj9D9m ECLyuBwl3lA8Sz 2dt4Ue1H1tT#n4mWgFssSh2n/eR1qpgnGRhc1cB2jqXtWuTW/cNQC#n`
net.giosis.shopping.id	Shopping	`Mozilla/5.0 (Linux; Android 11; Android SDK built for x86 Build/RSR1.210210.001.A1; wv)AppleWebKit/537.36 (KHTML, like Gecko)Version/4.0 Chrome/83.0.4103.106 Mobile Safari/537.36 Android_Gmarket Qoo10 ID_3.6.2_133(GMKTV2_ZlRnG1XAIzgwoC3OBe0hNjV4PfmyaC5RAIBqY+mkcipUGsSIiB19AyfIHQY1msEafG/xGz9RIS4=;AndroidSDK built for x86;11;en_US;2000010476)`
net.giosis.shopping. cn.nonepush	Shopping	`Mozilla/5.0 (Linux; Android 10; Android SDK built for x86 Build/QSR1.210802.001; wv)AppleWebKit/537.36 (KHTML, like Gecko)Version/4.0 Chrome/74.0.3729.185 Mobile Safari/537.36 Android_Gmarket Qoo10 CN NOPUSH_3.6.6_137(GMKTV2_/E/eowDAPJdLOH3or4b6kUZaqiQ9445kf50bcLzkkQeoFvJmnsEzdFnnyGmoyagfCYHYKlwCWP4=;AndroidSDK built for x86;10;en_US;2000000134)`
net.giosis.shopping.sg	Shopping	`Mozilla/5.0 (Linux; Android 10; Android SDK built for x86 Build/QSR1.210802.001; wv)AppleWebKit/537.36 (KHTML, like Gecko)Version/4.0 Chrome/74.0.3729.185 Mobile Safari/537.36 Android_Gmarket Qoo10 SG_6.5.1_269(GMKTV2_yQ+4mthiJO62KzrgMNh9rwIUgQVt5Aax6jISAXY3h++KFBJ4DO5 /YZdeiP3jYmD+hnf246qDDdk=;Android SDK built for x86;10;en_US;200007873;US;)`
com.accelainc.ihou.fr. droid	Adventure	`2NSsaFdT 60D1F74326F469CB__5DC12396C15AB57696B4A 69152169D 1.0.1 & Mozilla/5.0 (Linux; Android 10; Android SDK built for x86 Build/QSR1.210802.001; wv)AppleWebKit/537.36 (KHTML, like Gecko)Version/4.0 Chrome/74.0.3729.185 Mobile Safari/537.36`

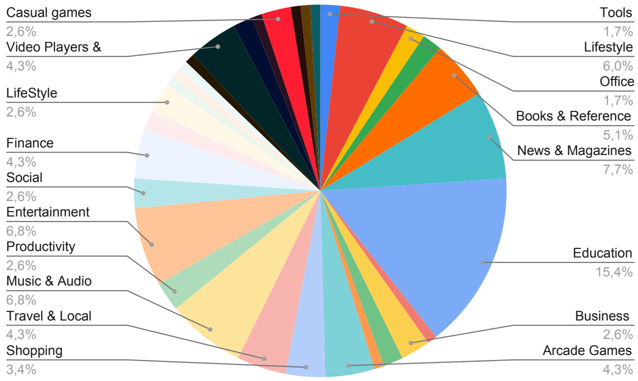


Figure 5: Unencrypted URLs by App Categories

ads user identifiers, and many other sensitive data. This is a severe problem, and unfortunately, 3922 applications from our dataset contain this flaw. Related work [16] has shown that the use of unencrypted communication is susceptible to simple man-in-middle attacks: An attacker can alter the server's response to an attacker-controlled web page without the user noticing any difference. Besides, the attacker learns the user's sensitive information by just observing the traffic; 281 apps share Google ads IDs, and 132 out of them also add IP addresses to the URLs. Interestingly, 214 of these 281 apps use URLs from the domain http://splash.appsgeyser.com domain, 28 from http://splash.appioapp.com, and 39 from http://ads.appioapp.com. Notably, all of these URLs belong to platforms (AppsGeyser and Appio) for creating Android apps, thus many other apps (not in our dataset) may be susceptible to unencrypted communication. Table V shows a list of ten apps that load at least one instance of an unencrypted URL. Figure 5 provides the distribution of apps using unencrypted URLs based on categories.

Table V: Ten Apps with Unencrypted URLs

Package Name	App Name	Hash
com.wWelcometoPurnia	Welcome to Purnia	004BDEAF41
com.wPBALogistics	PBALogistics APK	0094D388AB
com.wKPUKabKepulauanSelayar	KPU Kab Kepulauan Selayar	005F8F4E97
com.wsmile2	Smile APK 1.1	00F784BF5B
com.wAnEssayonManmoralessaysandsatires	An Essay on Man APK	0183B4DF5C
com.cultplaces	Cult Places	0665508043 1
com.wTrendyBotswana	Trendy Botswana	067999FD77
com.wProfDrMustafaKaratasSoruCevap	Mustafa Karataş ile Soru Cevap	06F06AFCB9
com.wTanksDecades	Tanks Decades	06F781FF93
com.wRapKlayBBJ	Rap Klay BB.J	0711A4A1AE

Finding 5: *9.6% of the apps in our dataset leak sensitive information via unencrypted communication protocols like HTTP. These URLs contain sensitive data such as device IDs, IP addresses, ad identifiers, locale information, and other sensitive data.*

VI. LIMITATIONS

CHARLIE is a dynamic instrumentation tool and relies on the instrumentation framework Frida to instrument the Android Framework and record the fingerprinting data. It inherits all the limitations of Frida, e.g., it is known to crash for the older version of Android apps[2]. Besides, to navigate through various app activities, i.e., for coverage, CHARLIE relies on the automated Android tester Monkey [25], and its coverage is limited to the activities visited by Monkey. Thus, CHARLIE misses Android components that Monkey does not explore.

VII. THREATS TO VALIDITY

Internal Validity: CHARLIE relies on existing dynamic analysis tools, and there are many automated testing tools. In particular, CHARLIE uses the Monkey tester, which might result in selection bias. We choose the Monkey tester as the research community widely uses it, and official Android document supports it. Another threat is related to the selection of our dataset, i.e., whether the chosen apps favor CHARLIE.

---

[2] https://frida.re/docs/android/

We mitigate this threat by selecting a large set of apps from the widely used AndroZoo dataset. Besides, we choose the most popular apps from the Google play store. One final threat is validating the results for the manually analyzed apps. To mitigate this threat, at least two authors of the paper cross-validated the analysis results.

External Validity: Threats to external validity relate to the generalization of our results, i.e., our results may not hold beyond the apps in our dataset. To mitigate this, we performed our study on a large set of apps from the widely accepted AndroZoo dataset and the most popular apps from the Google play store. Besides, the apps in our dataset belong to various categories, and the distribution over these categories is even.

VIII. LESSONS LEARNED

This work studies the application of browser fingerprinting in the Android WebView. Our study finds that the hybrid browser's fingerprints release more sensitive information than other browsers. In the followings, we summarize the lessons from our research:

- Fingerprinting is widely used with Android WebViews. The hybrid browser permits finer fingerprinting compared to other browsers. The permissiveness stems from the inability to configure system-wide privacy policies.
- The central problem lies with the non-user-centric design of the hybrid browser, i.e., the control of the WebView browser is with the developer, not the user. Policies to mask device-specific information for WebView browsers, like in full-fledged browsers, can help make it privacy-compliant.
- Many apps use unencrypted communication protocols via WebView. The majority of these URLs originate from unsafe third-party libraries. Developers should be cautious using such URLs with WebView.

IX. RELATED WORK

Fingerprinting in browsers has been studied for a little more than a decade. To the best of our knowledge, three large-scale studies have been conducted on browser fingerprints. The first study [1] showed how user-agents, list of plugins, and fonts available on a system can be used to fingerprint mobile devices. Their results showed that 83.6% of the user-agents strings are unique, hence, susceptible to fingerprinting. They coined the term *browser fingerprinting*, referring to the use of system information obtained from web clients as fingerprints. *AmIUnqiue* took it a step further and identified new attributes for fingerprinting such as HTML canvas elements. It also identified the most common attributes in fingerprinting for mobile devices. Oliver's thesis [11] showed that fingerprinting is "quite-effective" on mobile devices based on a preliminary investigation in susceptibility of mobile browsers towards fingerprinting. Our work is placed in the context of browsers embedded in hybrid apps. Hybrid-app browsers are customized by the developer and, in contrast to standalone browsers, users have little to no influence on its security and privacy policies. Therefore, these browsers are a fertile ground for profiling users through fingerprinting.

In a contrasting study, *HidingInTheCrowd* [2] studied the evolution of browser fingerprints over time. Their study shows that the number of unique fingerprints has reduced from the previous studies — more in the case of mobile browsers than desktop browsers. The fingerprints obtained from mobile browsers, in their study, present attributes having unique values and primarily use user-agent settings and HTML canvas elements. It conforms to Oliver's study [11], where it shows that a majority of mobile fingerprints are unique due to the presence of an unique identifier. This observervation also conforms with our study, where we have also obtained fingerprints which are also unique to users and devices.

Apart from these, earlier studies have also focussed on the sources of fingerprints. Acar *et al.*'s study [3] showed the use of HTML `canvas` elements in fingerprinting. Sources of fingerprinting also include WebGL [4], [5], Web Audio API [6], browser extensions [7], [8], [9], and CSS querying [29], among many others. Therefore, browser fingerprinting techniques have diversified their sources keeping in pace with evolution of the web. In comparison, we have confined our study to features in HTTP-headers in hybrid apps. Hybrid apps do not support browser extensions, and therefore, we have not considered these in our study. Also, we did not find other sources, such as canvas elements, WebGL resources in our study and choose to ignore these features.

The paper also overlaps with studies on privacy leakage in hybrid apps. Tiwari et al. [16] profiled privacy information leaked through the bridge interface. Rizzo et al. [14] studied the use of code injection attacks in WebView. Lee et al. [15] discovered the vulnerability of AdSDKs leaking sensitive information via *loadUrl*. Mutchler [17] conduced a large-scale study on the Android app ecosystem to detect vulnerabilities in hybrid apps. Their findings suggest that hybrid apps have at least one security vulnerability in the Android app ecosystem. Zhang [30] performed a large-scale study of Web resource manipulation in both Android and iOS WebViews. They discovered 21 apps with malicious intents such as collecting user credentials and impersonating legitimate parties. In comparison to all these works, we analyze the fingerprints obtained from the hybrid-browsers, and manually analyze the privacy-leakage thereof.

X. CONCLUSION

In this paper, we studied the fingerprints obtained in hybrid apps. To this end, we developed an instrumentation-based tool to record the user-agent strings and HTTP headers used in the webpage of the hybrid apps. Our study shows that hybrid apps are as susceptible to fingerprints as websites accessed on mobile browsers. However, the absence of mechanisms to enforce privacy policies makes it harder, if not impossible, for users to protect their privacy. Therefore, the recent advances in protecting privacy via fingerprinting do not translate into the realm of hybrid apps as the configuration remains in the hands of developers. Our study highlights the need for techniques to enforce privacy policies in hybrid apps.

REFERENCES

[1] P. Eckersley, “How unique is your web browser?” in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, ser. PETS’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–18.

[2] A. Gómez-Boix, P. Laperdrix, and B. Baudry, “Hiding in the crowd: An analysis of the effectiveness of browser fingerprinting at large scale,” in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, pp. 309–318. [Online]. Available: https://doi.org/10.1145/3178876.3186097

[3] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, “The web never forgets: Persistent tracking mechanisms in the wild,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 674–689. [Online]. Available: https://doi.org/10.1145/2660267.2660347

[4] Y. Cao, S. Li, and E. Wijmans, “(cross-)browser fingerprinting via os and hardware level features,” in *NDSS*, 2017.

[5] K. Mowery and H. Shacham, “Pixel perfect : Fingerprinting canvas in html 5,” 2012.

[6] S. Englehardt and A. Narayanan, “Online tracking: A 1-million-site measurement and analysis,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1388–1401. [Online]. Available: https://doi.org/10.1145/2976749.2978313

[7] O. Starov and N. Nikiforakis, “Xhound: Quantifying the fingerprintability of browser extensions,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 941–956.

[8] I. Sanchez-Rola, I. Santos, and D. Balzarotti, “Extension breakdown: Security analysis of browsers extension resources control policies,” in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 679–694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/sanchez-rola

[9] G. G. Gulyas, D. F. Some, N. Bielova, and C. Castelluccia, “To extend or not to extend: On the uniqueness of browser extensions and web logins,” in *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, ser. WPES’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 14–27. [Online]. Available: https://doi.org/10.1145/3267323.3268959

[10] P. Gupta, T. K. Wee, N. Ramasubbu, D. Lo, D. Gao, and R. K. Balan, “Human: Creating memorable fingerprints of mobile users,” in *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, 2012, pp. 479–482.

[11] J. Oliver, “Fingerprinting the mobile web,” Ph.D. dissertation, Master Thesis. London, UK: Imperial College London, 2018.

[12] “Webview,” https://developer.android.com/reference/android/webkit/WebView/, Google, 2021.

[13] Google, “Chromium webview browser,” https://developer.chrome.com/docs/multidevice/webview/, 2022.

[14] C. Rizzo, L. Cavallaro, and J. Kinder, “Babelview: Evaluating the impact of code injection attacks in mobile webviews,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 25–46.

[15] S. Lee and S. Ryu, *Adlib: Analyzer for Mobile Ad Platform Libraries*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 262–272. [Online]. Available: https://doi.org/10.1145/3293882.3330562

[16] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, “A large scale analysis of android — web hybridization,” *Journal of Systems and Software*, vol. 170, p. 110775, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301898

[17] P. Mutchler, A. Doupé, J. C. Mitchell, C. Kruegel, and G. Vigna, “A large-scale study of mobile web app security,” 2015.

[18] “Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers,” https://frida.re/docs/android/, FRIDA, 2022.

[19] (2023, March) Tool and dataset for fingerprinting the android hybrid web apps. Zenodo. [Online]. Available: https://doi.org/10.5281/zenodo.7728279

[20] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, “Browser fingerprinting: A survey,” *ACM Trans. Web*, vol. 14, no. 2, apr 2020. [Online]. Available: https://doi.org/10.1145/3386040

[21] I. E. T. F. (IETF), “Rfc 9110: Http semantics,” https://www.rfc-editor.org/rfc/rfc9110#name-user-agent, IETF, jun 2022.

[22] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 878–894.

[23] “Coveryourtracks,” https://coveryourtracks.eff.org/, EFF, 2014.

[24] E. F. Foundation, “A primer on information theory and privacy,” https://www.eff.org/deeplinks/2010/01/primer-information-theory-and-privacy, March 2023.

[25] Google, “Monkey tester,” https://developer.android.com/studio/test/other-testing-tools/monkey, 2022.

[26] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 468–471.

[27] “Research: Chrome for android reveals phone model and build,” https://wwws.nightwatchcybersecurity.com/2015/09/30/research-chrome-for-android-reveals-phone-model-and-build/, nightwatchcybersecurity, 2015.

[28] “Webview privacy issue,” https://bugs.chromium.org/p/chromium/issues/detail?id=494452, Chrome, 2015.

[29] N. Takei, T. Saito, K. Takasu, and T. Yamada, “Web browser fingerprinting using only cascading style sheets,” in *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, 2015, pp. 57–63.

[30] X. Zhang, Y. Zhang, Q. Mo, H. Xia, Z. Yang, M. Yang, X. Wang, L. Lu, and H. Duan, “An empirical study of web resource manipulation in real-world mobile applications,” in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC’18. USA: USENIX Association, 2018, pp. 1183–1198.

[31] P. S. Kochhar, D. Wijedasa, and D. Lo, “A large scale study of multiple programming languages and code quality,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 563–573.

[32] R. Monat, A. Ouadjaout, and A. Miné, “A multilanguage static analysis of python programs with native c extensions,” in *Static Analysis*, C. Drăgoi, S. Mukherjee, and K. Namjoshi, Eds. Cham: Springer International Publishing, 2021, pp. 323–345.

[33] J. Lerch, J. Späth, E. Bodden, and M. Mezini, “Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths,” in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’15. IEEE Press, 2015, pp. 619–629. [Online]. Available: https://doi.org/10.1109/ASE.2015.9

[34] S. Lee and S. Ryu, *Adlib: Analyzer for Mobile Ad Platform Libraries*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 262–272. [Online]. Available: https://doi.org/10.1145/3293882.3330562

[35] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, *Alias Analysis for Object-Oriented Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 196–232. [Online]. Available: https://doi.org/10.1007/978-3-642-36946-9_8

[36] S. Lee, H. Lee, and S. Ryu, “Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 127–137. [Online]. Available: https://doi.org/10.1145/3324884.3416558

[37] J. Whaley and M. Rinard, “Compositional pointer and escape analysis for java programs,” in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 187–206. [Online]. Available: https://doi.org/10.1145/320384.320400

[38] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, “Efficient construction of approximate call graphs for JavaScript IDE services,” in *ICSE*, 2013. [Online]. Available: files/ICSE-2013-Approximate.pdf

[39] S. Li and G. Tan, “Finding bugs in exceptional situations of jni programs,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS ’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 442–452. [Online]. Available: https://doi.org/10.1145/1653662.1653716

[40] G. Kondoh and T. Onodera, “Finding bugs in java native interface programs,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA ’08. New York, NY, USA:

Association for Computing Machinery, 2008, pp. 109–118. [Online]. Available: https://doi.org/10.1145/1390630.1390645

[41] U. P. Khedker, A. Sanyal, and A. Karkare, "Heap reference analysis using access graphs," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, pp. 1–es, nov 2007. [Online]. Available: https://doi.org/10.1145/1290520.1290521

[42] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: Static analysis framework for android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 250–261. [Online]. Available: https://doi.org/10.1145/2970276.2970368

[43] A. Tiwari, S. Groß, and C. Hammer, "Iifa: Modular inter-app intent information flow analysis of android applications," in *Security and Privacy in Communication Networks*, S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, Eds. Cham: Springer International Publishing, 2019, pp. 335–349.

[44] H. Illous, M. Lemerre, and X. Rival, "Interprocedural shape analysis using separation logic-based transformer summaries," in *Static Analysis*, D. Pichardie and M. Sighireanu, Eds. Cham: Springer International Publishing, 2020, pp. 248–273.

[45] S. Li and G. Tan, "Jet: Exception checking in the java native interface," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 345–358. [Online]. Available: https://doi.org/10.1145/2048066.2048095

[46] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[47] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," 2021.

[48] P. D. Schubert, B. Hermann, and E. Bodden, "Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Møller and M. Sridharan, Eds., vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:31. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14045

[49] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, "Ludroid: A large scale analysis of android—web hybridization," in *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2019, pp. 256–267. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SCAM.2019.00036

[50] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of Node.js applications," in *Proc. 30th International Symposium on Software Testing and Analysis (ISSTA)*, July 2021.

[51] P. Cousot and R. Cousot, "Modular static program analysis," in *Compiler Construction*, R. N. Horspool, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 159–179.

[52] Z. Li, X. Qi, Q. Yu, P. Liang, R. Mo, and C. Yang, "Multi-programming-language commits in oss: An empirical study on apache projects," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 219–229.

[53] S. Buro, R. L. Crole, and I. Mastroeni, "On multi-language abstraction," in *Static Analysis*, D. Pichardie and M. Sighireanu, Eds. Cham: Springer International Publishing, 2020, pp. 310–332.

[54] M. Grichi, E. E. Eghan, and B. Adams, "On the impact of multi-language development in machine learning frameworks," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 546–556.

[55] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps," in *Engineering Secure Software and Systems*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2016, pp. 72–88.

[56] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 567–577. [Online]. Available: https://doi.org/10.1145/1993498.1993565

[57] "Rcpp for seamless r and c++ integration," 2022. [Online]. Available: http://www.rcpp.org

[58] A. Turcotte, E. Arteca, and G. Richards, "Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 16:1–16:32. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10808

[59] "Rpy2: Python in r," March 2022. [Online]. Available: https://rpy2.github.io/doc/v2.9.x/html/index.html

[60] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about android interoperations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 223–233.

[61] "Webview privacy issue," https://bugs.chromium.org/p/chromium/issues/detail?id=494452, Chrome, 2015.

[62] S. Exchange, "Why are multiple programming languages used in the development of one product or piece of software?" March 2022.

[63] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web (TWEB)*, vol. 14, no. 2, pp. 1–33, 2020.

[64] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Fingerprinting mobile devices: A short analysis," in *CIEL 2017-6ème Conférence en IngénieriE du Logiciel*, 2017.

[65] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. C. Freiling, "Fingerprinting mobile devices using personalized configurations." *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 1, pp. 4–19, 2016.

[66] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 541–555.

[67] "Research: Chrome for android reveals phone model and build," https://wwws.nightwatchcybersecurity.com/2018/12/25/chrome-browser-for-android-reveals-hardware-information/, nightwatchcybersecurity, 2018.

[68] "User-agent reduction," https://developer.chrome.com/docs/privacy-sandbox/user-agent/, Google, 2021.

[69] "User-agent," https://datatracker.ietf.org/doc/html/rfc7231#section-5.5.3, IETF, 2014.

[70] "Targeting and exploiting android devices via hostile networks," https://wwwsnightwatchcybersecuritycom.files.wordpress.com/2016/11/bsides-philly-2016.pdf, nightwatchcybersecurity, 2014.

[71] Oracle, "oracle/graaljs: A ecmascript 2021 compliant javascript implementation built on graalvm. with polyglot language interoperability support. running node.js applications!" [Online]. Available: https://github.com/oracle/graaljs

[72] A. Tiwari, J. Prakash, S. GroB, and C. Hammer, "Ludroid: A large scale analysis of android – web hybridization," in *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2019, pp. 256–267. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SCAM.2019.00036

[73] M. Foundation. (2022) User-agent. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_agent