Not Your Average App: A Large-scale Privacy Analysis of Android Browsers

Amogh Pradeep
Northeastern University
Boston, USA

Alvaro Feal
IMDEA Networks Institute /
Universidad Carlos III de Madrid
Madrid, Spain

Julien Gamba
IMDEA Networks Institute /
Universidad Carlos III de Madrid
Madrid, Spain

Ashwin Rao
University of Helsinki
Helsinki, Finland

Martina Lindorfer
TU Wien
Vienna, Austria

Narseo Vallina-Rodriguez
IMDEA Networks Institute /
AppCensus Inc.
Madrid, Spain

David Choffnes
Northeastern University
Boston, USA

ABSTRACT

The privacy-related behavior of mobile browsers has remained widely unexplored by the research community. In fact, as opposed to regular Android apps, mobile browsers may present contradicting privacy behaviors. On the one hand, they can have access to (and can expose) a unique combination of sensitive user data, from users' browsing history to permission-protected *personally identifiable information* (PII) such as unique identifiers and geolocation. On the other hand, they are in a unique position to protect users' privacy by limiting data sharing with other parties by implementing ad-blocking features.

In this paper, we perform a comparative and empirical analysis on how hundreds of Android web browsers protect or expose user data during browsing sessions. To this end, we collect the largest dataset of Android browsers to date, from the Google Play Store and four Chinese app stores. Then, we develop a novel analysis pipeline that combines static and dynamic analysis methods to find a wide range of privacy-enhancing (e.g., ad-blocking) and privacy-harming behaviors (e.g., sending browsing histories to third parties, not validating TLS certificates, and exposing PII—including non-resettable identifiers—to third parties) across browsers. We find that various popular apps on both Google Play and Chinese stores have these privacy-harming behaviors, including apps that claim to be privacy-enhancing in their descriptions. Overall, our study not only provides new insights into important yet overlooked considerations for browsers' adoption and transparency, but also that automatic app analysis systems (*e.g.,* sandboxes) need context-specific analysis to reveal such privacy behaviors.

KEYWORDS

android, privacy, mobile browsers

Proceedings on Privacy Enhancing Technologies 2023(1), 29–46

https://doi.org/10.56553/popets-2023-0003

1 INTRODUCTION

Mobile browsers (*i.e.,* apps that allow users to visit websites) are complex, powerful, and poorly understood software systems that account for 55% of global website visits [119]. Their critical role as one of the primary gateways to the web, and the rich set of features that they support, make mobile browsers a particularly interesting platform to study from a privacy perspective. On the one hand, mobile browsers can *enhance* user privacy in unique ways by implementing features such as blocking web trackers and advertisers, enforcing secure network protocols wherever possible, and minimizing personal data exposure [75, 81, 96, 97, 130]. However, they can also inflict privacy harms by harvesting and exposing permission-protected information such as unique identifiers or user geolocation to third parties (as is commonly found in non-browser apps), or indirectly by making them available to website scripts via JavaScript APIs. Further, they may expose browser-specific sensitive data such as users' browsing history or credentials to third-parties due to poor design choices or a need to generate revenue, potentially at the expense of user privacy [42, 108, 113, 130].

Despite their potential for harm, the research community has largely overlooked the privacy threats inherent to mobile browsers. Early studies focused on a small set of browsers [74, 128] and identified isolated cases of "privacy protecting" browsers deceiving their userbase and abusing their access to personal and browsing data for tracking purposes [71, 102]. In this paper, we augment the state-of-the-art by conducting the first large-scale, systematic, and multidimensional analysis of the privacy behavior of 424 Android browsers available in public app markets (including the Google Play Store and four Chinese markets) and others pre-loaded by certain phone vendors. Specifically, we study and characterize: (1) how mobile browsers help or harm users' privacy during the course of web browsing sessions; (2) what additional permission-protected personal data mobile browsers collect and share with other parties, and the implications of such data collection; and (3) how the combination of these behaviors impacts the overall privacy disposition of the mobile browsers in our dataset.

While there is a significant amount of work on mobile app privacy space in general, the study of mobile *browsers* poses unique

methodological challenges that we address in this work. First, neither the Android OS nor app markets provide a comprehensive way to determine whether an app is a mobile browser. To address this challenge, we combine code inspection and dynamic analysis methods to identify those apps that match our definition of a mobile browser. Second, both browsers and websites expose data to other parties while in use, and simple analysis of network traffic traces is insufficient to distinguish whether the data was exposed by a browser or a website. To address this, we develop a novel browser analysis pipeline that uses webpage replays and a *baseline browser* to attribute data collection to either the website or the browser itself. We further use a combination of static and dynamic analysis techniques to identify which browser component (for instance, a third-party library) is responsible for data collection and gather actual evidence of these behaviors. Third, browsers can implement privacy-enhancing features to protect users from harm (*e.g.,* use of TLS encryption, limited access to JavaScript APIs that retrieve sensitive information, blocking websites from contacting third parties), but there are limited standard benchmarks to test for them. To overcome these challenges, we build a new test suite that addresses this need, along with instrumentation and automation to capture and study browsers' behaviors automatically and at scale.

We use the above methods to analyze a dataset of 424 browsers and observe the following:

- We find that 65% of browsers enhance privacy by blocking tracking scripts by default. Similarly, 51% of browsers block scripts that access protected JavaScript APIs.
- We see that most browsers do not default to HTTP**S** (only 2% of browsers do so) and that 10% of browsers do not properly validate TLS certificates, making them vulnerable to person-in-the-middle attacks.
- We find that 63% of browsers contain at least one third-party library related to advertisement and tracking, and that these libraries are often responsible for browsers requesting "dangerous" permissions. Furthermore, our run-time behavior analysis shows that 32% browsers share PII with other parties over the Internet, and that 19% browsers do the same for browsing history. While 14% of these browsers share this information for a specific feature (such as web search APIs), we find that 3% send it alongside personal data (thus harming user privacy). We also show that browsers often share both resettable and non-resettable identifiers with third-party servers across the Internet, including four instances of ID bridging, a practice that completely defeats the use of resettable identifiers.
- We conduct a multidimensional analysis of individual browsers to understand their overall privacy disposition. We find that few browsers uniformly improve privacy (*e.g.,* FOSS Browser), while many exhibit multiple harms (*e.g.,* Yandex, Baidu, Opera *etc.*). We also find mixed behaviors: of the 276 (65%) browsers that block tracking content, we see that 70% also allow tracking requests, 23% expose PII, 14% share browsing history, and 7% fail to validate certificates.

Our study has important implications for (1) end users who adopt a non-default web browser on Android, and for (2) automatic app analysis processes. For the former, our results can help guide their decisions about which browser to install from a privacy standpoint. In fact, understanding the privacy risks of the mobile browser ecosystem is critical in the EU as Google Chrome is no longer the default browser on Android and EU citizens can choose any browser when configuring their device [43, 52]. For the latter, we show that existing sandboxes may be ineffective for studying the privacy risks of mobile browsers, and they can benefit from our novel methodology to analyze browser apps and gain actual visibility into their behavior. We reported our findings to Google, which is currently investigating potential corresponding breaches of their policies. We also responsibly disclosed observed security vulnerabilities to developers. To support reproducibility and foster further research in this area, we make our code and data publicly available at https://github.com/NEU-SNS/mobile-browser.

2 THREAT & PROTECTION MODELS

This section describes our privacy threat and protection models. Our models are motivated by the fact that mobile browsers occupy a privileged position in terms of the type of data they can access: (1) all Android devices are expected to have at least one mobile browser [16, 51]; (2) like other apps, mobile browsers can access permission-protected device sensors, system resources and unique identifiers; and (3) being endpoints for web traffic, they have access to all data from web browsing *e.g.,* page content or browsing history. Note that our study focuses on browser behavior in the default mode, as opposed to "safe", "private", or "incognito" modes. Previous work has shown that these modes have their own security and privacy issues [13], and that they do not prevent browsers from collecting user data [50, 116].

2.1 Privacy Threat Model

We assume that a benign browser should render webpages, follow best practices for connection security, and adhere to data minimization principles when it comes to exposing user data. We acknowledge that there may not be such a browser, but we assume one for the sake of comparing against our threat model. We consider that a browser is "privacy harmful" if it deviates from this benign browser model by exhibiting any of the following behaviors:

Data dissemination not required for page rendering. Browsers may collect and share sensitive data (*e.g.,* location, unique identifiers), with first parties (the browser developer) or third parties (*e.g.,* data brokers, advertisers, analytics companies). Such sharing might be required to implement site features (*e.g.,* geofencing or localized search results); or used for secondary purposes (*e.g.,* for monetization through ads and tracking).

In this paper, sensitive data includes *personally identifiable information* (PII) and users' browsing history. Specifically, we consider the following data types to be PII: IMEI, Advertising ID (AdID), Android ID, MAC address, and geolocation. Note that the IMEI, Android ID, and MAC addresses are non-resettable IDs, while the AdID can be reset by users. We also consider the list of installed apps to be sensitive, as it can be used to profile users [115]. For browsing history leaks, we consider cases where over half of the websites visited are transmitted to another party over the Internet. While browsers may share the visited websites as part of their functionality (*e.g.,* URL safety checks [67, 101]), this data can

also be abused to build unique and stable identifiers for users, even across devices [28].

Website manipulation. Browsers may modify (*i.e.,* replace or add) content in a way that can harm user privacy. Examples of this behavior are replacing referral links [71] or injecting content and web elements associated with advertising companies that might collect user data (*e.g.,* advertisements [21, 121]).

Poor connection security. A failure to validate TLS certificates exposes users to person-in-the-middle attacks [107]. This can allow any arbitrary actor in the network to intercept all traffic, thus damaging the user's privacy (as the data can be observed) and security (data can be dropped or modified by the attacker).

2.2 Privacy Protection Model

We consider in our privacy protection model the following privacy-enhancing behaviors that are not implemented in the idealized benign browser behavior described above:

Blocking tracking domains. A browser that blocks connections to advertising and tracking services and scripts can potentially prevent third parties from collecting extensive data on users [11, 12, 42, 129]. We consider a browser to implement such privacy-protecting behavior if we can find evidence of blocked connections to such services.

Limiting access to WebAPIs. Websites may request access to sensitive data and sensors via JavaScript APIs (*e.g.,* the geolocation API). A privacy-protecting browser would block such requests by default. This includes cases where access is blocked until the user gives explicit permission to allow access.

Upgrading connection security. Many websites support both HTTP and HTTPS, with the latter being preferred for privacy and security. Privacy-protecting browsers should always upgrade to HTTPS when supported by the server.

3 MOBILE BROWSER DATASET

As most app stores do not consider a browser as a specific app category, we need to develop a sound method to automatically collect and identify mobile browsers at scale. However, this task is not trivial. Many Android apps rely on WebViews to show online content (*e.g.,* ads [64]) or provide in-app browsing functionality often part of social networks [78, 133]. Thus we need a way to differentiate those from actual browsers. In this section, we present our definition of a functional browser and describe our methodology to obtain a representative dataset of browsers from different sources.

3.1 Definition and Installation Sources

What is a browser? We define a browser as an app capable of accessing arbitrary websites using HTTP/S URLs. More specifically, these apps must declare this capability to the Android OS via *Intent filters* that handle *URL schemes*. The Android OS uses this same Intent-based filtering to display available browser options when users click a URL, supporting our definition. Our definition rules out both web-based mobile apps that render app-specific web pages and apps that render webpages through links within them (via WebViews), as they do not allow users to enter *arbitrary* URLs.

Where are browsers installed from? Android allows users to choose a default browser, which can either come pre-installed on the device, or can be downloaded by the user from the Google Play store. Android can also be configured to install apps from sources outside of Google Play. We compiled a list of such alternate stores and their corresponding popularity metrics (where available) and found that Chinese app stores stood out due to their wide adoption [18]. Thus, in this study, we include apps from these different sources: the Google Play Store, four popular Chinese app stores, and pre-installed browsers present in Android devices from multiple vendors. We believe this combination of sources captures apps used by most Android users worldwide.

3.2 Data Collection and Filtering

Browser collection. Collecting browsers from the sources listed above is challenging because most app stores lack a "browser" category. Thus, we use multiple strategies to collect *potential* browsers and filter them post collection to remove non-browsers. We collected apps from all sources in September 2021. First, we curate a list of 266 well-known browsers from multiple online sources including newspapers, blogs, and prior work [87]. We download the latest versions of these browsers from Google Play (accessed from the EU). For two Chinese stores, 360 [1] and Tencent [10], we use the search term "browser" in Chinese and collect all apps found with it. The other two Chinese stores, Anzhi [3] and AppChina [4] provide a dedicated "browser" category, thus we collect all apps listed here. Lastly, we include potential browsers for a dataset of pre-installed apps gathered by Gamba *et al.* [51]. For this set, we look for the string "browser" in the package name and app name (in the Android Manifest) to identify potential browsers.

Filtering non-browsers. Our data collection strategy might yield apps that are non-browsers, *e.g.,* the file explorer "Moto File Manager" passes our string-based filtering as its package name (`com.lenovo.FileBrowser2`) contains the word *browser*. To filter such cases, we need a test that reliably determines whether a potential browser is consistent with our definition of a browser (§3.1). To this end, we conduct a two-stage testing procedure on each app to identify browsers. First, we determine whether the app successfully installs on a test device running Android 11. If so, we then automatically trigger an *Intent* through the Android Debugging Bridge (adb) tool to drive it to open a test URL. It is possible for an app to handle this Intent but fail to load webpages due to run-time errors, or bad Intent-handling logic; we thus validate this approach using dynamic analysis techniques in §4.2. After the above filtering, our final dataset contains 424 browsers. To further contextualize our results, we group these browsers into three categories based on their origin: 266 browsers published on *Google Play*, 157 browsers published in *Chinese stores* and 5 browsers that are pre-installed. This last number is small because many pre-installed apps cannot run on our test device due to native or vendor-specific dependencies [29, 51]. If we fetch the same browser from two different origins (*e.g.,* Google Play and a Chinese store), we include it in both groups for our analysis.

3.3 Identifying Unique Browsers

Uniqueness across origins. We want to avoid over-reporting by including the same browser (collected from different sources) several times in our analysis. Grouping two apps can have certain

Table 1: Most popular browsers on Google Play (100M+ downloads), Anzhi (1M+ downloads), and AppChina (Top 5).

Name	# DL GPlay	# DL Anzhi	Top5 AppChina
Google Chrome	10B+	-	
UC Browser	1B+	200M+	•
Mozilla Firefox	100M+	1M+	
Opera Browser	100M+	7M+	
Yandex Browser	100M+	-	
UC-Mini	100M+	-	
Phoenix Browser	100M+	-	
Baidu	5M+	100M+	•
Baidu Browser	-	80M+	•
Hao 123	-	10M+	•
Opera Extreme	-	4M+	
Baidu Express	-	1M+	
QQ Browser	-	-	•

pitfalls due to the lack of robust methods for author attribution on Android. While Google Play ensures that package names are unique *within the store*, this is not the case for apps from a different origin. Thus, a package name on Google Play might be used by a different app on another store (*e.g.,* a Chinese store). In fact, prior research showed that apps sharing package names could be repackaged and thus may have no relation to the original developer [73]. Further, in the case of pre-installed browsers, any manufacturer can modify and pre-install any open-source browser app with the default package name, *i.e.,* `com.android.browser` [51].

Our key observation is that every app must be signed with a certificate to be installed on an Android device [56, 60]. Thus, we rely on certificate information to distinguish two apps with the same package name (assuming that the same app would not be signed by two different certificates). We note that identifiers within self-signed certificates (*e.g.,* the `Subject` field), may be forged [83]. Therefore, we do not associate a certificate with a particular developer, but rather use information relating to the (unforgeable) private key to identify unique developers. Specifically, we use the package name and SHA-256 of the signing certificate combination as a unique identifier for each browser app. This allows us to identify browser apps with the same package name that are potentially implemented by different entities, and thus might behave differently. We account for different signatures for the same app by looking at app hashes after stripping certificate information from them and deduplicating matches. We argue that our technique is a reasonable heuristic (without ground truth) to uniquely identify browsers across origins.

Uniqueness of browsers. In our dataset, the majority (94%) of browsers with the same package name are signed by the same certificate. However, we also find browsers with the same package name being signed with *different* certificates, up to 3 different ones (*e.g.,* `com.baidu.browser.apps` found in different Chinese stores). Anecdotally, we also find versions of Firefox signed with different certificates in Chinese stores and the Google Play Store. This shows that some apps might be developed by different companies depending on the store where they are available.

3.4 Characterization of Browsers

Dataset statistics. Our final dataset contains 424 browsers with target API levels ranging from 5 (Android 2.0) to 30 (Android 11) with a median of level 24 (Android 7.0). While some of these target API levels might seem outdated, we note that for every unique browser we fetch the latest available version. This suggests that pre-loaded browsers and those distributed through app stores are not kept up to date, and that they might not benefit from the latest privacy-enhancing techniques and security standards.

The size of our dataset (424 browsers) indicates that there is a diverse and vibrant market for mobile browsers, providing a wide range of features. To better understand the advertised features and compare them with actual browser behavior observed during dynamic analysis, we extract and use as a proxy the most frequently used words in their app market descriptions (63% of our dataset). We find that 44% of the browsers advertise themselves as *high performance* browsers—using words such as "Fast", "Quick" in their description, 42% use words related to *privacy and security*, 26% advertise their *low cost*—with words such as "Free", and 2% claim to be *child-friendly* browsers. Regarding their market share, we find that 56 browsers have over 1M downloads on Google Play, and 6 browsers have over 100M downloads. More broadly, we find 4 browsers with fewer than 10k downloads, 65 apps have between 10k and 500k downloads, and 85 apps have over 500k downloads. While we do not have a download figure for all of Chinese stores, both Anzi and AppChina have a browsers category in which the available apps are ordered by popularity. The differences between these stores and Play is notable as the five most popular browsers on AppChina are the QQ Browser, UC Browser and three products from Baidu: Baidu Search, Baidu Browser and Hao 123. In the case of Anzhi, browsers with over 1M downloads include: UC Browser, Baidu Search, Hao 123, Opera Browser, Opera Extreme, Baidu Express, and Mozilla Firefox. We list the most popular browsers published on Google Play, Anzhi and AppChina in Table 1. These significant differences across stores, combined with the large numbers of estimated users (*e.g.,* QQ Browser and UC Browser accounting for more than a half billion monthly active users [72]), highlight the importance of including Chinese stores in our analysis.

3.5 Browser Engine Attribution

Web browsers are complex software that requires substantial engineering effort to develop from scratch. While we cannot know for certain why there are so many browsers in the Android ecosystem (424 according to our count), one hypothesis is that many of them are built atop existing open-source browser engines (*e.g.,* Chromium, Gecko) or are built as a WebView wrapper.[1] To investigate whether this hypothesis holds, we conduct a multi-facet analysis to infer whether a browser uses an underlying browser engine or is implemented using a WebView component provided by the Android OS [26, 120].

Class implementations. We search for standard classes for implementations of WebViews, Chrome, Firefox, and ChromeCustomTabs. We note that, as any static analysis technique, this might

[1]WebView objects allow developers to display web content on an activity layout. However, it lacks some of the features of fully-developed browsers like Chromium.

lead to false positives because of legacy and dead code. *WebView*-based browsers need to implement the `android.webkit.WebViewClient` class to function, so we search for these implementations and find that 378 (89%) browsers use them. Similarly, we find that 12 (3%) implement Chrome related classes (`org.chromium.chrome.browser.ChromeTabbedActivity`) and that 2 (<1%) implement Gecko/Firefox related classes (`org.mozilla.geckoview.GeckoSession`). We also look for ChromeCustomTabs implementations (`androidx.browser.customtabs.CustomTabsIntent`), which could in theory be used to implement a browser [26, 120], but found no cases of it. We are unable to attribute the remaining 110 (26%) to any of these engines.

X-Requested-With header. We complement the results from our static analysis with looking at the headers sent during run-time requests made to test websites. One of our key observations is that *WebView*s set a *X-Requested-With* header to the package name of the app while loading pages [34]. We see that 300 (71%) browsers send this header for all test URL requests; *i.e.,*they use *WebView*s to render webpages users request. All but 17 of the *X-Requested-With* senders are also identified as *WebView* implementers, providing a high level of cross-validation. Manual inspection revealed that code obfuscation (preventing us from finding `android.webkit.WebViewClient`) is the likely reason for inconsistencies.

User-Agent strings. We look at the *User-Agent* strings advertised by browsers and match them to possible underlying engines. However, we can only attribute 84 to *WebView*s with this technique. Note that this technique has its own limitations, as *User-Agent* strings can easily be spoofed by the developers.

Code similarity. As observed by related work, browser engines are likely written in native code and included as a shared library for performance reasons [128]. Thus, as a last step, we tried to rely on the code similarity between every pair of browsers in our dataset to understand if they share the same browser implementation. Specifically, we compared the native libraries that are included in the apps using BinDiff [37, 135], but did not find conclusive results.

4 METHODOLOGY

The goals of our analysis are: identifying privacy-harmful and -enhancing behaviors in mobile browsers as outlined in §2, determining the root causes for the behaviors we observe, and determining the impact of such behaviors on users' privacy. To meet these goals, we rely on the complementary strengths of both static code analysis and dynamic analysis techniques.

We note that static and dynamic analysis techniques have limitations when used on their own. Static analysis can cover many different execution paths without running apps, but it may produce false positives resulting from legacy or dead code, and produce false negatives due to apps leveraging code obfuscation or loading code dynamically. Dynamic analysis reveals the impact of real code execution paths at run time, but has limited coverage, can be defeated by anti-testing methods and provides a lower-bound of all browser behavior due to its inability to trigger all code paths with existing fuzzing methods [32]. The combination of both methods gives us better visibility for understanding browser functionality than using any one technique independently. In addition to these techniques, when we identify potentially privacy-harming behavior with dynamic analysis, we manually analyze the code to better understand and validate the logic triggering it.

4.1 Static Analysis

Static code analysis allows us to understand the browsers' potential for privacy-invasive behavior. Namely, this analysis reveals the types of data that an app can access (*i.e.,* requested permissions), the types of third parties that are integrated in browsers (*i.e.,* SDKs), and whether these SDKs are piggybacking on the permissions requested by the browser to access sensitive data for secondary purposes.

Permission analysis. The Android Open Source Project (AOSP) implements a permission model to restrict access to some of its features and sensitive resources (*e.g.,* the user's location, or SMS messages), to protect user privacy and security [56]. However, these permissions are shared among the host app embedded third-party SDKs. These SDKs might access personal data for secondary purposes. This might also lead to apps requesting more permissions than strictly necessary [23, 49], as SDKs often need access to data that is not necessary for the functioning of the app or to SDKs leveraging the set of permissions of the host app to access sensitive resources (potentially without user consent).

Thus, to assess the privacy risks of mobile browsers, we parse each browser's manifest to extract the requested permissions. This approach provides an upper bound of permission protected data that can be shared with other parties. Then, we map API calls (*i.e.,* functions called within the browser's code) to the AOSP permissions that protect them in order to gain a more fine-grained perspective. Note that Google does not include such a mapping in their documentation. Therefore, we leverage three complementary sources: (1) the mappings implemented in Android Studio (Android's official IDE), which contains scripts to warn developers if they use an API without requesting the associated permission [54]; (2) mappings extracted by parsing the AOSP source code to extract the methods that use the `@RequiresPermission` annotation, part of the AndroidX and Android support libraries [58, 59]; and (3) the mappings generated by prior work (Axplorer [24]) for older Android versions (Android 5 and below). We acknowledge that, while we update the mappings released by prior work, our mappings might still be incomplete (something that we cannot measure due to the lack of ground truth).

Third-party libraries. Android apps often rely on third-party libraries (or SDKs) to integrate functionality offered by third-party entities, *e.g.,* for A/B testing, analytics, or advertisements [108]. We use LibRadar [90] to identify third-party libraries in browser code but, as its library fingerprints and library-to-company (LTC) mappings are outdated [47], we augment LibRadar's LTC mappings with data from Exodus [44] and other information gathered through online resources and open-source intelligence. We further use domain knowledge to identify the purpose of the library as well as the company behind the service, and use it to characterize the kind of third-party SDKs that browsers are using.

To further discern whether permission-protected APIs are requested by first-party or third-party code, we use Androguard [2]. Specifically, we use Androguard to extract all of the permission-protected API calls in the code, extract the package of the class

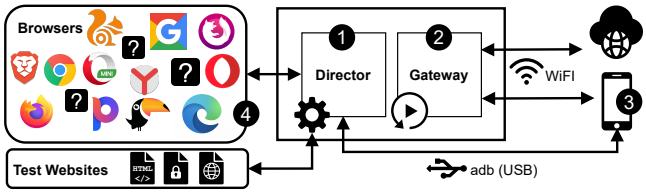

Figure 1: Overview of our dynamic analysis pipeline.

invoking it, and then use LibRadar to label whether this class belongs to the browsers' code or to a third-party SDK. We note that some of the SDK's code might never be used, and thus we filter out all third-parties for which we do not find a cross-reference in the browser's code to reduce over-reporting.

4.2 Dynamic Analysis

Dynamic analysis gathers evidence of browser behavior by executing browsers on an instrumented device and observing their behavior. Specifically, we monitor: sensitive data exposure, network traffic security practices, and web content manipulations as browsers fetch and load websites.

4.2.1 Pipeline. To perform dynamic analysis at scale, we develop an automated black-box testing instrumentation and logging testbed, which consists of several components as shown in Figure 1. Table 7 (in Appendix) summarizes the data collected by each component.

Director. The *Director* (Fig. 1 ❶) drives and orchestrates the analysis by installing the browsers from our dataset (Fig. 1 ❹) via `adb` on a physical mobile device (Fig. 1 ❸), instructing the device to open webpages using an `ACTION_VIEW` Intent [61], and uninstalling browsers via `adb` when tests are complete. The *Director* also collects device logs and screenshots during tests. The *Director* consists of JavaScript (≈500 LOC) and bash (≈400 LOC).

Traffic interception and injection with the gateway. The Gateway (Fig. 1 ❷) serves as a Wi-Fi hotspot for mobile devices. We route all mobile device traffic to an instance of `mitmproxy` [9] using `iptables` rules, and log all traffic with `tcpdump`. Using `mitmproxy`, the *Gateway* intercepts all mobile-device traffic. In combination with interception, we inject into each webpage a `tripwire.js` (inspired by Reis *et al.* [111]), which is a piece of JavaScript code (≈100 LOC) that collects the Document Object Model (DOM) of content delivered to the browser. We develop a Python `mitmproxy` add-on to perform these actions and use `tcpdump` to collect traffic that `mitmproxy` does not handle.

Test device. To enable the *Gateway* to modify traffic, we edit an Android 11 factory image (Build RQ3A.211001.001) and include our `mitmproxy` certificate as a root certificate on this image. In previous versions of Android (7 to 9), we could modify the root store (on the system partition) with root access; starting with Android 10 this is no longer possible because the system partition cannot be modified after installation. Thus, modifying the image before installation is the most convenient way to include a new root certificate. We run this modified factory image on a Pixel 3 device and use it as our test device (Fig. 1 ❸).

Assessing pipeline compatibility. Before conducting a large-scale analysis, we determine whether every browser in our dataset can run automatically (*i.e.,* without human intervention) in our testing pipeline. To do so, we induce each browser to open a simple webpage (using an Intent) hosted locally, and check whether the browser issues a request for it. We find that 381 browsers successfully pass this test without any interaction. In addition to these, we find that 19 browsers require minimal human interaction (dismissing onboarding screens) to be compatible with the pipeline. While testing these 19 browsers, we manually dismiss the onboarding screens so the automated tests can run. The remaining 24 browsers fail to follow the Intent to load pages and are thus incompatible with our test. These cases are indeed browsers, but fail to load a simple webpage automatically due to unexpected crashes or requiring users to login to use them. Nevertheless, we still include these browsers in our experiments, and report any privacy-related behavior we observe while they run (*e.g.,* some browsers send personal data to servers as soon as they are launched, even if they do not load a webpage). The experiment analysis logic consists of ≈4,000 lines of Python code.

4.2.2 Test Inputs. To observe any privacy-related browser behaviors, one must use browsers as they are intended: *i.e.,*visiting a series of real websites. Our testbed uses Intents to automatically induce browsers to fetch web content from the Internet, as if it were requested by a real user. We note that in our experiments, we do not click links inside the webpage, rather waiting for the page to fully load. For completeness, we rely on the following types of webpages (Appendix A provides more details): (1) a *honeypage* that contains several types of popular tracking services that browsers might block, (2) a *permissions page* that tries to access different device sensor APIs, (3) a *domain without a protocol specified* to study how browsers handles HTTP(S) by default, (4) a *HTTPS webpage* to analyze TLS security, and (5) 13 *popular webpages*. Of the 400 browsers that adhere to Intents and open webpages, we find that 367 load all of the test pages.

Handling dynamic web content. To build our cache of site content, we load all the selected webpages using a *baseline browser.* [2] Even with cached content, some webpages include non-deterministic and dynamic content (*e.g.,* a timestamp in the HTML source or ads resulting from real-time bidding processes [14, 131]) that leads to different requests over time and impedes comparing the same websites across browsers at different times. To address this challenge, we heuristically chose to rebuild the cache periodically (once every 50 browsers we test, which corresponds to about once every 2-3 hours). For each (non-baseline) browser test, our testbed prepares the mobile device by factory resetting it, installs necessary components, and collects identifiers (AdID and Android ID). Moreover, to ensure consistency in webpages delivered, the *Gateway* caches copies of tested websites and replays captured content using `mitmproxy`.

Detecting browser manipulations. Browsers can use their privileged position to actively modify web content in ways that protect (*e.g.,* ad-blocking) or harm users privacy (*e.g.,* JavaScript injection for tracking purposes). We design a novel methodology to detect such changes by considering a combination of network requests

[2]We note that any trusted browser can serve as a baseline, but we use the default Android WebView for our Android 11 test device.

seen across the browsers in our dataset, and instrumenting webpages to observe changes to each page's DOM. However, a key challenge is discerning whether such modifications are caused by the browser or by dynamic content as previously discussed. Therefore, our method for detecting content modification by browsers focuses on the most common known case, which is changing content in the DOM. Any other modifications that do not manifest in this way (*e.g.,* at rendering time) are out of scope. For that, we compare a *baseline* DOM of a webpage (assumed not to be modified) to the one rendered in the browser under test. Specifically, we generate a baseline DOM and instrument future webpage tests by injecting a JavaScript-based tripwire that detects and reports DOM changes compared to this baseline. To compare the browser DOM to our *baseline* DOM, we look at two particular HTML tags: `script` and `link` to find both added and missing elements. For each website in our test, we manually identify legitimate elements that vary across crawls and automatically remove them from the collected DOMs.

4.2.3 Network Traffic Analysis. We log all network traffic generated by browsers using `tcpdump`, and our testbed intercepts and injects/replays all HTTP/HTTPS traffic with our *Gateway* to serve consistent traffic and collect DOMs as described previously. However, we filter out flows that are not suitable for our webpage replay system. We drop all UDP traffic that is not DNS (port 53), including QUIC traffic. We found 8 browsers that use QUIC at least once and have no visibility into this traffic. Our approach also has no visibility into HTTPS traffic that uses any form of certificate pinning that `mitmproxy` cannot intercept. While pinning in browsers is a largely obsolete technique [106], we still find the case of 6 browsers that implement some for of pinning *and* that we cannot bypass. Given these limitations, the information exposure we do find serves as a lower bound.

5 BROWSER FUNCTIONALITY

We now analyze whether browsers protect or expose sensitive data during browsing sessions, according to our privacy protection and threat models in §2. Namely, we analyze whether browsers modify the content of a website that is served to users. This can potentially improve privacy by blocking requests to services known to track and profile users (*e.g.,* analytics services and ad networks) or to harm privacy (*e.g.,* when a browser adds tracking code to a website). We further analyze whether browsers block access to protected JavaScript APIs that can be used to gather sensitive data about the user and the device. Finally, we determine whether browsers help or harm privacy with respect to connection security/privacy.

5.1 Content Modification

DOM-based blocking. We use our DOM-based tripwires to investigate which content is blocked by browsers and the potential privacy impact of this behavior. We begin by identifying cases that are highly likely to correspond to tracking services, *i.e.,* the content matches an entry in at least one of two popular Adblock filters (`EasyList` [38] and `EasyPrivacy` [39]). We find that 276 browsers in our dataset (65%) block content flagged by these blocklists, strongly indicating that they are enhancing privacy for users.

Interestingly, most of the blocked content that we observe (73%) is not part of these lists. After manual analysis, we identify common trends that we can directly link to privacy-related behavior. Of the browsers blocking content not on these lists, we find that 60% block analytics content, 88% block ad content, and 77% block widgets (*e.g.,* an embedded Twitter feed). We look at libraries included in browsers to see if a common library results in this blocking, but found no evidence of such code. While we do not know exactly why these browsers block this content, we speculate that this could be an attempt to mitigate pervasive tracking from webpages. This also highlights the diversity in browsers' blocking behavior, indicating that different browsers use different anti-tracking and blocklist implementations [48, 132].

Allowed requests. In contrast to the above examples, browsers might impact privacy by changing and/or blocking *network requests* during page loads (*i.e.,* without changing the DOM). While it may seem trivial at first to detect blocked requests by comparing with a baseline browser, the key challenge for detecting this behavior is the lack of ground truth as to why a request is not issued. For example, a request could be absent due to a browser blocking a tracking service, but it also could be due to dynamic webpage behavior not captured in our baseline (*e.g.,* non-deterministic advertisements resulting from real-time bidding). Given this issue, we focus our analysis instead on the privacy impact of network requests to destinations matching the Adblock filters mentioned above. In this analysis, we focus simply on whether browsers seem to be using these filters to block all corresponding requests. If we see no requests at all to destinations on the Adblock filters, we assume the browser is using such filters to improve privacy. While we acknowledge that not all browsers implement blocklist approaches, we rely on these resources as ground truth to detect most browsers that are actively protecting the privacy of users.

We attribute requests to webpage loads in our tests by first restricting our analysis to only those with `HTTP referer` headers in a chain rooted at the loaded webpages. Next, we search for these requests in the Adblock filters. We find that 376 browsers (89%) permit at least one request that should be blocked based on these lists. One possible explanation for this behavior is that browsers use different blocklists, unblock lists, or neither—again, consistent with prior work observing variations in blocking behavior [48, 132]. Of the remaining 48 browsers, we look at browsers that load a majority of our test pages and make no requests on these lists and see that 17 browsers (4%) fit this criteria, including popular browsers like Firefox and Adblock Browser.

Content injection. Our DOM-based tripwire detects four browsers that inject extra scripts into loaded webpages. For two of them, we cannot locate them and thus we cannot asses their impact for privacy. Namely, Zdllq loads a script from the local file system that we could not locate even when de-compiling the app. The browser Yuyan injects a script downloaded from *pr.shuk.cn* which we only observed when loading the *office.com* and *patreon.com* websites, and for which we could not identify the purpose of the script or the server it contacts. Orbitum injects an open-source script called UseAllFive [45] that is related to UI functionality (and does not appear to be malicious). Both this browser and Dingzai also inject highly obfuscated scripts that prevent us from assessing their privacy impact. We argue that injecting remote scripts into

a webpage is generally problematic and a potential risk for users, particularly given the little transparency about their purpose.

5.2 Blocking Access to Protected APIs

We study how browsers enhance privacy by enforcing permissions when webpages invoke privacy-sensitive JavaScript APIs to access data. We do so by creating a test webpage that contains JavaScript code to access user data through WebAPIs. We test an extensive set of APIs including location, camera and microphone. We find that our test succeeds for 216 browsers (51%); all of these browsers reveal battery charging status, battery level and battery charging time without user permission. We also observe that 4 browsers support a permissions API that prompts users for access to various sensors (magnetometer, accelerometer, *etc.*). The default behavior for the remaining browsers is to deny access to any device sensors.

To understand why so many browsers block access to data from WebAPIs, we test the hypothesis that they could all simply be using default behavior built into a common implementation: Android *WebViews*. This is important as, by default, the *WebView* prevents exposure of the most sensitive data that has been shown to be useful for user tracking, allowing access only to the device accelerometer, magnetometer and battery [93]. Using the results from our browser engine attribution presented in §3, we see that of the 216 browsers that protect access to these APIs, 204 implement *WebView*s and send *X-Requested-With* headers. The 4 that support the permissions API are most likely to be Chrome-based, using this information (matches Chrome-based browsers). Lastly, we manually check the remaining 8 and see that they contain obfuscated code preventing our engine attribution.

5.3 Connection Security

We now describe how we conduct tests and capture TLS handshakes to understand whether browsers protect users' privacy and ensure connection security by correctly using TLS-based protocols. We focus on certificate validation and default connection security.

Default protocol preference. To determine whether browsers default to using the secure HTTPS protocol over HTTP, our testbed induces a browser to visit a domain without specifying the protocol to use (see §4.2 and Appendix A). For privacy and security, a browser should always favor HTTPS [104]. However, we find that only 10 (2%) of the browsers pick HTTPS by default. Of these browsers, just 1 is available on the Google Play Store (FOSS Browser), the remaining 9 browsers are from Chinese stores. None of the most popular browsers on Google Play, Anzhi, or AppChina (see Table 1) implement HTTPS by default.

Certificate validation. Browsers that do not properly validate TLS certificates compromise user security as they enable adversaries to mount person-in-the-middle attacks as they would accept arbitrary certificates. To detect whether such attacks are possible, we use the methodology presented in §4.2, but skip installing the `mitmproxy` root certificate on the mobile device. As a result, our self-signed certificates used during TLS interception is not trusted by the system and the corresponding connections should be dropped. We then visit an arbitrary HTTPS website with each browser on our test device; a correct implementation of TLS validation should reject the certificate. Nevertheless, we find that 44 (10%) browsers accept the invalid certificate and are thus vulnerable to arbitrary TLS interception attacks. These browsers originate both from Chinese stores (26) and Google Play (18, downloaded 11M+ times combined). Of the 44 browsers, 3 (1 from Google Play) display a warning stating that there is a possible security problem. Nevertheless, while the warning is in place, the browsers finish loading the webpage in the background. Ironically, of the 18 browsers that are available on Google Play, 2 (InBrowser Incognito and InBrowser Beta both developed by "Private Internet Access, Inc") advertised in their description that they provided users with "secure" and "incognito" features, yet they fail to validate certificates. We reached out to the 14 developers for which we could find contact information and also opened a bug with Google to report this issue.

Secure protocols. Finally, we look at how browsers affect users by relying on secure protocols (namely, DoH, DoT and OCSP). Our experiments revealed no use of DNS-over-HTTPS (DoH) or DNS-over-TLS (DoT) by default, which would provide browsers with private/secure DNS queries and responses. Thus, all browsers' DNS requests are sent in plaintext (allowing ISPs and other network observers to see them). We also did not see any Online Certificate Status Protocol (OCSP) traffic, a protocol that can be used to verify the revocation status of a X.509 certificate. Our results are consistent with prior work indicating that mobile browsers fare poorly when it comes to correct certificate validation [85].

6 PRIVACY ANALYSIS

As with any ordinary Android app, mobile browsers have access to PII, such as Android Advertisement IDs (AdID), persistent identifiers, or device location, most of which are protected by Android permissions. Further, some browsers might request permissions that are used for secondary purposes, *e.g.,* for embedded SDKs or to support JavaScript APIs that can in turn be used by code on arbitrary websites. In addition, they have unique access to browsing history and browsing patterns. In this section, we examine the permissions requested by mobile browsers regardless of the purpose. Then we identify PII in network traffic observed during dynamic tests and analyze their privacy implications for users. Finally, we analyze cases where browsers potentially harm user privacy by sharing browsing history.

6.1 Permission Analysis

To understand the type of access to user data that browsers have, we analyze the permissions they request following the methodology discussed in § 4.1. We extract a total of 1,181 unique permissions requested by the apps in our dataset. This includes both permissions defined in the Android Open Source Project (AOSP) [56] and custom permissions [65]. Table 2 shows the number of requested permissions per browser origin. We find that browsers that are pre-loaded or distributed through Chinese stores tend to request more permissions than those listed on Google Play: the median number of permission requests (AOSP and custom included) is 20 for pre-loaded browsers and 21 for browsers from Chinese stores, compared to 14 for browsers from Google Play. However, the maximum number of permissions requested by pre-installed browsers is *lower* than browsers from other sources: only 40 permissions, compared to 106 for browsers available on Google Play, and up to

Table 2: Number of requested permissions per dataset.

Origin	AOSP perms.		Custom perms.		All perms.	
	Median	Max	Median	Max	Median	Max
Chinese Stores	20	101	2	130	21	231
Google Play Store	11.5	39	2	79	14	106
Pre-installed	17	32	3	8	20	40
All origins	**13**	**101**	**2**	**130**	**15**	**231**

231 for browsers in Chinese stores. In total, we extract 522 custom permissions requested by the apps in our dataset. However, such permissions usually lack documentation [51], and their purpose is difficult to automatically and reliably infer from the application itself. We thus discard such permissions from the rest of our analysis.

Figure 2 shows the number of apps requesting AOSP permissions. We show only the permissions requested by at least 20% of the apps in our global dataset to maintain readability. The height of each bar indicates the percentage of apps, among the apps from the same origin, that request a given permission. Permissions with a black label (**normal permissions**) are classified as "low risk" for users in Android's AOSP documentation, they are granted at installation time (*e.g.,* `INTERNET`) [56]. Similarly, permissions with an orange label (**dangerous permissions**) are those that must be explicitly granted at runtime as they can put the user's privacy at risk (*e.g.,* `ACCESS_FINE_LOCATION` gives access to the fine-grained geolocation of the device). Finally, permissions with a red label (**signature permissions**) are automatically granted (with no user interactions) to an app if it is signed with the same certificate as the app that declared the permission. Only system apps (*i.e.,* apps located on one of the system partitions of the device) can be granted such permissions [17], as they allow an app to access very sensitive information (*e.g.,* the `READ_LOGS` permission allows an app to read the system logs) or to perform low-level operations such as mounting and unmounting filesystems. Some publicly available browsers request such permissions nevertheless. One possible explanation is that these browsers can also come pre-installed on some devices (*e.g.,* Google Chrome), and that developers do not remove these permission requests before submitting their apps to app markets.

Unsurprisingly, we find that the most frequently requested permission is `INTERNET`, which is required for Internet access, followed by `WRITE_EXTERNAL_STORAGE`, for writing to the SD card, and `ACCESS_NETWORK_STATE`, for checking the device's connectivity. These permissions are necessary for implementing basic features of a web browser. Figure 2 also includes now-deprecated permissions that we keep in our analysis, as they still give access to protected resources to apps on devices running an old-enough Android version. This includes `WRITE_` and `READ_ HISTORY_BOOKMARKS` which, until Android 6.0, would allow apps to "read (resp. modify) the history of all URLs that the Browser has visited, and all of the Browser's bookmarks" [62, 63].

Our results show that a large number of browsers also request access to sensitive features, such as the location or recording audio. These permissions may have a legitimate use case, such as enabling access to JavaScript APIs (*e.g.,* a weather website requesting the user's location, or a videoconferencing site needing access to the microphone and camera). In any case, the fact that the browser has access to this information can pose a privacy risk if that data is used and shared in unintended or unexpected ways (*e.g.,* shared with third-party SDKs for secondary purposes, as we study next).

Access to permissions for secondary purposes. Using the pipeline described in §4.1, we find that the inclusion of third-party libraries is common in mobile browsers: we find at least one advertising (*e.g.,* Google Ads, Baidu Mobile Ads), analytics (*e.g.,* Crashlytics, AppsFlyer) or social network (*e.g.,* Facebook, Umeng) library in 63% of browsers. The most common SDK across all browsers is Google Ads (30%), which means that developers can potentially add their own advertisements on top of those that are present in websites (however we did not find any instance of this behavior at run time). We also find that apps from Chinese stores rely on libraries that target Asian markets (*i.e.,* Baidu, Umeng or Tencent).

As discussed in §4.1, Android's permission model allows third-party libraries included in Android apps to inherit the permissions requested by the host app (*i.e.,*the browser). We now analyze whether permission-protected methods are requested by third-party libraries embedded in each browser in order to infer potential secondary purposes. We focus on libraries offering advertising, analytics and social networking services, as they are more likely to leverage the set of dangerous permissions granted to the host app to collect unique identifiers and behavioral data.

Table 3 shows, for each permission labeled by Android as dangerous, the percentage of browsers in which at least one permission-protected API that requires such permission is present only in browser code, third-party code or in both. As we explained in §4, we rely on a mapping from API calls to AOSP permissions to generate this data. In addition, when the API call is present only in a third-party SDK or both in a third-party SDK and first-party code, we specify in parentheses whether any of the third-party SDK has Advertisement and Tracking (A&T) capabilities. Note that due to our manual classification effort, we might have missed some packages that have A&T capabilities. This distinction is important as non-A&T SDKs might access permissions for features inherent to the browser while SDKs with tracking capabilities might use personal data from users for secondary purposes.

We find significant fractions of third-party SDKs with A&T capabilities using dangerous permissions. The results show that most permissions are requested by A&T companies for accessing unique identifiers (`READ_PHONE_STATE` by SDKs such as Facebook Ads or Google Mobile Services), and location info (`ACCESS_FINE_LOCATION` by SDKs such as Umeng, Amplitude or AppsFlyer). Note that static analysis allows us to detect only potentially privacy-harming behaviors, as it is prone to miss behaviors due to code obfuscation, reflection or dynamic code loading. In the next section, we investigate how the presence of these third-party SDKs and their access to the permissions of the host browser translates to data dissemination to third-party servers at run time.

6.2 Observed PII Exposure

We search the network traffic collected during dynamic testing to identify whether it contains PII, and if so, where the PII is sent. Similar to prior work [113], we look for sensitive data (as listed in §2) both in clear text and using popular hashing algorithms (MD5, SHA-1, SHA-224, SHA-256). For network traffic, we consider HTTP

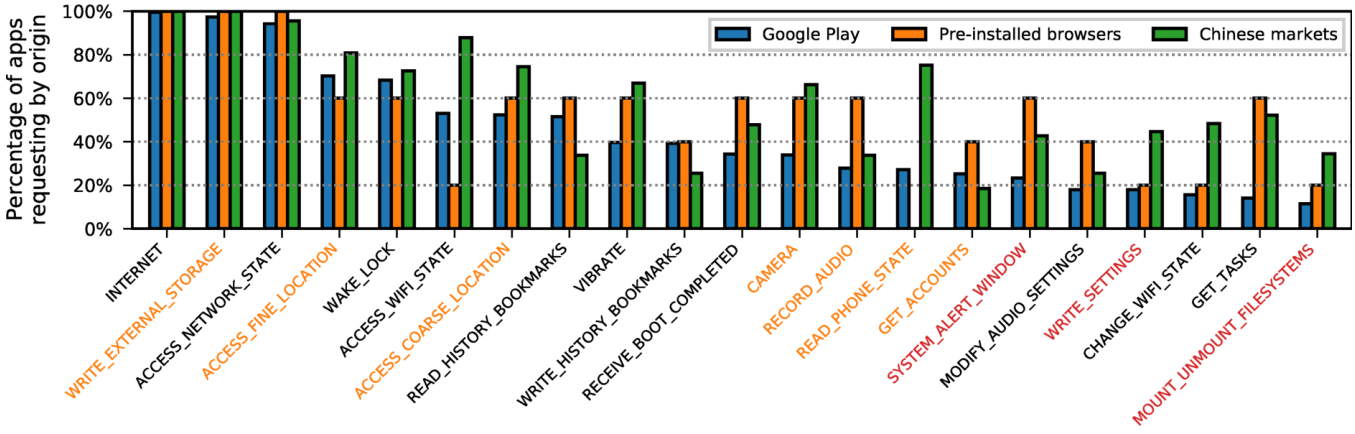

Figure 2: Frequently requested permissions: dangerous are in orange and in red those not available to third-party developers.

Table 3: Percentage of browsers where a given permission is accessed only in browser code, in third-party code or in both.

Permission	Browser only	SDK only (A&T)	Browser and SDK	# of browsers
ACCESS_COARSE_LOCATION	2%	83% (39%)	14% (125%)	113
ACCESS_FINE_LOCATION	2%	83% (37%)	13% (133%)	112
READ_PHONE_STATE	25%	41% (73%)	32% (144%)	55
GET_ACCOUNTS	56%	28% (14%)	16% (25%)	25
SEND_SMS	0%	100% (0%)	0% (0%)	1

message bodies, decoded with both the content-encoding header (*e.g.,* `gzip`) and content-type header charset. Of course, any PII not using these encodings will be missed by our analysis. In the case of files, we look at the content that is written to or read from them. Last, we use data from our *baseline browser* to clean up these results. Specifically, we consider the data shared by the baseline browser to be due to the webpage itself and not the browser that visits them, and so we ignore it. Thus, we are left with data that is generated by the function of the browser and use this for our analysis.

PII exposure in network traffic. We first analyze network traffic (both encrypted and plaintext) to identify whether browsers (or embedded third-party SDKs) exfiltrate sensitive data, and determine which destinations receive this data. We emphasize that a number of these identifiers (*e.g.,* AdID, Android ID, MAC Addr.) cannot be accessed using Web APIs; thus, for an endpoint to be receiving them, the browser must be accessing them directly from Android.

We find that PII exposure is extensive: 32% of browsers disseminate at least one type of PII. The most common type of PII shared over the network is the AdID, "a unique, user-resettable ID for advertising, provided by Google Play services" [53]. Nevertheless, our results also show apps still collect non-resettable identifiers contrary to Google's best practices [55]. In fact, starting with Android 10 (released in September 2019), Google Play added more restrictive permission requirements to access non-resettable identifiers for apps published in the Google Play Store [57]. Yet we find that collecting and sharing non-resettable identifiers (*e.g.,* the device MAC address and Android ID) still persists.

We observe PII dissemination regardless of the origin of the browser (Table 4). The types of PII exposed span all the categories

Table 4: Number of browsers across datasets sharing PII (by type) via the network.

PII Type	Google Play	Pre-installed	Alt. Markets	All Origins
AD ID	83	2	13	98
Location Info.	16	0	25	41
MAC_D	2	0	6	8
Android ID	1	0	6	7
Installed Packages	2	0	1	3
MITM Cert.	0	0	1	1
Total	**92**	**2**	**43**	**135**

of PII tested, even for apps from the Google Play Store (where there are stricter rules for compliance with data-collection policies [66]). Perhaps in part due to such policies, a larger percentage (6%) of browsers originating from Chinese stores collect non-resettable IDs when compared to those from Google Play (2%). Similarly, a larger percentage of Google Play browsers (31%) collect resettable IDs, as compared to those from Chinese stores (8%).

Next, we investigate the destinations that receive this PII, a complete list of these is in Appendix B. As expected, most of these destinations align with our static-analysis findings(§6.1). Google SDKs (which can be used for tracking and advertisement purposes) such as Google Analytics and Firebase are commonly included in mobile browsers and thus they frequently transmit PII. Similarly, we observe that well-known social media companies, such as Facebook and Twitter, also receive PII from browsers. Our SDK analysis showed that apps from Chinese stores rely more often on providers specialized in the Chinese market, a finding that is confirmed by our runtime analysis (*e.g.,* Tencent and Baidu).

The presence of SDKs in apps does not always result in PII being shared with these parties (*e.g.,* Umeng). On the contrary, dynamic analysis allows us to find third parties that receive sensitive data (*e.g.,* Alibaba) but went unnoticed during our static analysis. This shows the importance of complementing static analysis with a run-time analysis of browser behavior. In terms of the type of PII most commonly shared, we find that those related to targeted advertisement and tracking of users are the most prevalent (namely the AdID and the device geolocation). This is not surprising, as third-party SDKs often track users, *e.g.,* to build comprehensive profiles of their behavior and preferences [25]. We also find 3 apps

Table 5: Number of browsers sharing visited domains via search and suggestion queries (* First party).

Destination Service	# of Browsers
Google Suggest	34
Google Search	16
Yahoo Suggest	2
Yandex Favicon*	2
Ninesky Favicon*	1
Opera Sitecheck*	1
Yahoo Search	1
Yandex Favicon	1
Baidu Suggest	1
Opera Sitecheck	1
Total	**60**

bridging IDs, that is, sending both resettable and non-resettable identifiers to the same host (a domain that belongs to UC). This goes against Google's policy [55, 68] as it allows companies to track users longitudinally even if they do reset their AdID. Two of these apps come from Chinese stores and one (UC Browser Turbo) is available on the Google Play Store.

6.3 Browsing History Exposure

To understand the exposure of browsing history, we search for network requests that contain the names of the automatically visited websites. As with our analysis of other types of PII in §6.2, we account for a wide range of content and character encodings (*e.g.,* gzip, JSON, hashed values). We find that 81 browsers expose the identity of a majority of visited websites to an unrelated destination (*i.e.,* one not contacted when loading the tested websites in a baseline browser). We manually confirm that these domains are not included in any of the websites' source code. To understand why browsing history is being shared, we manually analyze the payload of the requests and group behavior into two categories: those where the browsing history is exposed in support of primary browser functionality and those where we could not find any legitimate justification for such behavior.

First, we find 60 browsers where we identified a feature that requires this data, *i.e.,* search and suggestion APIs, site checks, compatibility checks, URL safety checks and favicon services. None of these requests included any other type of PII (such as unique identifiers). We manually inspected the code of one of the browsers in which we identify such behaviors and confirm that indeed, whenever the user inputs a domain, this generates a requests to a search suggestion API (in this particular case either Google or DuckDuckGo). This is in line with behavior reported by previous work [80]. Table 5 shows the different services that we identified in our analysis. Querying a search API might be expected in some use cases, *e.g.,* when the user inputs a term that is not a valid URL. Nevertheless, when a user enters a complete URL, they may not expect or want the URL to be exposed to another party.

Table 6 shows the destinations that receive browsing history along with PII for the cases in which could not identify a feature. Destinations found in only one browser are shown in Appendix C. Of the 37 browsers where we cannot identify a feature that requires sharing browsing history, we find that 13 browsers send this data

Table 6: Number of unique browsers sharing browsing history and PII with other parties (* First party), allowing to link the history to a unique user.

Destination	# of Brow.	Loc.	AdID	IMEI	MAC_D
AppsFlyer	8	1	8		
Casale Media	7				
Rubicon Project	6				
Yandex*	3		1		
Moat	3				
Verizon	2		1		
360	2	1			
Adnxs	2				
Total	**37**	**4**	**10**		

to endpoints along with unique user identifiers (the AdID in 9 browsers, location data in 3 browsers and both of these in one browser). Browsers that send both visited websites and other PII to destinations can be harmful to user privacy because they have the ability to link the browsing history to an individual. We find browsing history exposed to third-party organizations known for offering tracking and advertising solutions, such as AppsFlyer [5], Firebase [7] or Verizon (owner of SDKs such as Flurry [8]).

To further understand the privacy risks posed by these third-party SDKs, we manually analyze some of their code. For instance, the MiOTA browser implements its own version of Android's *WebView*. After every request has finished, the browser runs JavaScript code that sends the visited URL along with identifiers (*e.g.,* the Android ID) to a first-party domain. Another example is Stealth Browser which sends the visited URL to a destination that belongs to Fillr [6]. The browser's description on Google Play [69] confirms that this browser embeds the Fillr third-party SDK, explaining why we see these requests in our test. Ironically, both browsers claim to offer privacy as one of their features in their descriptions. Some of the observed browser-history sharing might serve to improve the browsing experience. For instance, we find destinations related to the browsers' own companies (for browsers like *UCWeb*, *Opera*, or *Kiddoware*), and argue that this data collection could be used for telemetry, safe browsing or parental control. Users might decide that the benefits of such collection may offset the privacy risks.

7 MULTIDIMENSIONAL ANALYSIS

In this section, we combine the findings of previous sections that showed instances of privacy enhancing and harming behavior by browsers to gain a global picture of the privacy disposition of browsers through a multidimensional lens. This allows us to identify instances where browsers are relatively helpful or harmful when it comes to exposing users to data collection, and identify other interesting cases of unexpected behavior.

Methodology. To quantify the privacy disposition of each browser, we translate each observed behavior into a quantified, normalized score. Specifically, we focus on four main categories of privacy-impacting behavior for which we assign each browser a score between 0 and 1. (*i*) *Blocking tracking content or allowing requests to tracking services*, for which we use simple binary values. (*ii*) *Connection security* with binary values as well. (*iii*) *PII exposure*, for which we assign three levels of values, highest for *Non-resettable*

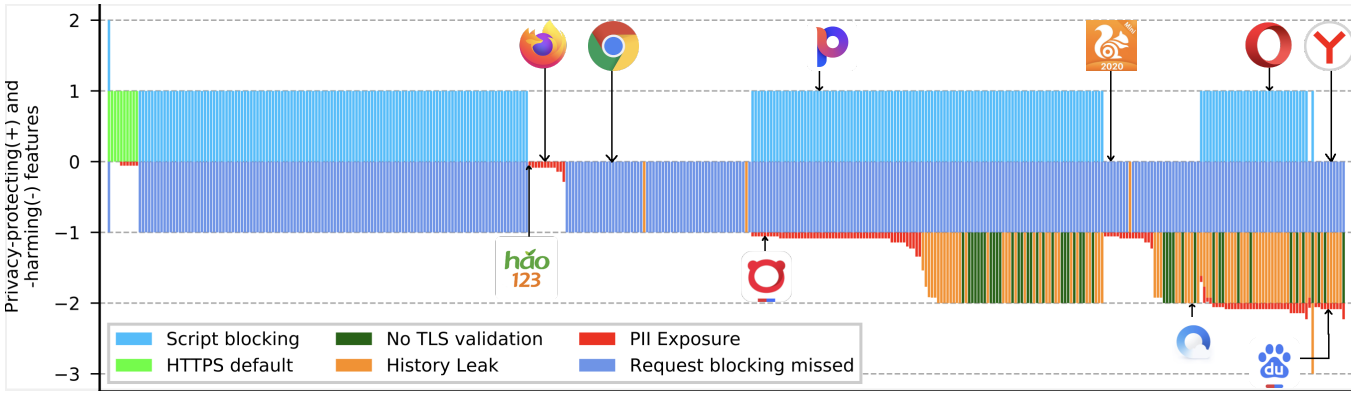

Figure 3: Quantification of browser's privacy-enhancing and -harming behaviors. Browsers icons can be found in Table 1.

IDs, medium for *Resettable IDs*, and lowest for other identifiers. We sum these factors across browsers and normalize them to a maximum value of 1. (*iv*) *Sharing browsing history*, for we use binary values: if a browser shares the vast majority of websites visited with another party, the value is 1; otherwise it is zero.

Results. Figure 3 represents for each browser (x-axis), the observed behaviors that improve user privacy in the positive y-axis and the ones that may harm them on the negative y-axis. Thus, we combine information from all our tests to help understand how each browser protects or harms privacy. The graph is ordered by decreasing privacy protection from left to right, meaning the browsers that mostly protect user privacy are on the left and those with significant potential harms to user privacy are on the right.

Figure 3 shows wide variations across browsers in terms of behaviors that impact user privacy, and emphasizes the importance of a multidimensional analysis when considering the privacy disposition of a browser. Of the 5 worst browsers (towards the right end in Figure 3), 4 are available on Google Play (Savannah, Star, Yandex, and Yandex Beta). The Yandex browser is extremely popular (> 100 million downloads, but harms user privacy by collecting Advertisement IDs (red in Figure 3) along with browsing history (orange in Figure 3). Meanwhile, a browser found on a Chinese store TxWebLibrary collects user history while disregarding TLS certificates (dark green in Figure 3). On the left end, browser that protect user privacy tend to implement HTTPS by default and block content like trackers and ads (FOSS Browser). Among the most popular browsers, Chrome and Firefox are ranked best for privacy, Phoenix is middle-of-the-pack, and Opera, UC-mini, and Yandex fare poorly. We also find interesting cases with mixed impact on privacy between these extremes. For example, out of the 276 browsers that block tracking content, we see that 195 also allow tracking requests, 63 expose PII, 38 share the browsing history, and 20 fail to validate certificates. Finally, KKBrowser (with two different signatures in our dataset) uses HTTPS by default and blocks tracking content, but also fails to correctly validate TLS certificates and exposes PII.

8 RELATED WORK

Previous work leveraged static [22] and dynamic [41, 79, 109] analysis to identify security and privacy threats in mobile apps, reporting on: malicious and insecure behaviors [30, 77, 83, 100, 110, 114, 127], presence of third-party code [90, 103, 108, 112, 117] and lack of transparency when it comes to privacy [99, 134]. Researchers have shown that privacy issues can occur in all types of Android apps, regardless of their origin [51, 126], or target audience [113, 123]. Previous work has also shown the pitfalls of static techniques [35, 46, 91, 105] and dynamic techniques [32, 51, 70, 118]; finally showing that combining both analysis techniques can be useful for better understanding security and privacy issues, such as the use of covert and side channels [110] or the privacy risks of analytics libraries [84]. Our work combines static and dynamic analysis to provide a comprehensive view of privacy issues in mobile browsers.

In the mobile browser ecosystem, Niu *et al.* highlighted the security problems of browsers on mobile devices in comparison to those on desktops [98]. Leith studied privacy issues of five major mobile browsers [80], showing that indeed some browsers are more privacy-protecting than others. Luo *et al.* studied the evolution of UI vulnerabilities in mobile browsers [87], as well as provided a longitudinal analysis of security mechanisms supported by mobile browsers [86]. Vila *et al.* demonstrated the possibility to perform side-channel attacks on Chrome's event loops to identify websites and user behavior [125]. Lin *et al.* explored the privacy and security threats of Chromium-based browsers' autofill functionality, allowing attackers to extract user data and studying the potential for such an attack in the wild [82]. Wu *et al.* [128] highlighted vulnerabilities in mobile browsers that could give attackers access to users' cookies and browsing history through local file access. Kondracki *et al.* identified potential security issues introduced by data saving browsers [76]. Furthermore, several related papers identified possible attacks on Android's WebView [26, 31, 88, 89, 94, 95], which is used by some browsers and many other types of mobile apps to display web content. Most recently, Krause [78] used an instrumented website to reveal that that in-app browsers, such as the one in TikTok, collect extensive data about user interactions with websites via JavaScript. However, in-app browsing interfaces are out of scope for our work, as they do not allow users to visit arbitrary URLs as freely as with a dedicated browser. In contrast, we aim to analyze the whole ecosystem of Android browsers, from those that are more well-known and available in the Google Play Store to those less-known and from Chinese stores.

In terms of server-side privacy attacks, Eckersley looked at the possibility to fingerprint browsers based on their uniqueness [40].

Vastel *et al.* further showed that browser fingerprints are a viable option to perform long-term tracking of users, even when those fingerprints change over time [124]. Das *et al.* showed how mobile specific sensors are used for tracking [36] and Marcantoni *et al.* have extended this work, studying the prevalence of WebAPI sensor accesses on the web [92]. To the best of our knowledge, our work is the first to investigate *privacy implications* of Android browsers from different sources, relying on a novel methodology to study whether browsers promote user privacy by blocking data collection by other parties, or whether they harm it by collecting and/or sharing user data. We develop novel analysis techniques and apply them to a large collection of mobile browsers.

9 DISCUSSION

Limitations. We limit our analysis to the Android ecosystem of mobile browsers, due to the open nature of Android, which permits browsers from very different origins and business models. In contrast, Apple's iOS restricts all browsers to use WebKit [19], and users were not allowed to change their default browser until the recent iOS 14 [20]. At the methodology level, it is important to note that our static analysis results (§ 4.1) could be impacted by apps making use of code obfuscation and dynamic code loading. This is a well known limitation of static analysis, and it can impact our ability to find third-party SDKs and identify calls to permission-protected methods. Our dynamic analysis pipeline (§ 4.2) also presents limitations, preventing us from testing 24 browsers dynamically. Moreover, our behavioral analysis represents a lower bound on potentially problematic browser behaviors, as we do not exhaustively cover all code paths due to limitations of current fuzzing methods. We also refrain from looking at the purposes of data collection, due to the difficulty and scalablity issues caused by trying to automatically extract this information from privacy policies at scale. The number of pre-installed browsers that we successfully tested is small due to their native dependencies which cannot be reproduced on our test device [51]. Finally, we report on all run-time data collection from mobile browsers by approving all permission requests, which may differ from how any individual user gives consent.

Mitigation. Given the lack of transparency into how browsers impact privacy and security, we argue that app stores could play a vital role in mitigating harms for users. Specifically, app stores can use our technique (building on our open-source code) to measure the privacy and security of browsers and decide on whether to publish a browser based on the results. We believe this would incur low additional cost since app stores already review apps and run automated tools to analyze them, and doing so could protect large numbers of users. In addition, stores (or other sites) can make a transparency report available to users at install time, similar to the data Apple and the Google Play store report to users regarding an app's access and use of personal or sensitive data.

Lessons learned. At first glance, a privacy analysis of mobile browsers seems straightforward. However, we learned a number of lessons in addressing key challenges that demonstrated it was far from trivial. One such challenge was dealing with web page dynamics, which make it difficult to reliably infer browser behavior. Our web page replay approach, while not entirely novel, required substantial effort to properly handle page dynamics—including those that occur even when the page source stays the same. Another challenge was determining how a browser modifies page content without instrumenting or analyzing the code of hundreds of browsers. We found that injecting our own Javascript into webpages provided a surprisingly straightforward way to capture this information across all browsers without needing manual instrumentation or analysis. Last, we stress that simply looking at PII exposure is not enough when considering the purpose of apps like browsers. In our case, we found browsers to share potentially highly sensitive metadata (browsing history) with other parties. For future work, it is important to consider such non-standard types of sensitive data that can be exposed by special-purpose apps that handle sensitive data beyond those protected by OS permissions.

Future work. Investigating regulatory compliance of consent, opt-out, and the accuracy of privacy policy text is an interesting area for future work, but it complicated by the complexity and scalability issues of automatically inspecting privacy policies. Interestingly, only 13 of the browsers had some kind of initial on-boarding screen that the user has to interact with before using the browser, while 19 implicitly or explicitly present users with a privacy policy. Therefore, for the majority of browsers, we arguably never give explicit consent to data collection (other than by simply installing and running the browser). We note that we also looked for file modification by browsers in an attempt to find apps writing sensitive information to the external storage as previous work has shown the risks of exposing of PII to external storage, and how this can be used as a covert channel [27, 110]. However, we did not find any instance of this behavior during our tests.

10 CONCLUSION AND KEY FINDINGS

This paper conducted the first large-scale privacy analysis of Android browsers. To that end, we analyzed a set of 424 browsers collected from different sources, including the Google Play Store, popular Chinese stores, and pre-installed apps. We developed a custom, novel methodology that combines static and dynamic analysis to identify and quantify privacy-protecting and -harming behaviors. We show that some browsers have the ability to protect user privacy, as 65% of these browsers block tracking scripts, and 65% block access to protected JavaScript APIs (§ 5.1 and § 5.2). However, we also find security issues in 10% browsers that fail to validate TLS certificates, and discover that only 2% of browsers default to HTTP**S** (§ 5.3). We also find four browsers that modify webpages and inject scripts into loaded webpages. Our analysis of mobile browser code shows that these apps request a wide range of permissions and that the personal data protected by them can be accessed by third-party libraries that may use the data for secondary purposes (§ 6.1). To identify a lower bound of how much personal data is exposed by these browsers at runtime, we analyzed network traffic while browsers visit a controlled set of websites. We find evidence of 32% of browsers disseminating at least one type of PII, including resettable and non-resettable identifiers (§ 6.2). Given the increased flexibility for users to select their own default browser and wide range of privacy-impacting behaviors observed, we argue there needs to be greater transparency and auditing of mobile browsers, which our techniques can readily inform.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback on improving our paper. We also thank Jakob Bleier for his assistance with the browser engine attribution.

This research was partially funded by the NSF under SaTC-1955227. This research also received funding from the Vienna Science and Technology Fund (WWTF) through project ICT19-056, as well as SBA Research (SBA-K1), a COMET Centre within the framework of COMET - Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

IMDEA Networks' researchers are funded by EU's H2020 Program (TRUST aWARE Project, Grant Agreement No. 101021377) and the Spanish Ministry of Science (ODIO Project, PID2019-111429RB-C22). Dr. Narseo Vallina-Rodriguez is funded by a Ramon y Cajal Fellowship from the Spanish Ministry of Science and Innovation.

REFERENCES

[1] 2022. 360. http://zhushou.360.cn/.
[2] 2022. Androguard. https://github.com/androguard/androguard.
[3] 2022. Anzhi. http://www.anzhi.com/.
[4] 2022. AppChina. http://m.appchina.com/.
[5] 2022. AppsFlyer. https://www.appsflyer.com/.
[6] 2022. Fillr. https://www.fillr.com/.
[7] 2022. Firebase. https://firebase.google.com/.
[8] 2022. Flurry. https://www.flurry.com/.
[9] 2022. mitmproxy. https://mitmproxy.org/.
[10] 2022. Tencent. https://android.myapp.com/.
[11] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.
[12] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: Dusting the Web for Fingerprinters. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.
[13] Gaurav Aggarwal, Elie Bursztein, Collin Jackson, and Dan Boneh. 2010. An Analysis of Private Browsing Modes in Modern Browsers. In *Proc. of the USENIX Security Symposium*.
[14] Syed Suleman Ahmad, Muhammad Daniyal Dar, Muhammad Fareed Zaffar, Narseo Vallina-Rodriguez, and Rishab Nithyanand. 2020. Apophanies or Epiphanies? How Crawlers Impact Our Understanding of the Web. In *Proc. of the Web Conference (WWW)*.
[15] Alexa. 2020. The Top 500 Sites on the Web (By Category). https://www.alexa.com/topsites/category/. (accessed 2020-08-28).
[16] Android Open Source Project. 2021. Android 11 Compatibility Definition Document: Software. https://source.android.com/compatibility/11/android-11-cdd#2_2_3_software. (accessed 2021-06-08).
[17] Android Open Source Project. 2021. Privileged Permission Allowlisting. https://source.android.com/devices/tech/config/perms-whitelist. (accessed 2021-06-08).
[18] AppInChina. 2022. App Store Index. https://www.appinchina.co/market/app-stores/. (accessed 2022-03-15).
[19] Apple. 2020. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/.
[20] Apple. 2020. New features coming with iOS 14. https://www.apple.com/ios/ios-14-preview/features/.
[21] Sajjad Arshad, Amin Kharraz, and William Robertson. 2016. Identifying Extension-based Ad Injection via Fine-grained Web Content Provenance. In *Proc. of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.
[22] 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. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[23] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.
[24] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proc. of the USENIX Security Symposium*.
[25] Muhammad Ahmad Bashir and Christo Wilson. 2018. Diffusion of User Tracking Data in the Online Advertising Ecosystem. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)* (2018).
[26] 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 (SecWeb)*.
[27] Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2017. Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.
[28] Sarah Bird, Ilana Segall, and Martin Lopatka. 2020. Replication: Why We Still Can't Browse in Peace: On the Uniqueness and Reidentifiability of Web Browsing Histories. In *Proc. of the Symposium on Usable Privacy and Security (SOUPS)*.
[29] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. 2021. Trouble Over-The-Air: An Analysis of FOTA Apps in the Android Ecosystem. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.
[30] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proc. of the USENIX Security Symposium*.
[31] Erika Chin and David Wagner. 2013. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Proc. of the International Workshop on Information Security Applications (WISA)*.
[32] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[33] Chromium. 2022. permissions.site. https://github.com/chromium/permission.site.
[34] Chromium. 2022. Sending X-Requested-With header to every website is a device fingerprinting risk. https://bugs.chromium.org/p/chromium/issues/detail?id=960720.
[35] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[36] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.
[37] Thomas Dullien and Rolf Rolles. 2005. Graph-based Comparison of Executable Objects. *Symposium sur la sécurité des technologies de l'information et des communications* 5, 1 (2005), 3.
[38] EasyList. 2021. Version 202105250854. https://easylist.to/easylist/easylist.txt. (accessed 2021-06-07).
[39] EasyPrivacy. 2021. Version 202105250854. https://easylist.to/easylist/easyprivacy.txt. (accessed 2021-06-07).
[40] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*.
[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.
[42] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.
[43] European Commission. 2018. Antitrust: Commission fines Google €4.34 billion for illegal practices regarding Android mobile devices to strengthen dominance of Google's search engine. https://ec.europa.eu/commission/presscorner/detail/en/IP_18_4581.
[44] Exodus. 2020. Trackers. https://reports.exodus-privacy.eu.org/en/trackers/. (accessed 2020-09-02).
[45] Jason Farrell. 2021. UseAllFive. https://github.com/UseAllFive/true-visibility.
[46] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2014. Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation. In *Proc. of the International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.
[47] Álvaro Feal, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez. 2021. Don't Accept Candy from Strangers: An Analysis of Third-Party Mobile SDKs. *Data Protection and Privacy: Data Protection and Artificial Intelligence* (2021).
[48] Álvaro Feal, Pelayo Vallina, Julien Gamba, Sergio Pastrana, Antonio Nappa, Oliver Hohlfeld, Narseo Vallina-Rodriguez, and Juan Tapiador. 2021. Blocklist Babel: On the Transparency and Dynamics of Open Source Blocklisting. *IEEE Transactions on Network and Service Management* (2021).

[49] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.

[50] Gabi Cirlig. 2021. The 4th largest mobile browser exfiltrates users' data even in Incognito mode. https://hookgab.medium.com/ucbrowser-privacy-study-ecff96fbcee4. (accessed 2021-06-07).

[51] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An Analysis of Pre-installed Android Software. *Proc. of the IEEE Symposium on Security and Privacy (S&P)* (2020).

[52] Google. 2019. Presenting search app and browser options to Android users in Europe. https://www.blog.google/around-the-globe/google-europe/presenting-search-app-and-browser-options-android-users-europe/.

[53] Google. 2020. Advertising ID. https://support.google.com/googleplay/android-developer/answer/6048248. (accessed 2020-09-03).

[54] Google. 2020. Android Studio Code Annotations. https://android.googlesource.com/platform/tools/adt/idea/+/refs/heads/mirror-goog-studio-master-dev/android/annotations/android/. (accessed 2020-03-23).

[55] Google. 2020. Best practices for unique identifiers. https://developer.android.com/training/articles/user-data-ids. (accessed 2020-07-29).

[56] Google. 2020. Permissions overview. https://developer.android.com/guide/topics/permissions/overview. (accessed 2020-03-02).

[57] Google. 2020. Privacy changes in Android 10: Restriction on non-resettable device identifiers. https://developer.android.com/about/versions/10/privacy/changes#non-resettable-device-ids. (accessed 2020-07-29).

[58] Google. 2020. RequiresPermission: Android Support Library. https://developer.android.com/reference/android/annotation/RequiresPermission. (accessed 2020-03-23).

[59] Google. 2020. RequiresPermission: AndroidX. https://developer.android.com/reference/androidx/annotation/RequiresPermission. (accessed 2020-03-23).

[60] Google. 2020. Sign your app. https://developer.android.com/studio/publish/app-signing. (accessed 2020-07-28).

[61] Google. 2020. Use cases for package visibility in Android 11: Open URLs. https://developer.android.google.cn/preview/privacy/package-visibility-use-cases#open-urls. (accessed 2020-09-02).

[62] Google. 2021. Bookmarks permissions. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-5.1.1_r38/core/res/AndroidManifest.xml#625. (accessed 2021-06-06).

[63] Google. 2021. Bookmarks permissions descriptions. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-5.1.1_r38/core/res/res/values/strings.xml#3191. (accessed 2021-06-06).

[64] Google. 2021. Integrate the WebView API for Ads. https://developers.google.com/ad-manager/mobile-ads-sdk/android/webview.

[65] Google. 2022. Define a custom App Permission. https://developer.android.com/guide/topics/permissions/defining. (accessed 2022-09-15)).

[66] Google. 2022. Play Protect: The most widely deployed mobile threat protection service in the world. https://www.android.com/play-protect/.

[67] Google. 2022. Safe Browsing. https://transparencyreport.google.com/safe-browsing/overview.

[68] Google Play. 2021. Developer Content Policy. https://play.google.com/about/developer-content-policy/.

[69] Google Play Store. 2021. Stealth Browser - Fast Private. https://play.google.com/store/apps/details?id=com.stealthmobile.browser. (accessed 2021-06-07).

[70] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, and Haixin Duan. 2020. TextExerciser: Feedback-Driven Text Input Exercising for Android Applications. *Proc. of the IEEE Symposium on Security and Privacy (S&P)* (2020).

[71] Scott Ikeda. 2020. Brave Privacy Browser Caught Automatically Adding Affiliate Links to Cryptocurrency URLs. https://www.cpomagazine.com/data-privacy/brave-privacy-browser-caught-automatically-adding-affiliate-links-to-cryptocurrency-urls/.

[72] iNews. 2022. The number of users of the QQ browser APP has shown a downward trend in the past year. https://inf.news/en/tech/ecaf2df1401fe9818e947dc11e3591d2.html. (accessed 2022-09-15).

[73] Kobra Khanmohammadi, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury. 2019. Empirical Study of Android Repackaged Applications. *Empirical Software Engineering* 24, 6 (2019).

[74] Jeffrey Knockel, Adam Senft, and Ronald Deibert. 2016. Privacy and Security Issues in BAT Web Browsers. In *Proc. of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.

[75] Caroline Knorr. 2020. Parents' Ultimate Guide to Parental Controls. https://www.commonsensemedia.org/blog/parents-ultimate-guide-to-parental-controls.

[76] Brian Kondracki, Assel Aliyeva, Manuel Egele, Jason Polakis, and Nick Nikiforakis. 2020. Meddling Middlemen: Empirical Analysis of the Risks of Data-Saving Mobile Browsers. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[77] Platon Kotzias, Juan Caballero, and Leyla Bilge. 2021. How Did That Get In My Phone? Unwanted App Distribution on Android Devices. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[78] Felix Krause. 2022. iOS Privacy: Announcing InAppBrowser.com - see what JavaScript commands get injected through an in-app browser. https://krausefx.com/blog/announcing-inappbrowsercom-see-what-javascript-commands-get-injected-through-an-in-app-browser.

[79] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices. In *Proc. of the ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*.

[80] Douglas J Leith. 2021. Web Browser Privacy: What Do Browsers Say When They Phone Home? *IEEE Access* 9 (2021), 41615–41627.

[81] Zhuowei Li, XiaoFeng Wang, and Jong Youl Choi. 2007. SpyShield: Preserving Privacy from Spy Add-Ons. In *Proc. of the International Workshop on Recent Advances in Intrusion Detection (RAID)*.

[82] Xu Lin, Panagiotis Ilia, and Jason Polakis. 2020. Fill in the Blanks: Empirical Analysis of the Privacy Threats of Browser Form Autofill. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*. 507–519.

[83] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. 2014. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proc. of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*.

[84] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. 2019. Privacy Risk Analysis and Mitigation of Analytics Libraries in the Android Ecosystem. *IEEE Transactions on Mobile Computing* 19, 5 (2019).

[85] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. 2015. An End-to-End Measurement of Certificate Revocation in the Web's PKI. In *Proc. of the ACM Internet Measurement Conference (IMC)*.

[86] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. 2019. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[87] Meng Luo, Oleksii Starov, Nima Honarmand, and Nick Nikiforakis. 2017. Hindsight: Understanding the Evolution of UI Vulnerabilities in Mobile Browsers. In *Proc. of the ACM Conference on Computer and Communication Security (CCS)*.

[88] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android System. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[89] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. 2012. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proc. of the International Symposium on Foundations and Practice of Security (FPS)*.

[90] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps. In *Proc. of the IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*.

[91] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware. *Computers & Security* (2015).

[92] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. 2019. A Large-scale Study on the Risks of the HTML5 WebAPI for Mobile Sensor-based Attacks. In *Proc. of the Web Conference (WWW)*.

[93] Mozilla. 2021. Web APIs. https://developer.mozilla.org/en-US/docs/Web/API. (accessed 2021-06-06).

[94] Patrick Mutchler, Adam Doupé, John Mitchell, Christopher Kruegel, and Giovanni Vigna. 2015. A Large-Scale Study of Mobile Web App Security. In *Proc. of the Mobile Security Technologies Workshop (MoST)*.

[95] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. 2013. A View To A Kill: WebView Exploitation. In *Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[96] Dabid Nield. 2019. It's Time to Switch to a Privacy Browser. https://www.wired.com/story/privacy-browsers-duckduckgo-ghostery-brave/.

[97] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J Murdoch. 2016. Adblocking and Counter-Blocking: A Slice of the Arms Race. In *Proc. of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*.

[98] Yuan Niu, Francis Hsu, and Hao Chen. 2008. iPhish: Phishing Vulnerabilities on Consumer Electronics. In *Proc. of the Conference on Usability, Psychology, and Security (UPSEC)*.

[99] Ehimare Okoyomon, Nikita Samarin, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Irwin Reyes, Álvaro Feal, and Serge Egelman. 2019. On The Ridiculousness of Notice and Consent: Contradictions in App Privacy Policies. In *Proc. of the Workshop on Technology and Consumer Protection (ConPro)*.

[100] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro. 2018. A Family of Droids – Android Malware Detection via Behavioral Modeling: Static vs Dynamic Analysis. In *Proc. of the Annual Conference on Privacy, Security and Trust (PST)*.
[101] Opera. 2021. Making browsing safe from phishing. https://blogs.opera.com/security/2021/01/making-browsing-safe-from-phishing/.
[102] Charlie Osbourne. 2020. DuckDuckGo CEO clarifies favicon script use, seeks to dispel privacy worries. https://portswigger.net/daily-swig/duckduckgo-ceo-clarifies-favicon-script-use-seeks-to-dispel-privacy-worries.
[103] Elleen Pan, Jingjing Ren, Martina Lindorfer, Christo Wilson, and David Choffnes. 2018. Panoptispy: Characterizing Audio and Video Exfiltration from Android Applications. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)* (2018).
[104] Muhammad Talha Paracha, Balakrishnan Chandrasekara, David Choffnes, and Dave Levin. 2020. A Deeper Look at Web Content Availability and Consistency over HTTP/S. In *Proc. of the Network Traffic Measurement and Analysis Conference (TMA)*.
[105] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[106] Amogh Pradeep, Talha Muhammad Paracha, Protick Bhomwick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. 2022. A Comparative Analysis of Certificate Pinning in Android & iOS. In *Proc. of the ACM Internet Measurement Conference (IMC)*.
[107] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. 2017. Studying TLS Usage in Android Apps. In *Proc. of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*.
[108] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. 2018. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[109] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. 2015. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419* (2015).
[110] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *Proc. of the USENIX Security Symposium*.
[111] Charles Reis, Steven D Gribble, Tadayoshi Kohno, and Nicholas C Weaver. 2008. Detecting In-Flight Page Changes with Web Tripwires. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[112] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. Recon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proc. of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[113] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. 2018. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. In *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*.
[114] Gian Luca Scoccia, Ibrahim Kanj, Ivano Malavolta, and Kaveh Razavi. 2020. Leave my Apps Alone! A Study on how Android Developers Access Installed Apps on User's Device. In *Proc. of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*.
[115] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, and Anirban Mahanti. 2014. Predicting User Traits From a Snapshot of Apps Installed on a Smartphone. *ACM SIGMOBILE Mobile Computing and Communications Review* 18, 2 (2014).
[116] Kevin Shalvey. 2021. A lawsuit that accused Google of collecting the data of people who were using incognito mode can continue, said a federal judge. https://www.businessinsider.com/google-lawsuit-incognito-mode-user-data-privacy-can-continue-judge. (accessed 2021-06-07).
[117] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. 2018. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)* (2018).
[118] Shiwangi Singh, Rucha Gadgil, and Ayushi Chudgor. 2014. Automated Testing of Mobile Applications using Scripting Technique: A Study on Appium. *International Journal of Current Engineering and Technology (IJCET)* (2014).
[119] Statista. 2021. Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 1st quarter 2021. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/. (accessed 2021-06-08).
[120] Thomas Steiner. 2018. What is in a Web View? An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser. In *Proc. of the Web Conference (WWW)*.
[121] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, et al. 2015. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.
[122] Giorgos Tsirantonakis, Panagiotis Ilia, Sotiris Ioannidis, Elias Athanasopoulos, and Michalis Polychronakis. 2018. A Large-scale Analysis of Content Modification by Open HTTP Proxies. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[123] Junia Valente and Alvaro A Cardenas. 2017. Security & Privacy in Smart Toys. In *Proc. of the Workshop on Internet of Things Security and Privacy (IoT S&P)*.
[124] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.
[125] Pepe Vila and Boris Köpf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *Proc. of the USENIX Security Symposium*.
[126] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. 2018. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *Proc. of the ACM Internet Measurement Conference (IMC)*.
[127] Haoyu Wang, Xupu Wang, and Yao Guo. 2019. Characterizing the Global Mobile App Developers: A Large-Scale Empirical Study. In *Proc. of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*.
[128] Daoyuan Wu and Rocky KC Chang. 2014. Analyzing Android Browser Apps for file:// Vulnerabilities. In *Proc. of the International Conference on Information Security (ISC)*.
[129] Zhiju Yang and Chuan Yue. 2020. A Comparative Measurement Study of Web Tracking on Mobile and Desktop Environments. *Proc. of the Privacy Enhancing Technologies Symposium (PETS)* (2020).
[130] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. 2016. Tracking the Trackers. In *Proc. of the Web Conference (WWW)*.
[131] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. 2013. Real-time Bidding for Online Advertising: Measurement and Analysis. In *Proc. of the International Workshop on Data Mining for Online Advertising (ADKDD)*.
[132] Ahsan Zafar, Aafaq Sabir, Dilawer Ahmed, and Anupam Das. 2021. Understanding the Privacy Implications of Adblock Plus's Acceptable Ads. In *Proc. of the ACM Asia Computer and Communications Security (ASIA CCS)*.
[133] Zicheng Zhang, Daoyuan Wu, Lixiang Li, and Debin Gao. 2021. On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[134] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[135] Zynamics. 2021. BinDiff. https://www.zynamics.com/bindiff.html.

Table 7: Data collected during dynamic analysis.

Analysis Component (Technique)	Collected Data
Director (`adb logcat`)	Logcat dump
Director (`adb screencap`)	Mobile device screenshots
Director (`adb-sync`)	Files on external storage
Director (`adb`) + Mobile Device (`fsmon`)	Filesystem changes on external storage
Mobile Device (`tripwire`)	Browser DOM
Gateway (`mitmproxy`)	TLS traffic
Gateway (`tcpdump`)	All network traffic

A WEBSITE SELECTION

- **Honeypage.** To understand how browsers handle referral links (whether they modify them to generate revenue by users' purchases), social media plugins, and ads (whether they block them to enhance users' privacy), we use a honeypage. This honeypage is inspired by Tsirantonakis *et al.* [122] and contains *fake advertisements*, an Amazon referral link, and social media plugins. The source code for this page can be found on Pastebin (https://pastebin.com/qd25cNmC) and in our artifact repository.
- **Permissions page.** We extend a permissions test page [33] to add tests for an exhaustive list of JavaScript WebAPIs. We also modify this site to send result data to our testbed and use it to test access to device sensors.

- **Domain without protocol.** To see if browsers pick *http://* or *https://* by default when a user visits a domain without specifying a protocol, we use a webpage that supports both these protocols and omit mentioning the protocol. The webpage is under our control and performs no redirection but serves content over the requested protocol.
- **HTTPS webpage.** To understand the nature of TLS connections a browser uses, including TLS versions, ciphers supported *etc.*, we use a HTML webpage hosted on a server that supports HTTPS.
- **Popular webpages.** To understand how browsers handle regular websites, we test the 16 websites listed in Table 8, one from each category of Alexa's category rankings [15]. (Note that this feature was retired in September 2020.)

Table 8: Popular websites from Alexa's category rankings.

Category	Website
Adult	https://www.xvideos.com/
Business	https://www.office.com/
Regional	https://www.yahoo.com/
Games	https://m.twitch.tv/
Health	https://www.nih.gov/
Society	https://www.patreon.com/
Home	https://finance.yahoo.com/
Science	https://www.researchgate.net/
News	https://www.reddit.com/
Recreation	https://www.booking.com/
Reference	https://stackoverflow.com/
Shopping	https://www.amazon.com/
Sports	https://www.espn.com/

B BROWSERS SHARING PII

Table 9 lists all the observed destinations (manually grouped by the organization that owns them), as well as the type of data that they receive.

C BROWSING HISTORY EXPOSURE

The remaining rows of Table 6, listing destinations that receive browsing history along with PII (if any).

Table 9: Entities receiving PII from browsers (* First party).

Destination	AdID	Loc	MAC_D	Total
Firebase	33			33
Facebook	24	2		24
Alibaba	1	14	2	18
AppsFlyer	13	1		13
Verizon	9			9
Baidu		7		7
Adjust	7			7
Twitter	6			6
umsns	1		1	6
Appnext	6			6
OneSignal	5			5
StartAppService	4	1		4
Google Analytics	3	1		4
AppsGeyser	3			3
StartApp	3			3
ctobsnssdk	1		3	3
toutiao		2		3
Mobvista	1	1		2
Microsoft		2		2
360		2		2
Taboola	2			2
smardroid	1			1
Google devicecheck		1		1
Yahoo Search		1		1
Cloudflare Certcheck				1
Tencent		1		1
Mindworks	1			1
Urora				1
impression	1			1
Brave		1		1
Opera*	1			1
Amap		1		1
Kuaishou		1		1
yladm			1	1
1look			1	1
AdColony	1			1
UCWeb*		1		1
uodoo		1		1
UCWeb	1		1	1
yohoads	1			1
Flymobi	1			1
suibyuming	1	1		1
oakmastering	1			1
cloudmobi	1			1
MiOTA*		1		1
Amazon	1			1
Yandex*	1			1
Mail.ru*	1			1
Amplitude	1			1
Baidu*		1		1
tclclouds			1	1
Xiaomi		1		1
Opera	1			1
Vserv	1			1
Vmax	1			1
duapps	1			1
Total	**98**	**41**	**8**	**135**

Table 10: Number of unique browsers sharing browsing history and PII together, allowing other parties to link the history to a unique user (* First party).

Destination	# of Browsers	Location	AdID
Oupeng	1		
Baidu	1	1	
ilovegame	1		
baoruan	1		
FDVR*	1		
Google Datamixer	1		
Kiddoware*	1		
Maxthon*	1		
Orbitum*	1		
Fillr	1		
Tencent*	1		
fddm	1		
MiOTA*	1	1	
Bing	1		
Mail.ru*	1		1
Aonbrowser	1		
Google Analytics	1		
Orbitum	1		
Total	**37**	**4**	**10**