This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration

Michalis Diamantaris
FORTH
Heraklion, Crete, Greece
diamant@ics.forth.gr

Serafeim Moustakas
FORTH
Heraklion, Crete, Greece
mustakas@ics.forth.gr

Lichao Sun
Lehigh University
Bethlehem, Pennsylvania, USA
james.lichao.sun@gmail.com

Sotiris Ioannidis
Technical University of Crete
Chania, Crete, Greece
sotiris@ece.tuc.gr

Jason Polakis
University of Illinois at Chicago
Chicago, Illinois, USA
polakis@uic.edu

ABSTRACT

Mobile sensors have transformed how users interact with modern smartphones and enhance their overall experience. However, the absence of sufficient access control for protecting these sensors enables a plethora of threats. As prior work has shown, malicious apps and sites can deploy a wide range of attacks that use data captured from sensors. Unfortunately, as we demonstrate, in the modern app ecosystem where most apps fetch and render third-party web content, attackers can use ads for delivering attacks.

In this paper, we introduce a novel attack vector that misuses the advertising ecosystem for delivering sophisticated and stealthy attacks that leverage mobile sensors. These attacks do *not* depend on any special app permissions or specific user actions, and affect all Android apps that contain in-app advertisements due to the improper access control of sensor data in WebView. We outline how motion sensor data can be used to infer users' sensitive touch input (e.g., credit card information) in two distinct attack scenarios, namely *intra-app* and *inter-app data exfiltration*. While the former targets the app displaying the ad, the latter affects *every* other Android app running on the device. To make matters worse, we have uncovered serious flaws in Android's app isolation, life cycle management, and access control mechanisms that enable persistent data exfiltration even after the app showing the ad is moved to the background or terminated by the user. Furthermore, as in-app ads can "piggyback" on the permissions intended for the app's core functionality, they can also obtain information from protected sensors such as the camera, microphone and GPS. To provide a comprehensive assessment of this emerging threat, we conduct a large-scale, end-to-end, dynamic analysis of ads shown in apps available in the official Android Play Store. Our study reveals that ads in the wild are already accessing and leaking data obtained from motion sensors, thus highlighting the need for stricter access control policies and isolation mechanisms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485366

CCS CONCEPTS

• **Security and privacy → Mobile platform security**.

KEYWORDS

Android in-app ads; WebView; mobile HTML5; sensor attacks;

ACM Reference Format:
Michalis Diamantaris, Serafeim Moustakas, Lichao Sun, Sotiris Ioannidis, and Jason Polakis. 2021. This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3485366

1 INTRODUCTION

The ubiquitous nature of mobile devices and the plethora of rich functionalities they offer has rendered them an integral part of our daily routines. The advent of smartphones has also transformed how users experience and interact with web services, and Android has become the most prevalent mobile operating system currently powering over 85% of devices worldwide [51]. Android's app ecosystem is dominated by free apps and in-app advertisements have become the de facto source of revenue for app developers [6, 53]. Even major tech companies heavily rely on mobile advertising, with Facebook earning 94% of its ad revenue from mobile devices [109].

Recently, mobile motion sensors (e.g., accelerometer and gyroscope) have started playing an increasingly important role in the mobile advertising ecosystem, as motion-based ads allow for more interactivity and higher user engagement, leading to increased revenue [103]. Even though mobile sensors provide functional diversity that is reshaping how users interact with and consume ads, they also introduce a significant security and privacy threat. In more detail, a plethora of prior studies have demonstrated that data obtained from mobile sensors can be used for identifying and tracking users across the web [11, 12, 17, 26–29, 33, 39, 45, 50, 52, 61, 66, 82, 86, 115, 116], inferring physical activities [45, 48, 59, 66, 82] and in more severe scenarios inferring users' touch screen input [18, 47, 59, 70, 97, 111]. Das et al. [25] also demonstrated that web scripts accessing mobile sensors allow for stateless tracking on the mobile web, while Marcantoni et al. [57] described how a plethora of mobile sensor-based attacks that previously required a malicious app to be installed can easily migrate to the mobile web using the HTML5 WebAPI.

However, as users spend the majority of their browsing time within mobile apps [2], mobile ads will often reach their audience through in-app ads. These ads are shown inside the context of a mobile app and allow developers to release their apps for free while earning revenue from the embedded ads. Unfortunately, this symbiotic relationship, combined with ads' ability to access mobile device sensors, creates the opportunity for delivering a variety of sensor-based attacks. While prior work has proposed separating the privileges offered to applications and advertisements [75], Android has not adopted such an approach. To make matters worse, mobile motion sensors are *not* guarded by a specific permission and are freely accessible to in-app ads. Comparatively, the iOS operating system blocks in-app ads from accessing motion sensors or explicitly requests user approval when websites attempt to access them. To the best of our knowledge no prior study has explored in-depth the security risks posed by Android's access control and permission system policies that govern how in-app ads can use mobile sensors.

In this paper we introduce a novel attack vector that misuses the ad ecosystem for delivering sophisticated and stealthy attacks. Our threat model captures a malicious actor delivering a seemingly legitimate mobile ad campaign, targeting benign mobile apps downloaded from the official Play Store and targeting the rich data returned from motion sensors to perform a plethora of sensor-based attacks, including stealing login credentials and credit card information. While in practice *any* sensor-based attack demonstrated in prior work is feasible, we focus on inferring the user's touch input due to the severe risk posed to users.

Our empirical investigation captures two separate attack scenarios for inferring sensitive data, namely intra- and inter-application data exfiltration. In the intra-application attack scenario, a motion-based ad is able to infer users input when ads are "co-located" with `Views` that contain sensitive input information. Even though Google's ad placement policies [7] instruct developers to not show ads in `Views` that contain sensitive information, we found that developers do not always adhere to safe practices. More importantly, we have identified a flaw that allows us to target apps even when the ads are not "co-located" with the sensitive data. In more detail, Google's interstitial ads can be easily misused for capturing sensitive input even if they are not displayed on top of sensitive `Views`, since the JavaScript code of interstitial ads is executed from the moment the ad is preloaded up to the moment the user clicks the corresponding application element. As such, even if users are exploring other parts of the app when entering sensitive content (e.g., billing information for in-app purchases) they remain vulnerable.

Next, our inter-application attack scenario significantly expands the attack surface, as it allows attackers to target *any other app currently running* on the device, if the app showing advertisements holds the `SYSTEM_ALERT_WINDOW` permission. Specifically, if the host app has been granted the aforementioned permission and an ad-related WebView is attached to the `WindowManager`, ads are essentially allowed to execute JavaScript in the background, therefore making *every* other Android app vulnerable to sensor-based side-channel attacks. Despite the known risks associated with this permission [41], in certain cases (i.e., [32, 44]) it is still automatically granted to apps installed from the Play Store. Our experiments reveal that and it is obtained by 9.28% (416 out of 4,478) of the most popular apps. To make matters worse, we discovered a critical security flaw in Android that prevents the user from killing the host app from the task manager, while users are deceived as the host app is no longer shown in the list of background apps despite not having been terminated.

Our empirical analysis demonstrates that in-app advertisements not only have the potential to access mobile sensors but are also able to silently leak that data. Due to the severe implications of these attacks, we build a novel automated framework for analyzing in-app advertisements, which provides an in-depth view of requests to access mobile sensors and distinguishes sensor access requested by in-app advertisements from those requested by the app's functionality. We bridge the semantic gap for identifying the origin of sensor calls by combining low-level hooks at the Android layer with high-level hooks at the Network layer. We leverage our framework to conduct a study of in-app advertisements in the wild, by analyzing how they access mobile sensors across 4.5K of the most popular apps obtained from the official Google Play Store. We conduct a longitudinal study by periodically repeating the dynamical analysis of the apps in our dataset over a period of several months, so as to capture a more varied collection of ad campaigns. To further diversify our study's view of the ad ecosystem, we repeat a set of experiments across different countries using VPN services. Our study reveals that a large number of apps (27.28%) display in-app ads that perform some form of device tracking or fingerprinting, we also find several instances of ads accessing and exfiltrating motion sensor values to third-parties without the user's knowledge or consent. As the use of motion sensors in advertisements is gaining traction, we expect such invasive advertisements to become far more common in the near future.

In summary, we make the following research contributions:

- We introduce a novel attack vector that abuses the advertising ecosystem for stealthily delivering attacks that abuse mobile sensors, magnifying the impact and scale of sensor-based attacks. Our empirical analysis reveals several flaws in Android's isolation, life cycle management, and access control mechanisms that can be exploited for increasing the attack's coverage and impact.
- We conduct an extensive investigation of in-app ads accessing mobile sensors in the wild and identify several instances, highlighting the threat posed by our attacks. To facilitate additional research we publicly share our code.
- To mitigate our attacks, we propose a set of access control policies and guidelines for the Android OS, app developers, and ad markets. We have disclosed our findings to Android's security team, who acknowledged the potential for abuse.

2 BACKGROUND

This section provides background information and technical details regarding the display of in-app ads. We also discuss pertinent mobile sensor-based attacks demonstrated in prior work.

Mobile Sensors. A plethora of studies (e.g., [13, 27, 42, 58, 61, 66, 79, 108, 113, 114]) have demonstrated that apps can use the data acquired from sensors like the Accelerometer, Gyroscope and Light sensor for various sophisticated and often highly accurate attacks [59], without requiring any permission from the operating

system or the user. Researchers previously presented a taxonomy of these sensor-based attacks [34, 57], where attacks are classified in four major categories; Physical Activity Inference, Acoustic Attacks, Digital Activity Inference and User Tracking. A notable example is the Touchscreen Input Attack from the Digital Activity Inference category that shows how sensors can be used to infer what the user is typing [15, 18, 19, 47, 59, 62, 70, 77, 111]. This attack is made possible by the changes in the screen's position and orientation, and the motion that occur while the user types.

Ads, WebViews & Sensors. Advertisements are usually written in JavaScript, which enables the use of a plethora of powerful API calls. Amongst these API calls are the HTML5 functions responsible for accessing mobile motion sensors. Specifically the accelerometer sensor is accessed using the `DeviceMotionEvent.acceleration` [14] `DeviceOrientationEvent` [88] APIs, while the `DeviceMotionEvent.rotationRate` [63] API gives access to the gyroscope sensor. Moreover, the Generic Sensor API [106] bridges the gap between native and web applications, is not bound to the DOM (nor the Navigator or Window objects) and can be easily extended with new sensor classes with very few modifications.

Recent work [25, 34] reported that many websites and third-party scripts access the information provided by these sensors when accessed through a mobile browser. In practice, the mobile advertising ecosystem has two different paths for displaying advertisements to users, either through the advertisements that are embedded in a website that is accessed using a mobile browser app (i.e., website-ads) or through embedded advertisements. The latter, hereby referred to as *in-app ads*, are displayed inside the context of a mobile application with the use of an Android WebView [43]. WebView is based on the Chrome/Chromium and WebView objects are able to display web content as part of an activity layout. Specifically, WebView for Android 7 - 9 is built into Chrome, while in newer versions Chrome and WebView are separate apps. Even though WebView lacks some of the features of a full-fledged browser, it can evaluate JavaScript (e.g., `evaluateJavascript()`), interact with cookies (e.g., `setCookie()`/`getCookie()`) and access a plethora of mobile HTML5 APIs. Additionally, since WebView exists in the same context as the actual application's process, it also shares all of the host application's privileges (including normal and dangerous permissions). To verify this, we created a mock app and separately executed all HTML5 APIs that access mobiles sensors. We found that WebViews are able to call every mobile sensor. Moreover, we found that all mobile sensors (except GPS and Camera) do not require the host app to hold any specific Android permissions. Furthermore, if the app holds the appropriate permissions for additional capabilities, then WebView automatically and without any interaction gains access to these as well.

3 MOTIVATION AND EXPLORATION

Here we describe some initial experiments and findings that motivate our attack and our subsequent large-scale study.

Permissions and access control. In the first experiment we verify that Android's access control policies and permission management allow in-app advertisements to access motion sensors and leak these values using common network techniques. We set up a test bed consisting of an actual Android device playing the role of the victim, while a Raspberry Pi was used for deploying an Ad Server that will deliver the invasive advertisement. We deployed a simple test application on our device, which includes an embedded advertisement rendered within a WebView. In our experiments the ad is successfully displayed and able to access the motion sensors, while we can send the sensor values back to the Raspberry server through an `XMLHttpRequest` or the `GET/POST` methods. We performed this experiment twice, to verify that ads are not limited to a one-time sensor reading but can also collect and exfiltrate continuous sensor readings.

Sensor data leakage in practice. During a preliminary analysis of ads in the wild, we identified an ad campaign accessing motion sensors and also sending that data to a remote server. Specifically, we identified an in-app advertisement from a major telecommunication provider accessing motion sensors even if the user did not interact with the ad, and leaking those values to a DoubleVerify domain through a `GET` request. Since DoubleVerify provides online media verification and campaign effectiveness solutions, we believe that this could potentially be used for bot detection and ad fraud prevention. Nonetheless, even though we can not assign (nor disprove) malicious or invasive intentions behind this specific case, we believe that users should explicitly be given the option to allow or deny access to their sensor data.

Publishing sensor-based ads. Next, we wanted to investigate whether any business-level or technical "countermeasures" exist in practice, to prevent ads from accessing sensor data. Prior to conducting this experiment, a description of our study and experimental protocol was submitted to and approved for exemption by our university's Institutional Review Board (IRB). Appendix A includes a detailed ethical analysis of our experiment. For this exploratory experiment, we signed a contract with a DSP and published an in-app ad campaign accessing motion sensors. At the end of the campaign, which reached 13K impressions at a cost of ~ 15€, we obtained a report from the DSP with information for the ad campaign (e.g., apps displayed, impressions, clicks, etc.). It is important to note that in this experiment we did *not* gather any user information nor did we exfiltrate any sensor values. Furthermore, the DSP report contains only aggregate statistics and information, which cannot be used to identify or infer any personal user data.

Summary. Based on our findings we argue that it is trivial for privacy-invasive entities and cybercriminals to abuse the mobile ad ecosystem for exfiltrating data by delivering advertisements that capture the rich information provided by these sensors.

4 THREAT MODEL & ATTACK SCENARIOS

Here we introduce our threat model and provide details on how we exploit flaws in Android's isolation, life cycle management, and access control mechanisms to expand the attack surface and magnify our impact and coverage. We illustrate our findings through two distinct scenarios, namely *intra-app* and *inter-app* data exfiltration, and detail how attackers can exfiltrate billing information typed by the user in popular and widely available Android apps.

4.1 Threat Model

We demonstrate a new attack vector that abuses the mobile advertising ecosystem for delivering a mobile sensor-based attack

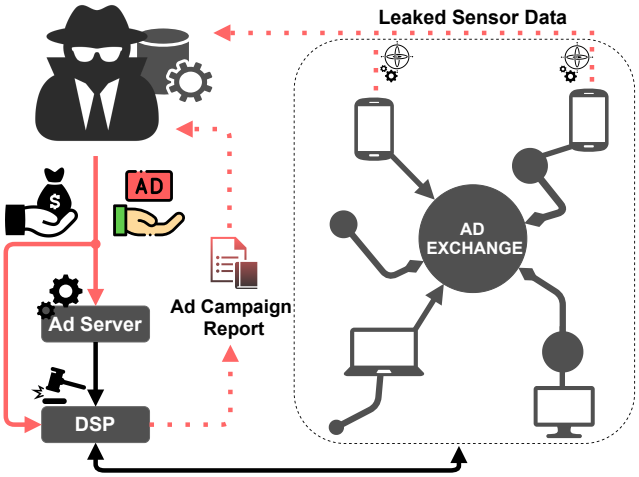

Figure 1: Overview of our attack vector. A malicious actor publishes an ad campaign that accesses mobile sensors, for delivering sophisticated and stealthy attacks.

which affects every Android device (71.93% of mobile users worldwide [100]). As opposed to prior attacks, our attack vector does not require any malicious app to be installed on the device, nor does it rely on a user visiting a malicious website. Furthermore, as these are embedded in-app advertisements, they cannot be blocked through ad-blocking browser extensions. Our presented attack uses in-app advertisements to obtain the device's motion sensor readings, allowing the attacker to stealthily infer sensitive user information including any information that is typed on the screen (e.g., credentials, credit card information, and pin numbers). While we use the inference of user input as our driving scenario, since it is also the most frequently feasible sensor-based attack [34], our attack vector can be tailored for any sensor-based attack. This is possible due to the lack of any restriction in accessing the device's sensors (except for the camera and microphone) through an Android permission or a user prompt.

Figure 1 provides an overview of our attack, where the attacker creates a seemingly-benign mobile ad campaign. Given that accessing sensor-based data is an emerging trend in mobile ads, with up to a nine-times higher engagement rate than simple mobile banners [74], the attacker can release their ad campaign through major legitimate ad platforms. Since ad campaigns can be tailored to specific needs, the attacker can instruct the Ad Server or DSP to only display the ad on mobile devices and, specifically, as an in-app ad. The attacker can even specify a set of select apps to maximize the impact of the attack, as we describe in §4.2.

The actual context of the advertisement does not really matter as our attack does not require the user to click on the ad or interact with it in any way. The advertisement will go through the normal process of publishing and eventually be displayed as an in-app advertisement across different apps. When the advertisement reaches the user's device, the JavaScript code leverages the appropriate HTML5 API calls for accessing the motion sensors and then exfiltrates this data to a server controlled by the attacker.

Table 1: Feasible intra- and inter-app data exfiltration scenarios of in-app ads that access mobile sensors. In the inter-app scenario, a (✓) denotes that access is still granted after the corresponding user action.

		Motion sensors	CAM P.O.1	MIC P.O.1	GPS P.O.1 | P.O.2
without SYSTEM_ALERT_WINDOW		Intra-app data exfiltration			
with SYSTEM_ALERT_WINDOW		Inter-app data exfiltration			
User Actions	**Device Lock**	✓	✓	✓	✓
	UI Swipe	✓	✓	✓	✗ | ✓
	Swipe + Lock	✗	✗	✗	✗
	Force Stop	✗	✗	✗	✗

4.2 Intra & Inter-Application Attacks

Here we provide technical details about two distinct attack scenarios that can be used to exfiltrate sensitive data from an Android device, namely intra and inter-application data exfiltration. We present notable examples for exfiltrating billing information (e.g., credit card number, paypal account, etc.) for both attack scenarios by targeting (i) the Google Play Billing Library, widely used for in-app purchases in popular applications, and (ii) the official Play Store app. Table 1 summarizes the app permission requirements (if any) and whether sensor access is granted for different mobile sensors in each attack scenario. CAM, MIC and GPS require that the app holds the appropriate permissions. Apps targeting API versions greater than API 28, also need `ACCESS_BACKGROUND_LOCATION` for accessing GPS in the background. Additionally, since API 30 allows different options for dangerous permissions, we tested the permission option "Allowed only while in use" (P.O.1) for CAM and MIC. For GPS we tested "Allowed only while in use" (P.O.1) and "Allowed all the time" (P.O.2). The *User Actions* rows denote whether sensor access by in-app ads continues after specific user interactions (e.g., UI Swipe) for the inter-application data exfiltration scenario.

Intra-Application Data Exfiltration. In this attack we can capture the input data of the Android app that is displaying the sensor-capturing advertisement. This can be done through two different techniques, which we describe next, or using a combination of both. In practice, advertisements are displayed inside a `WebView` object, which is responsible for fetching and loading all the ad resources from the web. Each WebView is displayed as part of an activity layout and is co-located with other `View` objects. When the WebView has finished loading the ad's content, the appropriate HTML5 APIs are executed and the advertisement can capture touch input from the co-located `View` objects. This is extremely important since many `Views` in Android apps contain sensitive input. We note that apart from the WebView object responsible for displaying the ad, other WebViews may coexist for handling other app functionality such as logging in or completing a payment. Therefore, any part of the application that is attached to the `View` that contains the ad is vulnerable for input hijacking. Even though it is considered good practice to not show ads in `Views` with sensitive input, in our analysis we found several cases of apps violating this guideline.

Interestingly, the attack's coverage significantly increases if the application is using Google's *interstitial ad placements*. Interstitial ads are interactive advertisements that cover the interface of their host app. These ads appear between content or activities and allow for a more natural transition. In order to achieve this effect, interstitial ads preload the advertisement's content before being displayed on the screen. Our empirical analysis revealed that Google's library for interstitial ad placements allows interstitial ads' code to execute from the moment they are preloaded until the user has closed the advertisement. Since an interstitial ad will be displayed only when a specific element of the app is pressed (and they can be attached to any element) the code of the advertisement will continue running until this specific element is pressed. As such, the user may be exploring other parts of the app, including `Views` with sensitive content, while the interstitial ad is capturing the motion sensor data. It is important to emphasize that loading the interstitial ad (i.e., `loadAd()`) as early as possible to ensure it is available during the `show()`, is encouraged by the developer documentation [30].

Furthermore, our experiments with Google's library for interstitial ad placements revealed that these ads continue to execute code not only in different `Views` but also in different `Activities` within the same app. To make matters worse, the code will continue executing even if the application `Activity` that initiated the preloading mechanism *has been destroyed* (e.g., `activity.finish()`). As such, interstitial ads not only increase the attack's robustness, but also increase the attack's stealthiness since even more security-cautious users that do not input sensitive data when ads are being displayed would be deceived. As we discuss in §6, our measurements reveal that the use of interstitials is commonplace in popular apps.

Case Study - Play Billing library. Apart from login credentials, an attacker using the techniques described above can also target apps that offer in-app purchases in order to steal the user's billing information. Since in-app purchases are the most common monetization model, with users spending $380 billion worldwide [99], apps that integrate them are ideal candidates for this attack. As such, we tested Google's Play Billing library version 2, as well as the latest version 3.0.3 and found that in-app ads can capture motion sensor data while the user is providing input in any of the available billing options of the library (e.g., credit card, Paypal and Paysafe).

Inter-Application Data Exfiltration. Android apps are executed in a sandbox environment and in different processes to prevent unintended data leakage from one app to another. WebViews, by default, are attached to the app's UI thread and are not able to execute code in the background if the user switches apps. Nonetheless, Android offers a mechanism for executing code in the background, specifically, by attaching a `View` in the `WindowManager`. Surprisingly, we found that the same applies for WebViews; if the host app holds the `SYSTEM_ALERT_WINDOW` permission for it's core functionality, then an ad-related WebView can be configured to run in the background and continue accessing motion sensors even if the user switches apps. The `SYSTEM_ALERT_WINDOW` permission, according to the official Android SDK [92], falls into a special category of permissions that require the user to explicitly grant it when requested (the app opens the Android Settings for this specific app and informs the user of the permission's abilities). However, if an application is downloaded directly from the official Google Play, then this permission is *granted automatically and without any user interaction*. Specifically, as mentioned in [44], an app's developer can issue a request to the Google Play App Review team so that the `SYSTEM_ALERT_WINDOW` permission is granted automatically. Additionally, as mentioned in [32], if apps have the `ROLE_CALL_SCREENING` and request the `SYSTEM_ALERT_WINDOW` they are also automatically granted the permission. For instance, the `com.truecaller` app has this functionality and if during the initial setup the user sets the app as the default caller id and spam app, then the permission is automatically granted. Moreover, during this step the app falsely informs the user that no permissions are needed.

We argue that such instances of relaxed policies, not only confuse users and developers alike but can lead to misuses with severe ramifications. Furthermore, even experienced users that can identify suspicious apps that were automatically granted the permission can be mislead. This is especially true for popular apps that need this permission for showing pop up messages and providing additional functionality on top of other apps. Applications requesting this permission include Skype, Facebook Messenger and Viber. We note that Viber, a very popular messaging app that is used by banks for sending two-factor authentication codes, contains ads and is susceptible to our inter-application data exfiltration attack. Furthermore, our manual investigation revealed that several apps request this permission for their core functionality. For example, apps request this permission for playing videos in the background while the user is performing other tasks. These apps attach a WebView in the `WindowManager` and are vulnerable to the inter-application scenario, since the embedded in-app ads (including video ads) have access to the motion sensors. To better illustrate the magnitude of this attack scenario, we note that if one application holds this specific permission and is displaying ads, *all* apps installed on the device can be compromised and are vulnerable to input hijacking. Even banking apps that use the `WindowManager.LayoutParams.FLAG_SECURE` option, a security feature to treat the contents of the window as "secure" [93], are vulnerable to sensor-based inter-app side channel attacks. As we describe later on, we found that 9.28% of the apps in our dataset hold this permission, and 69.95% also display ads.

To make matters worse, we have also identified a security vulnerability that further magnifies the attack's impact. In more detail, when an app's WebView is executing content in the background, the Android operating system *will not* terminate the code even if the user "kills" the host application using the traditional UI swipe method. This issue is further complicated and the deceptiveness of the attack is enhanced by the fact that the app will no longer appear in the list of background apps, even though the application and the WebView still exist and are executing code.

In fact, as can be seen in Table 1, we have only identified two ways for the user to successfully close the app and terminate any background executed code. One way is to navigate to the Android Settings, select the app and then select the force-stop option. Another way for stopping all app activities is to perform the UI swipe for the host app and also lock the device. We tested this abnormal functionality on Pixel devices running (AOSP) API 29 and API 30 using a mock app with a WebView that accesses mobile sensors using HTML5 WebAPIs. The Pixel 4 device had Android v11 and the latest security updates at the time of writing (April 2021).

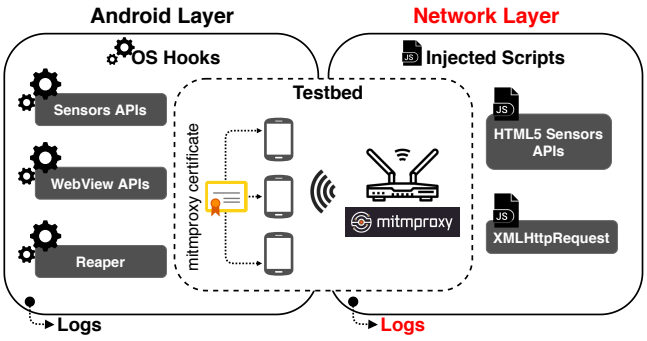

Figure 2: Overview of our framework's infrastructure. The combined components of both layers provide an in-depth view of requests to access mobile sensors. Components in the Android layer (left) are responsible for monitoring system API calls, while components in the Network layer (right) monitor JavaScript calls and network traffic.

Case Study - Play Store. Even though many popular apps contain sensitive input information, one app that is pre-installed on *every* Android device is widely used and contains sensitive input information. Specifically, we tested the official Play Store app and found that through the in-app ads of background apps, attackers can capture the motion sensor values while the user is typing billing information in the Play Store's "Payments & Subscriptions" section.

5 SYSTEM DESIGN AND IMPLEMENTATION

Motivated by our preliminary findings, we conduct a large-scale, end-to-end automated study of in-app advertisements accessing mobile sensors. We dynamically analyze applications with in-app advertisements and monitor access to *all* available mobile sensors and record any potential leakage of this type of data.

One of the challenges for dynamically analyzing in-app ads is being able to differentiate sensor accesses issued by embedded ads from those that originate from the app's core functionality. Our framework obtains an in-depth view of sensor data access by combining logs from two different layers. As can be seen in Figure 2, for each of these layers (Android and Network) we monitor different API calls using multiple components. At the Android layer we monitor system call using modules from the Xposed framework [91], while at the Network Layer we monitor HTML5 WebAPI calls using injected JavaScript code. Our testbed consists of three Nexus 5x devices, running Android 7.1.1 that we configured with the `mitm` proxy's root certificate that allows us to intercept HTTPS traffic. In Appendix A, we provide additional technical details concerning our methodology for monitoring in-app advertisements.

Android Layer. This part of our framework monitors apps' access to sensors by intercepting Android system calls using a custom Xposed module that detects and hooks requests to sensor-specific Android API calls. Since values from the accelerometer and the gyroscope are expected to change when the device is used by an actual human and because motion sensors have been used by apps to evade analysis or hide suspicious activity [71], we made our infrastructure more robust by intercepting the values returned by certain sensors and slightly modifying them within appropriate and legitimate bounds. We identify ad hyperlinks inside WebViews by hooking the appropriate WebView and Chromium APIs. Additionally, we leverage functionality from prior work [35] for (i) verifying which of the sensor-specific Android API calls are permission-protected and (ii) traversing the app's graph using a breadth-first traversal for achieving high coverage.

Network Layer. The other major component of our framework employs a transparent proxy server for intercepting all network traffic by using `mitmproxy` [23] and injecting code for intercepting JavaScript calls. We used the `javascript-hooker` Node.js module [16] which allows us to hook any JavaScript function called inside a WebView and intercept the method to be called and its arguments. Using this approach we hook all the functions that access and retrieve mobile-specific sensor data through the official mobile HTML5 WebAPI [40]. We also monitor any calls of the `XMLHttpRequest` function, since in-app advertisements can leak data using this method. The injected JavaScript logs all information to the console. To log this information to the Android `logcat`, we created an Xposed module and during run-time hooked the `android.webkit.WebChromeClient.onConsoleMessage()` function and performed any necessary instrumentations for redirecting any console messages to the `logcat` along with other useful information, such as the package name of the app being tested. Using this technique we can also verify that network flows and JavaScript accessing motion sensors or other tracking related WebAPIs originate from the app being tested. Table 7 in Appendix A, provides a complete list of all the HTML5 WebAPIs monitored by our system.

By combining hooks from low-level sensor-related system calls as well as JavaScript calls from the network, we can successfully distinguish sensor access requested by in-app advertisements from those requested by the app's functionality. Specifically, if we identify a sensor system call from the OS without a corresponding sensor API call at the network layer, then we can deduce that the app itself requested access to this sensor. On the contrary if we identify both a sensor call (e.g., for the Accelerometer) at the network layer and the Android layer then we can successfully deduce that the in-app advertisement accessed the mobile sensor. It is worth noting that in cases where both the application and the in-app advertisement perform the same sensor call, our analysis is not affected. Finally, to avoid contamination from other apps accessing sensors, we analyzed each app individually and limited other background app activities using the adb toolkit. We verified that our framework behaves as expected by executing separately all HTML5 APIs that access mobile sensors using a mock application.

6 LARGE SCALE MEASUREMENT STUDY

Here we present our findings from our large-scale study on the use of HTML5 WebAPI calls by embedded in-app ads in the wild.

6.1 Dataset & Experimental Setup

App selection. Our main app dataset consists of free apps downloaded from the official Google Play market. First, we selected the top 100 apps (or as many as were available) from 61 categories. Next, using two lists of websites that access mobile sensors [25, 34], we tried to download the corresponding mobile app if it exists in

Table 2: Number of apps containing in-app ads accessing WebAPIs, analyzed across different countries.

WebAPI	#Apps per country					
	US	RU	IN	UK	DE	GR
Mobile-specific						
window - devicemotion	4	2	3	4	3	11
window - deviceorientation	0	1	0	1	1	1
window - orientationchange	3	1	2	8	4	29
screen - change	0	0	0	1	1	1
getBattery	1	0	1	3	4	10
General						
XMLHttpRequest.send	20	19	16	20	87	1056
getTimezoneOffset	8	2	2	5	82	958
toDataURL	0	0	0	0	0	7
getContext	4	3	3	2	7	63
WebGLRenderingContext	0	0	0	1	2	6
setItem	2	1	0	2	92	1,171
getItem	1	1	1	2	81	1,026
removeItem	2	1	0	1	92	1,149
key	0	0	0	0	0	14
createElement(canvas)	7	17	2	8	13	65

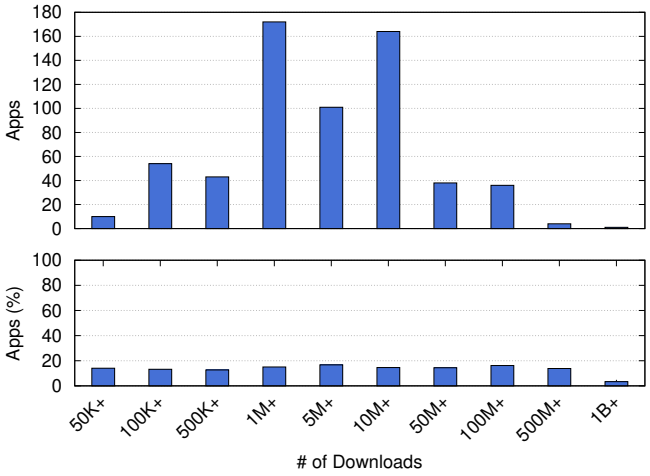

Figure 3: Number (top) and ratio (bottom) of apps with Google's interstitial ad placements, per ranking bin.

the official store. Overall, we downloaded 4,478 apps from Google Play using the Raccoon [1] framework.

Analysis and location. Since we cannot have a-priori knowledge about when a specific ad campaign that accesses motion sensors will run, nor can we know which apps may be targeted by such advertisements, we opt for using a large number of apps from different categories, which we periodically re-examine over the course of eight months (9/01/2020 - 4/30/2021). Furthermore, to avoid biasing our study by constraining it to ads displayed in a specific country, since policies and legislation may govern their behavior and differ across jurisdictions, our infrastructure leverages a VPN service for simulating users browsing from different countries. For our experiments using the VPN service, we selected a subset of 200 apps and analyzed them in several countries. Even though techniques exist for identifying whether an app is hiding behind a VPN (e.g., GPS coordinates, nearby WIFI access points), we empirically verified that this straightforward approach is effective for obtaining foreign ads. Overall, we analyzed 4,478 apps in our main experiment, and 200 apps for each VPN session in other countries. As such, our analysis includes advertisements from USA, Russia, India, the United Kingdom, Germany and Greece.

App installation and exercising. Our framework installs and analyzes each application individually. At installation time we approve all permissions that the apps may request, including run-time permissions, using the "`adb install -g`" option. Finally, using the UIHarvester module [35], our framework interacts with each application for five minutes using a breadth-first traversal strategy.

6.2 Intra-app Data Exfiltration

WebAPI Accesses. As can be seen in Table 2, in-app ads access a plethora of HTML5 WebAPIs, both mobile-specific and not, across all countries. We found several instances of in-app ads accessing motion sensors using the WebAPIs `addEventListener(devicemotion)` and the `addEventListener(deviceorientation)`, which return continuous values from the Accelerometer and Gyroscope respectively. We did not find any in-app advertisements accessing the camera, the microphone or the GPS of the device, even though many of the tested apps had these permissions in their Manifest file and were, thus, allowed to use them at run-time. Regarding the GPS sensor, in-app ads may use another non-intrusive way for roughly estimating the device's location, by utilizing the `getTimezoneOffset` function to infer the user's timezone.

We also observe several ads using the `navigator.getBattery` API, which provides information about the battery status and can be used to effectively track users across the web [69]. Moreover, we observe that in-app ads access functions that are known to be used for canvas fingerprinting, such as `HTMLCanvasElement.toDataURL`, `HTMLCanvasElement.getContext`, `createElement(canvas)` and `WebGLRenderingContext`. Finally, we find in-app ads reading, writing and deleting data from local storage using `getItem`, `setItem` and `removeItem` respectively. Even though we did not further investigate whether in-app ads access local storage for malicious activities, since it falls outside of the scope of this paper, we believe that such functions should be restricted since local storage can be used for re-identifying mobile devices [101].

Google's Interstitial Ad Placements. Google's library for interstitial ad placements allows ads to capture sensor data not only from the `View` displaying them but from others as well, thus increasing the attack surface of the intra-app data exfiltration attack. Our analysis shows that Google's interstitial ad placements can be found on average in 14.14% of the apps; Figure 3 shows the number of apps that contain Google's interstitial ad placements based on their numbers of downloads. We observe that interstitial ad placements are more prevalent across apps that have between 100K+ and 100M+ downloads. Apps with 5B+ downloads are rare and most of them either do not contain ads (e.g., WhatsApp, Messenger) or may use their own tools for interstitial ad placements (e.g., Facebook). We argue that Google's interstitial ad library currently presents a significant threat to users, as it allows ads to execute their JavaScript before they are displayed on the screen, affecting even apps that adhere to secure development practices and separate sensitive functionality and Views from ad-related content.

Table 3: Top 10 most popular apps with the `SYSTEM_ALERT_WINDOW` permission. Additional app permissions (CAM, MIC and GPS) allow in-app ads to silently capture photos, listen to conversations and retrieve the device's position even if the app is in the background.

⇓ DLs	Package Name	CAM	MIC	GPS
5B+	com.google.android.music	✗	✗	✗
5B+	com.facebook.katana	✓	✓	✓
1B+	com.lenovo.anyshare.gps	✓	✓	✓
1B+	com.twitter.android	✓	✓	✓
1B+	com.facebook.lite	✓	✓	✓
1B+	com.skype.raider	✓	✓	✓
500M+	com.imo.android.imoim	✓	✓	✓
500M+	jp.naver.line.android	✓	✓	✓
500M+	com.viber.voip	✓	✓	✓
500M+	com.mxtech.videoplayer.ad	✓	✗	✓

6.3 Inter-app Data Exfiltration

`SYSTEM_ALERT_WINDOW` permission. Apps that request this dangerous permission and are downloaded from Google Play may automatically obtain the permission without any user interaction or consent. This permission allows WebViews to be attached to the `WindowManager` and execute code that can access sensors in the background. To make matters worse, unaware users do not know that such background activities remain alive even if they perform a UI swipe to terminate the app. In our dataset 416 apps hold this permission and 291 out of them are marked by Google Play as "Contains Ads" (i.e., in-app ads). Table 3 shows the 10 most popular apps that contain ads and hold this permission.

Apart from motion sensors that do not require a permission, for each app we also include other dangerous permissions that provide access to additional sensors (e.g., CAM, MIC and GPS) and can be abused by in-app ads. We note that if one of these apps is installed on the device and a WebView displaying ads is configured to run in the background (due to intentional or accidental misconfiguration, by the developer or an integrated third-party ad library), all of the user's apps are vulnerable to the touch input inference attack. Based on our findings we argue that these apps should carefully review the security implications of obtaining this dangerous permission and whether it is really needed for their functionality; if it is indeed necessary, apps should explicitly inform users and ask for consent.

Motion Sensor Leaks. During our experiments with in-app advertisements, we found several cases where motion sensors were accessed and the values were leaked to third-party domains. Table 4 presents these results with applications tested multiple times over several months. Each app that we list may have displayed more than one in-app ad that accessed motion sensors (e.g., Vodafone ad) during a single execution. For each in-app ad that listens to `devicemotion` and `deviceorientation` events, since these APIs return continuous data, we also mark whether the corresponding app is vulnerable to the intra or the inter-app data exfiltration attack. For the former, an app is marked with a (◖) if it displays ads in sensitive `Views` (e.g., login), or with a (◗) if it uses Google's interstitial ad placements. If both are true they are marked with (●). In the inter-app data exfiltration attack, we mark all apps that hold the `SYSTEM_ALERT_WINDOW` permission and give the ability to in-app ads to run in the background, rendering any other app running on the device vulnerable. In more detail, this is possible if the WebView displaying the ad is attached to the WindowManager using the `WindowManager.addView()` and provides the `TYPE_APPLICATION_OVERLAY/TYPE_PHONE` layout parameter. Even though we statically analyzed these apps for instances of ad-related WebViews being attached to the WindowManager we didn't find any. Nonetheless, it is well-known that mobile ad fraud is on a constant rise (e.g., [20, 24, 55]) and since ad libraries are mostly responsible for ad fraud activities [55], it would not be surprising if ad libraries are found to abuse the `SYSTEM_ALERT_WINDOW` permission in the future. Finally, for each entry we list the ad placement's domain and the last column denotes whether we could identify any motion data leakage in the network traffic and the corresponding JavaScript.

We found that motion sensor values are leaked to DoubleVerify's domains. Interestingly, even though DoubleVerify's policies state that data is collected to help customers measure the performance of the advertisement [36], they do not provide a detailed explanation or analysis on sensor data collection. Furthermore, as the use of motion sensors in advertisements is gaining traction, we believe that more publishers will likely appear soon. For entries that are not marked with sensor data leakage our system automatically identified that the advertisement accessed the motion sensors but we were not able to identify such values in the network traffic. This is due to the fact that most of the analyzed JavaScript code was heavily obfuscated and performed some form of data transformation, and also used additional libraries downloaded from the network. We observe that in-app ads that access motion sensors are not limited to a specific country since in all of our VPN sessions we identified such cases. Moreover, in certain cases (e.g., com.genius.android) we found that apps display in-app ads with access to motion sensors independently of the origin country. The actual content of the in-app ads we analyzed varies and we found that the ads accessing sensors included, among others, Vodafone products, Disney+ promotions and online gambling services. We observe that in many cases, the apps displaying ads with access to motion sensors are vulnerable to at least one of our attack scenarios and, in certain cases, to both.

Browser Apps present an interesting category of apps that requires a tailored approach to their analysis due to inherent characteristics of their functionality, e.g., the ability for multi-tab browsing. As such, it is important to better understand whether they enforce some access control policy for in-app ads, which requires manual analysis in a controlled and targeted experiment. In general, our next experiment aims to identify whether in-app ads are allowed to access motion sensors and if they are displayed (or execute JavaScript) in webpages with sensitive content.

Out of the most popular browser apps that are marked by Google Play as "Contain Ads", we selected those that we found to display in-app ads after ten minutes of manual interaction. Table 5 lists the browser apps that we tested, their number of downloads, and additional dangerous permissions for sensors that they hold. In order to exclude website-ads from our analysis, for each browser we visited a website with sensitive content that we know a priori does not display advertisements (i.e., the Facebook login page) and checked for in-app ads that are displayed on the screen, and for network flows that originate from ad domains. To identify whether

Table 4: Non-browser apps with in-app ads that listen to `devicemotion` and `deviceorientation` events. *Intra Vuln* denotes that the app either displays ads in sensitive `Views` (◖) or uses Google's interstitial ad placements (◗). If both occur they are marked with (●). *Inter Vuln* denotes apps with the `SYSTEM_ALERT_WINDOW` permission.

⇓ DLs	Package Name	Motion Events	Orientation Events	Intra Vuln	Inter Vuln	Ad Placement	Sensor Leaks
USA							
10M+	com.bigduckgames.flowbridges	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20228.doubleverify
10M+	com.resultadosfutbol.mobile	✓	✗	◗	✗	pubads.g.doubleclick.net	-
5M+	com.genius.android	✓	✗	◖	✓	pubads.g.doubleclick.net	-
10K+	com.kdrapps.paokfcnet	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20512.doubleverify
Russia							
5M+	com.genius.android	✓	✗	◖	✓	pubads.g.doubleclick.net	tps20512.doubleverify.com
1M+	com.studioeleven.windfinder	✓	✓	✗	✗	pubads.g.doubleclick.net	-
India							
5M+	com.bingoringtones.birds	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20516.doubleverify
500K+	com.appscores.football	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20516.doubleverify
500K+	com.promiflash.androidapp	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20519.doubleverify
United Kingdom							
100M+	com.melodis.midomiMusicIdentifier.freemium	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20518.doubleverify
10M+	com.livescore	✗	✓	✗	✗	pubads.g.doubleclick.net	-
10M+	com.ilmeteo.android.ilmeteo	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20518.doubleverify
5M+	com.genius.android	✓	✗	◖	✓	pubads.g.doubleclick.net	tps20514.doubleverify
500K+	com.famousbirthdays	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20515.doubleverify
Germany							
10M+	com.resultadosfutbol.mobile	✓	✓	◗	✗	pubads.g.doubleclick.net	-
5M+	com.genius.android	✓	✗	◖	✓	pubads.g.doubleclick.net	tps20515.doubleverify
1M+	com.studioeleven.windfinder	✓	✗	✗	✗	googleads.g.doubleclick.net	tps20515.doubleverify
Greece							
10M+	com.ilmeteo.android.ilmeteo	✓	✗	◗	✗	pubads.g.doubleclick.net	tps20519.doubleverify
5M+	com.genius.android	✓	✗	◖	✓	pubads.g.doubleclick.net	tps20512.doubleverify
1M+	com.studioeleven.windfinder	✓	✗	✗	✗	googleads.g.doubleclick.net	tps20237.doubleverify
1M+	hurriyet.mobil.android	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20520.doubleverify
1M+	com.mynet.android.mynetapp	✓	✓	✗	✗	embed.dugout.com	-
1M+	com.finallevel.radiobox	✓	✗	✗	✗	googleads.g.doubleclick.net	tps20515.doubleverify
1M+	netroken.android.persistfree	✓	✗	✗	✓	pubads.g.doubleclick.net	tps20515.doubleverify
1M+	com.phototoolappzone.gallery2019	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20516.doubleverify
500K+	com.famousbirthdays	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20236.doubleverify
500K+	com.kupujemprodajem.android	✓	✗	✗	✗	pubads.g.doubleclick.net	tps20514.doubleverify
100K+	de.heise.android.heiseonlineapp	✓	✗	●	✗	googleads.g.doubleclick.net	tps20520.doubleverify

browser apps enforce any access control for what an in-app ad (and its WebView) can access, we injected JavaScript code that accesses motion sensors only in network flows originating from ad domains.

In Table 5 we list the results of this experiment. We found that none of these browsers enforce any access control for in-app ads that access motion sensors, and all of them allow in-app ads to capture sensor data. Even though most of the browsers we tested did not display ads while visiting Facebook's log in page, we found that in-app ads displayed in the Home tab (or in any other tab) of the browser continue to access sensors even if the user switches tabs. As such, *all* browsers indirectly allow in-app ads to access sensors while a sensitive `View` is displayed, even if there is no ad in the current tab. According to Google's general policies [7] for web ads, it is forbidden to place ads in login pages. While this is a security practice that should be followed by all ads, we find that this is not the case with mobile apps, as Puffin displayed an in-app advertising banner on Facebook's login page. In summary, we found that (i) all browsers allow access to motion sensors by in-app ads, (ii) all browsers allow in-app ads to capture sensor data while a sensitive `View` is displayed, (iii) two browsers use Google's interstitial ad placements and (iv) four browsers hold the `SYSTEM_ALERT_WINDOW` permission. As such, all tested browsers are vulnerable to either the intra or the inter-app data exfiltration scenario, or both.

7 INPUT INFERENCE

Many prior studies have demonstrated the feasibility of input inference attacks using sensor data. While our main focus is exploring the feasibility of using the ad ecosystem as a sensor-based-attack delivery system and the underlying flaws in Android, we also explore the actual input inference phase of our attack. To that end, we build an input inference classifier based on Axolotl [83]. Since Axolotl's `learn_location` classifier is intended for use with iPhone devices, we modified it to work with a Google Pixel 4 device by changing different settings (e.g., display resolution, ppi density, etc.). Furthermore, as our goal is to predict the label of each keystroke (i.e., which key was pressed) we have developed a component for mapping the predicted coordinates into key labels.

Table 5: Browsers marked by Google Play with in-app ads that listen to `devicemotion` and `deviceorientation` events. CAM, MIC and GPS application permissions allow in-app ads to access additional sensors.

⇓ DLs	Browser Package Names	Motion Events	Orientation Events	Intra Vuln	Inter Vuln	CAM	MIC	GPS	In-app ads displayed on FB's login page
500M+	com.opera.mini.native	✓	✓	◖	✓	✓	✗	✓	✗
100M+	com.opera.browser	✓	✓	◖	✓	✓	✓	✓	✗
50M+	com.cloudmosa.puffinFree	✓	✓	●	✗	✓	✓	✓	✓
10M+	fast.explorer.web.browser	✓	✓	●	✗	✗	✗	✓	✗
10M+	browser4g.fast.internetwebexplorer	✓	✓	◖	✓	✓	✗	✓	✗
10M+	com.apusapps.browser	✓	✓	◖	✓	✗	✗	✓	✗

We use Axolotl's deep neural network (DNN) model as our baseline and propose three additional DNN models. First, Axolotl's DNN model has multiple layers for progressively extracting higher-level features from the sequential inputs from the accelerometer and gyroscope sensors. To precisely predict the location of each keystroke, this model applies the linear activation function for each layer and mean squared error (MSE) loss [10] for gradient computation. This model predicts the coordinates of the point on the screen that the user pressed, which we then map to the corresponding key label. Next, we build two novel models that directly predict key labels based on the input data. Second, we build a DNN model that uses the Rectified Linear Unit (ReLU) [65] as the hidden layer activation function and softmax activation for the output layer. To compute the multi-class classification loss, we use the Categorical Cross-Entropy Loss to update model weights during training. Our third model uses Recurrent Neural Network (RNN) techniques that capture the relationship between recent keystroke information for prediction. However, vanilla RNNs can be affected by long-term sequential data, and Long Short Term Memory (LSTM) networks have been proposed for learning long-term dependencies [46]. As such we use a Gated Recurrent Unit (GRU), which is a special case of LSTM but with simpler structures (e.g., uses fewer parameters) [21], to build our prediction model. Compared to LSTM, GRU also works well on long-term sequential data but is more efficient. Moreover, we also use the Dropout technique [98] to make the model less prone to over-fitting and achieve better performance. Finally, we also develop a GRU-based model that predicts coordinates, similar to Axolotl's approach, instead of key labels.

Our input inference attack captures and uses motion sensor values from in-app ads. We created two datasets for training our classifiers using a similar setup. A mock app is used for loading a webpage that calls the HTML5 functions that access motion sensors and outputs sensors values to `logcat`. Additionally, apart from the accelerometer and gyroscope values, we log the coordinates (i.e., x,y) while touching the screen, which are then normalized between -1 and 1. A value of -2 is used to indicate that no touch occurred at that time. Using this setup we created two different typing datasets. One dataset contains samples created using two-handed typing, while the other contains samples created using one-handed typing. In both datasets keys were pressed randomly for one hour.

Our motivating attack example paper is inferring the credit card number and CCV being typed by the user. As such our models attempt to identify and label any key presses that correspond to a digit; all other key presses are labelled as "*other*". We present the results from our experimental evaluation in Table 6. In both typing scenarios, we evaluated our classifiers using different dataset sizes by sampling 5, 10, 20 and 30 minutes from the corresponding one hour dataset. In each experiment we used 2/3 of the dataset for training and 1/3 for testing. Our two models that directly predict key labels outperform Axolotl's baseline model (MLP-MSE) and our version of a coordinate-predicting model (GRU-MSE) across all experimental setups, with the GRU model that returns key-press labels exhibiting the highest accuracy in most datasets. As one might expect, two-handed typing is more consistent and stable, resulting in a more accurate inference by our system. We observe that the GRU model is accurate for two-handed typing even when trained with a small dataset (e.g., 5 minutes) and reaches 87.51% when trained with enough samples. Additionally, the ReLU and GRU models performance is comparable across datasets, while in a single case the ReLU model outperforms GRU.

Table 6: Inference accuracy of the classification models.

Typing	Duration	MLP-MSE	GRU-MSE	ReLU	GRU
two-handed	5 minutes	47.63%	62.87%	74.32%	**74.56%**
one-handed	5 minutes	37.73%	40.92%	**44.57%**	44.49%
two-handed	10 minutes	50.49%	70.53%	78.63%	**79.19%**
one-handed	10 minutes	39.04%	44.67%	50.07%	**50.10%**
two-handed	20 minutes	52.19%	79.23%	82.53%	**82.87%**
one-handed	20 minutes	39.76%	45.76%	52.51%	**54.11%**
two-handed	30 minutes	52.68%	81.70%	84.79%	**85.66%**
one-handed	30 minutes	40.17%	51.11%	55.64%	**56.67%**
two-handed	60 minutes	53.38%	85.25%	87.06%	**87.51%**
one-handed	60 minutes	40.57%	50.48%	59.70%	**59.99%**

The intent of this exploratory experiment is to demonstrate the feasibility of misusing in-app ads for conducting input inference attacks. While the two models we propose achieve high accuracy, and we will open-source our code to facilitate additional research, our goal is not to replicate the extensive experiments conducted by studies that focused on input inference. Importantly, findings from prior work further support the generalizability of our results and the practicality of our proposed attack. Specifically, prior work has shown that techniques for reconstructing users' touch input are effective even when tested against a variation of devices with different hardware characteristics, screen orientation, display dimensions or keyboard layouts [19]. In most studies [15, 19, 47, 59, 62] a diverse training dataset with multiple users was used, and experiments suggest that inferring PINs is actually more consistent and accurate when training and testing is done across multiple users and devices

rather than a single device or user [19]. Hodges et al. [47] demonstrated that even when using a very short training dataset (i.e., less than the size of a tweet) the accuracy of these techniques remains surprisingly high (they report 81% accuracy in bigram prediction). Similar findings were observed by Miluzzo et al. [62], further suggesting that a pre-trained classifier from a small number of people could be successfully used to infer other users' taps at a large scale.

8 DISCUSSION

Here we discuss various dimensions of the emerging threat of in-app ads accessing rich features of the operating system, and propose a set of guidelines for better protecting users.

Automatically Identifying Sensor Leaks. While our system is able to automatically identify WebAPIs that access mobile sensors by in-app ads, it is also important to identify whether motion sensor data is exfiltrated over the network. Several challenges exist for tracking sensor values from low-level system calls to the network layer. Prior work (e.g., [72, 85]) proposed mechanisms that identify device identifiers (e.g., MAC address, Advertising ID, etc.) being leaked over the network. These techniques can not be applied directly in our case because mobile sensors provide continuous values that change based on the device's position. While one could intercept the appropriate APIs so as to always return the same unique value, prior work has shown that apps (and by extension in-app ads) can hide suspicious activity when provided with a constant sensor value [71]. Another mechanism for identifying leaks in Android apps is AGRIGENTO [22], which is based on blackbox differential analysis and detects leaks by observing deviations in the resulting network traffic even in the presence of obfuscation. Unfortunately, this approach requires at least two executions and is thus inherently better suited for experiments that focus on app-specific behavior; due to the dynamic nature of the advertising ecosystem different in-app ads may be shown across executions of the same app.

While in our study we manually analyzed the JavaScript code and the network flows of in-app ads that access motion sensors, motivated by prior work we propose a more systematic methodology for identifying sensor leaks over the network. Specifically, we developed a tool for identifying sensor leaks (i) by tracking the raw sensor values provided by the motion sensors of the operating system and (ii) searching for specific keywords used for labelling sensor values in network traffic. To track sensor values, first, we manually identified which Android sensors are triggered when specific WebAPIs are called. For example, when the function `window.addEventListener("devicemotion",function(event))` is triggered, the event `rotationRate` maps to the `TYPE_GYROSCOPE` sensor, while the events `accelerationIncludingGravity` and `acceleration`, map to `TYPE_ACCELEROMETER` and `TYPE_LINEAR_ACCELERATION` sensors respectively. Next, we modified the SensorDisabler [107] module to return values (within the appropriate range for each sensor) from a list of predefined values. These steps ensure that the HTML5 WebAPIs responsible for accessing motion sensors always return legitimate predefined values which can be identified in network flows. Since these values can be leaked in an encoded form we also check for these values in common encoding formats (e.g., base64). We consider a large-scale measurement and evaluation of this tool in the wild as future work. We also note that our technique for identifying sensor data in network flows suffers from certain limitations; we can not handle cases where sensor values in network traffic have been encrypted or are heavily obfuscated.

Responsibilities, countermeasures and guidelines. Due to the severity of the attacks enabled by mobile sensors inside in-app advertisements, it is imperative to inform the advertising community and establish guidelines for access control policies. We strongly believe that users should be given the option to allow or deny access to any sensor information. Even though access control policies enforced using Android permissions exist for sensors such as GPS, Camera and Mic, we found that it is also crucial to guard with an Android permission motion sensors. Unfortunately, even if this policy is enforced by the OS, it only partially solves the problem since in-app ads exists in the same address space as the actual application's process, and share all of the application's privileges. As such, it is also a responsibility of the World Wide Web Consortium (W3C) to update the HTML5 policies for access to motion sensors by coupling them with the Permissions API. To bridge the gap between policies of the OS and the HTML5, Android can establish a general interface that allows users to distinguish access control to sensitive data and sensors between the native part of the app and WebViews dedicated for displaying advertisements (since WebViews that are part of the core functionality of the app may require access to these sensors). These complex policies, if they are to be introduced, require careful design and a strong collaboration between OS vendors and the W3C. Bellow we list a set of guidelines that users, developers and the ad ecosystem can follow as a temporary solution until a more generic policy is enforced.

Ad ecosystem. Advertising entities responsible for creating, selling and publishing ads must enforce stricter policies. They should not allow JavaScript in advertisements to access motion sensors unless there is a specific and well-documented reason to do so in the ad campaign contract. Furthermore, all ads must be dynamically analyzed in a sandboxed environment before publication, to eliminate cases of suspicious obfuscated behavior and data leakage. Ad-related entities that collect sensor data for their own purposes should provide a detailed explanation in their privacy policies.

Android access control and permissions. We argue that interstitial ads should *not* be allowed to execute JavaScript before they are displayed on the screen. Even though the main purpose of interstitials is to effectively load JavaScript and prepare the ad's content so it is ready for display at the desired time, it is challenging to enforce access control mechanisms for motion sensors at this layer. We believe that a possible solution for motion-based side channel attacks is to extend the functionality of the `FLAG_SECURE` option to also block access to motion sensors whenever a `View` with this option is in the foreground. The `FLAG_SECURE` option is already used by system apps when displaying `Views` with sensitive content, such as the billing information in the Play Store app and the Play Billing lib used for in-app purchases. Additionally, user applications (e.g., banking apps) already use this flag to prevent other apps from taking screenshots or reading the contents of the screen, which benefit from this solution. Additionally, apps that render web content (including in-app ads) should ask users' for their consent prior to accessing sensor information. Apps that do not require access to motion sensors for their core functionality must also inform users and ask for their consent, since it is possible

for embedded in-app ads to access these sensors. If users do not agree, WebViews with in-app ads should have limited functionality (e.g., `setJavaScriptEnabled(False)` and only display static ads.

Apps & Devs. Applications with in-app ads should never allow them to be displayed in sensitive forms (e.g., login). Moreover, browser apps should enforce navigational and cross tab isolation. In-app ads displayed while visiting a specific domain must not exist when visiting another domain. Additionally, the execution of JavaScript from in-app ads displayed in browser's Home screen must terminate when users open a new tab. Developers should thoroughly review the ad libraries they integrate in their apps. If in-app ads from the embedded ad libraries are responsible for sensor data collection then it is also their responsibility to inform users and ask for consent. Moreover, developers should not allow ad libs to include additional permissions without a detailed explanation.

Users. The `SYSTEM_ALERT_WINDOW` is a dangerous permission and users should carefully revise which of their installed apps have been granted access. Furthermore, we urge users to be cautious while operating apps in multi-window mode [31]. The multi-window mode (i.e., split screen) is used for displaying more than one app simultaneously and allows in-app ads to capture motion sensor values while the user is interacting with another app. Additionally, it is possible for in-app ads to access motion sensors even if the second application in the split screen mode is the Android Settings app, which processes sensitive data (e.g., account credentials).

Ethical Considerations. We carefully designed our experiments to minimize the effect of our experiments. Specifically, in our large-scale analysis experiments our framework did not click on ads to avoid incurring additional costs on advertisers. As such, the impact of our experiments is that of any measurement study that dynamically analyzes free Android apps, which commonly show in-app ads. Additionally, our IRB-exempted experiment with the ad campaign did not gather any information that can be used to identify or harm users in any way, and the only information made available in the report returned by the DSP was aggregate results about the ad's performance (e.g., apps displayed, impressions, clicks).

Disclosure. We submitted a detailed report with our findings to Google's Android security team and in their response they recognize the potential for abuse. They informed us that they are generally aware of attacks using motion sensors, and their plan to address them in an upcoming quarterly release. Furthermore, they informed us that they are investigating ways to provide app developers with tools that will help them fortify their apps against this sort of attack. Concerning the issues we described with (i) the `SYSTEM_ALERT_WINDOW` permission, (ii) the library for interstitial ads, and (iii) background WebViews not being terminated, the security team replied that they consider these to be functioning as intended. We disagree with this assessment and argue that these issues not only mislead app developers and users, but also create opportunities for attacks with severe implications. We hope that our work will draw additional focus from researchers and will, eventually, incentivize better access control and isolation enforcement.

9 LIMITATIONS AND FUTURE WORK

Our study on the collection of sensor data by in-app ads in the wild relies on our framework dynamically exercising apps. As with any dynamic analysis experiments with Android apps, our study presents certain limitations which we discuss bellow.

Element Coverage. Prior work [35] has explored how to improve UI element coverage when automatically exercising Android apps, and publicly released a tool that outperforms Android's Monkey. Their tool, Reaper, performs a breadth first traversal for identifying an app's visual and "interactable" elements. However, there are cases that exercising tools can not cover (e.g., playing a complex game). Another potential obstacle relates to apps that require the user to login prior to interacting with the app. While one could leverage Single Sign-On support , we opted against that as it might potentially influence the in-app ads delivered to our device.

Advertisement coverage and bias. Due to the inherently complex and dynamic nature of the ad ecosystem, coupled with the prevalence of personalized and micro-targeted advertisements, it is likely that our experiments reveal only a limited snapshot of the ad campaigns (mis)using motion sensors in the wild, and as such should be considered a lower bound. While providing a comprehensive measurement of the use of sensor data from in-app ads, we leverage a VPN service to diversify our device's geolocation and reduce the potential bias in our ad collection process. Nonetheless, we note that prior work has demonstrated how to detect that users are behind a VPN, which could allow ad libraries to infer our device's true location [73]. Additionally, persistent and hardware identifiers can be used to track users even when using a VPN. While we empirically found that using a VPN is sufficient for obtaining foreign ads, it is possible that certain apps or ads modified their behavior based on the use of VPN; in our analysis ads fetched over VPN sessions were less likely to collect sensor data. Overall, due to the ramifications of our attacks, and reports on the increase of sensor-based ads [74], we argue that there is dire need for stricter access control policies for mobile sensor data.

Network flows and JavaScript. Our study involves the analysis of network traffic and JavaScript code for potentially suspicious behavior and data leakage. In most cases, the network flows and JavaScript code were encrypted and obfuscated respectively, while dynamic code loading for fetching additional libraries further complicated the process. While we also manually examined these cases, it is possible that we missed additional cases of suspicious behavior. As such our findings should be considered a lower bound of the privacy risks posed by in-app ads that access motion sensors.

Interstitial ad libraries. Interstitial ads are very popular and many third-party libraries provide such functionality. In our study we focused on Google's library due to its popularity, and our analysis resulted in the identification of flaws that magnify the impact of our attacks. In practice, other third-party ad libs that offer interstitials may suffer from similar (or additional flaws).

Ad ecosystem practices. Based on our findings we believe that it is possible for anyone to abuse the mobile ad ecosystem for exfiltrating data by delivering an ad that captures the rich information provided by sensors. However, we note that different ad networks and DSPs may have different policies and constraints for the JavaScript code permitted in ads. Additionally, ad networks and DSPs may dynamically analyze submitted ads in a sandboxed environment before publishing them, to eliminate cases of malvertising. Given that the ability for ads to access sensor data is an emerging

trend for increasing user engagement [74] it seems unlikely that this will be prevented by many ad networks or DSPs.

Malvertising. Our study identifies an emerging threat that originates from popular apps downloaded from the official Google Play Store and advertisements fetched from major and legitimate services, as these affect even the most cautious users. We did not analyze malware or suspicious apps from third-party markets, and as such do not explore if ads fetched from less reputable or malicious ad networks are misusing sensor data.

10 RELATED WORK

To the best of our knowledge, this paper presents the first exploration of how the ad ecosystem can be misused for stealthy sensor-based attacks. Here, we briefly discuss pertinent prior work on in-app ads, the HTML5 WebAPI, and the risks posed by WebView.

In-app advertising. In-app ads are an essential part of the mobile ecosystem and the defacto source of revenue for app developers. This relationship introduces several privacy issues, as PII are accessed and leaked by embedded ad libraries [22, 35, 72, 84, 85]. Meng et al. [60] collected more than 200K real user profiles and found that mobile ads are personalized based on both users' demographic and interest profiles. They conclude that in-app ads can possibly leak sensitive information and ad networks' current protection mechanisms are insufficient. Reardon et al. [81] found that third-party SDKs and ad companies also use covert and side channels in order to obtain and leak permission protected data from apps that do not hold the appropriate permissions. Reyes et al. [87] performed an analysis of COPPA compliance and found that the majority of the apps and the embedded third-party SDKs contain potential COPPA violations. Nguyen et al. [67] performed a large scale study to understand the current state of the violation of GDPR's explicit consent and found that 34.3% of the apps sent personal data to advertisement providers without the user's explicit prior consent. Contrary to the popular belief that ad networks are responsible for user privacy, a recent study found that the privacy information presented from ad networks to developers complies with legal regulations and app developers are the responsible entity [102]. Another issue with in-app advertising is the potential for ad fraud from the apps or embedded advertising libraries. Interestingly, a recent study revealed that most ad fraud activities (e.g., triggering URL requests without user interaction) originate from ad libraries, with two libs also committing ad fraud by displaying ads in invisible WebViews that do not appear on the screen [54]. Several studies have also proposed solutions for preventing privacy leakage. Adsplit [94] allows the ad library to run in a separate process with different permissions, AdDroid [76] separates the privileged advertising functionality, and CompARTist [49] enforces privilege separation using compiler-based instrumentation. More restrictive solutions [4, 5, 72, 85, 95] have also been introduced that completely block advertising using network filtering or by employing VPNs.

HTML5 WebAPI. The standardized features of the WebAPI allow developers to create interactive elements and greatly improve the web experience, leading to higher user engagement [80]. However, these rich features can also be misused by privacy-invasive or malicious entities, such as web tracking and fingerprinting; the research community has extensively studied and presented such techniques [3, 37, 38, 68, 90]. For example, Eckersley et al. [37] explored browser fingerprinting in depth and introduced the Panopticlick project for identifying common fingerprinting features in web browsers. While traditional fingerprinting techniques [104] are used heavily to track desktop users, smartphone devices offer additional features for this purpose. Das et al. [25] presented a study on web scripts accessing mobile sensors in 100K websites. Apart from privacy-invasive tracking techniques, the rich features of mobile devices can also be used for augmenting security. Alaca et al. [9] explored device fingerprinting for enhancing web authentication, while Goethem et al. [105] proposed an accelerometer-based mechanism for multi-factor mobile authentication.

WebView. Numerous studies have showed that misconfigured hybrid apps pose a significant risk to users' privacy, and Luo et al. [56] identified several attacks against WebViews. The most notorious example is the `@JavascriptInterface` that allows JavaScript code to access Java methods. Rizzo et al. [89] evaluated the impact of such possible code injection attacks using static information flow analysis, while BridgeScope [112] assesses JavaScript interfaces based on a custom flow analysis. Additionally, Mutchler et al. [64] performed a large-scale analysis of more than a million mobile apps and identified that 28% contains at least one WebView vulnerability.

11 CONCLUSION

The unique hardware capabilities (i.e., sensors) of modern smartphones enable a series of features that allow for increased interaction with users, which can significantly improve their overall experience. Unfortunately, novel features also introduce new opportunities for misuse. In this paper we demonstrated a novel attack vector that misused the ad ecosystem for delivering sensor-based attacks. The key differentiating factor of our attack vector is that it magnifies the impact and scale of sensor-based attacks by allowing attackers to stealthily reach millions of devices without the need for a malicious app to be downloaded or users to be tricked into visiting a malicious page. To make matters worse, we have uncovered a series of flaws in Android's app isolation, life cycle management, and access control mechanisms that enhance our attacks' coverage, persistence and stealthiness. Subsequently, we created a realistic dynamic analysis framework consisting of actual smartphone devices for providing an in-depth view of mobile-sensor access, which allowed us to analyze a large number of popular apps and ads over a period of several months. Our findings reveal an emerging threat, as we were able to identify in-app advertisements accessing and leaking motion sensor values. Accordingly we propose a set of guidelines that should be adopted and standardized to better protect users. We hope that our study will contribute to the ongoing body of research pushing for better permission and access control management in Android by highlighting a previously-unexplored attack vector.

ACKNOWLEDGMENTS

This project has received funding from Horizon 2020 under grant agreements No 777855, 830927, 833683, and the National Science Foundation under contract CNS-1934597. This paper reflects only the views of the authors and the funding bodies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] 2018. Raccoon - APK downloader. https://bit.ly/1yIT4bR.
[2] 2020. JMango - Mobile Apps vs. Mobile Websites: User Preferences. https://jmango360.com/wiki-pages-trends/mobile-app-vs-mobile-website-statistics/.
[3] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1129–1140.
[4] AdAway. 2021. Ad-blocking for your Android. https://adaway.org/. Accessed: 2021-02-25.
[5] AdGuard. 2021. Surf the Web Ad-Free and Safely. Shield up! https://adguard.com/en/welcome.html. Accessed: 2021-02-25.
[6] AdMob. 2021. How much revenue can you earn from AdMob. https://admob.google.com/home/resources/how-much-revenue-can-you-earn-from-admob/. Accessed: 2021-02-25.
[7] AdSense. 2021. Ad implementation policies - Ad placement policies. https://support.google.com/adsense/answer/1346295. Accessed: 2021-02-23.
[8] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. 2015. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*. Internet Society.
[9] Furkan Alaca and Paul C van Oorschot. 2016. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 289–301.
[10] David M Allen. 1971. Mean square error of prediction as a criterion for selecting variables. *Technometrics* 13, 3 (1971), 469–475.
[11] Irene Amerini, Rudy Becarelli, Roberto Caldelli, Alessio Melani, and Moreno Niccolai. 2017. Smartphone fingerprinting combining features of on-board sensors. *IEEE Transactions on Information Forensics and Security* 12, 10 (2017), 2457–2466.
[12] Irene Amerini, Paolo Bestagini, Luca Bondi, Roberto Caldelli, Matteo Casini, and Stefano Tubaro. 2016. Robust smartphone fingerprint by mixing device sensors features for mobile strong authentication. *Electronic Imaging* 2016, 8 (2016), 1–8.
[13] S Abhishek Anand and Nitesh Saxena. 2018. Speechless: Analyzing the Threat to Speech Privacy from Smartphone Motion Sensors. In *2018 IEEE Symposium on Security and Privacy (SP). Vol. 00*. 116–133.
[14] Anssi Kostiainen, Alexander Shalamov. 2018. Accelerometer. https://www.w3.org/TR/accelerometer/. Accessed: 2018-07-13.
[15] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. 2012. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*.
[16] Ben Alman. 2018. Monkey-patch (hook) functions for debugging and stuff. https://github.com/cowboy/javascript-hooker. Accessed: 2018-04-23.
[17] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. 2014. Mobile device identification via sensor fingerprinting. *arXiv preprint arXiv:1408.1416* (2014).
[18] Liang Cai and Hao Chen. 2011. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. *HotSec* 11 (2011), 9–9.
[19] Liang Cai and Hao Chen. 2012. On the practicality of motion based keystroke inference attack. In *International Conference on Trust and Trustworthy Computing*. Springer, 273–290.
[20] Gong Chen, Wei Meng, and John Copeland. 2019. Revisiting mobile advertising threats with MAdLife. In *The World Wide Web Conference*. 207–217.
[21] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
[22] 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 *NDSS*.
[23] Cortesi, Aldo and Hils, Mayimilian and Kriechbaumer, Thomas. [n. d.]. mitmproxy. https://mitmproxy.org. v. 3.0.3.
[24] Jonathan Crussell, Ryan Stevens, and Hao Chen. 2014. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 123–134.
[25] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of ACM CCS, October 2018*.
[26] Anupam Das, Nikita Borisov, and Matthew Caesar. 2014. Do you hear what i hear?: Fingerprinting smart devices through embedded acoustic components. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 441–452.
[27] Anupam Das, Nikita Borisov, and Matthew Caesar. 2016. Tracking Mobile Web Users Through Motion Sensors: Attacks and Defenses.. In *NDSS*.
[28] Anupam Das, Nikita Borisov, and Edward Chou. 2018. Every Move You Make: Exploring Practical Issues in Smartphone Motion Sensor Fingerprinting and Countermeasures. *Proceedings on Privacy Enhancing Technologies* (2018).
[29] Erhan Davarci, Betul Soysal, Imran Erguler, Sabri Orhun Aydin, Onur Dincer, and Emin Anarim. 2017. Age group detection using smartphone motion sensors. In *Signal Processing Conference (EUSIPCO), 2017 25th European*. IEEE, 2201–2205.
[30] Android Developers. 2021. Interstitial (legacy API). https://developers.google.com/admob/android/interstitial. Accessed: 2021-02-25.
[31] Android Developers. 2021. Multi-Window Support. https://developer.android.com/guide/topics/ui/multi-window. Accessed: 2021-04-26.
[32] Android Developers. 2021. Permissions updates in Android 11. https://developer.android.com/about/versions/11/privacy/permissions. Accessed: 2021-04-26.
[33] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable.. In *NDSS'14*.
[34] Michalis Diamantaris, Francesco Marcantoni, Sotiris Ioannidis, and Jason Polakis. 2020. The Seven Deadly Sins of the HTML5 WebAPI: A Large-Scale Study on the Risks of Mobile Sensor-Based Attacks. *ACM Trans. Priv. Secur.* 23, 4, Article 19 (July 2020), 31 pages. https://doi.org/10.1145/3403947
[35] Michalis Diamantaris, Elias P. Papadopoulos, Evangelos P. Markatos, Sotiris Ioannidis, and Jason Polakis. 2019. REAPER: Real-time App Analysis for Augmenting the Android Permission System. In *9th ACM Conference on Data and Application Security and Privacy, CODASPY '19*. ACM.
[36] DoubleVerify. 2021. DOUBLEVERIFY PRIVACY NOTICES - SOLUTIONS PRIVACY NOTICE. https://doubleverify.com/privacy-notice/.
[37] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.
[38] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1388–1401.
[39] Tobias Fiebig, Jan Krissler, and Ronny Hänsch. 2014. Security Impact of High Resolution Smartphone Cameras.. In *WOOT*.
[40] Maximiliano Firtman. 2018. Mobile HTML5 Compatibility on Mobile Devices. http://mobilehtml5.org/. Accessed: 2018-04-22.
[41] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *2017 IEEE Symposium on Security and Privacy (SP)*. 1041–1057.
[42] Daniel Genkin, Mihir Pattani, Roei Schuster, and Eran Tromer. 2019. Synesthesia: Detecting Screen Content via Remote Acoustic Side Channels. In *2019 IEEE Symposium on Security and Privacy (SP)*.
[43] Google. 2020. WebView - A View that displays web pages. https://developer.android.com/reference/android/webkit/WebView. Accessed: 2020-10-28.
[44] Google. 2021. Provide advance notice to the Google Play App Review team. https://support.google.com/googleplay/android-developer/contact/adv_note. Accessed: 2021-04-26.
[45] Jun Han, Emmanuel Owusu, Le T Nguyen, Adrian Perrig, and Joy Zhang. 2012. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*. IEEE.
[46] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[47] Duncan Hodges and Oliver Buckley. 2018. Reconstructing what you said: Text Inference using Smartphone Motion. *IEEE Transactions on Mobile Computing* (2018).
[48] Jingyu Hua, Zhenyu Shen, and Sheng Zhong. 2017. We can track you if you take the metro: Tracking metro riders using accelerometers on smartphones. *IEEE Transactions on Information Forensics and Security* 12, 2 (2017), 286–297.
[49] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1037–1049.
[50] Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. 2015. On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms?. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 191–200.
[51] Hayden James. 2020. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/os. Accessed: 2020-10-11.
[52] Felix Juefei-Xu, Chandrasekhar Bhagavatula, Aaron Jaech, Unni Prasad, and Marios Savvides. 2012. Gait-id on the move: Pace independent human identification using cell phone accelerometer dynamics. In *Biometrics: Theory, Applications and Systems (BTAS), 2012 IEEE Fifth International Conference on*. IEEE, 8–15.
[53] Matthew Kaplan. 2021. 52 In-App Advertising Statistics You Should Know. https://www.inmobi.com/blog/2019/05/21/52-in-app-advertising-statistics-you-should-know. Accessed: 2021-02-25.
[54] Joongyum Kim, Jung-hwan Park, and Sooel Son. 2011. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In *NDSS*.
[55] Joongyum Kim, Jung-hwan Park, and Sooel Son. 2020. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In *28th Network & Distributed System Security Symposium (NDSS'21)*. 1–16.

[56] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 343–352.

[57] 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 *30th International World Wide Web Conference, WWW '19*.

[58] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. 2011. (sp) iPhone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*.

[59] Maryam Mehrnezhad, Ehsan Toreini, Siamak F Shahandashti, and Feng Hao. 2018. Stealing PINs via mobile sensors: actual risk versus user perception. *International Journal of Information Security* 17, 3 (2018), 291–313.

[60] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. 2016. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads.. In *NDSS*.

[61] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. 2014. Gyrophone: Recognizing Speech from Gyroscope Signals.. In *USENIX Security Symposium*. 1053–1067.

[62] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. 2012. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*.

[63] Mounir Lamouri, Marcos Cáceres. 2018. Screen orientation API. https://www.w3.org/TR/screen-orientation/. Accessed: 2018-07-13.

[64] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. 2015. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*.

[65] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Icml*.

[66] Sashank Narain, Triet D Vo-Huu, Kenneth Block, and Guevara Noubir. 2016. Inferring user routes and locations using zero-permission mobile sensors. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 397–413.

[67] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. 2021. Share First, Ask Later (or Never?)-Studying Violations of GDPR's Explicit Consent in Android Apps. In *USENIX Security Symposium*.

[68] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*. IEEE, 541–555.

[69] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2015. The leaking battery. In *Data Privacy Management, and Security Assurance*. Springer.

[70] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. 2012. ACCessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*.

[71] Pierluigi Paganini. 2019. Android apps use the motion sensor to evade detection and deliver Anubis malware. https://securityaffairs.co/wordpress/80037/malware/android-apps-motion-sensor.html.

[72] Elias P Papadopoulos, Michalis Diamantaris, Panagiotis Papadopoulos, Thanasis Petsas, Sotiris Ioannidis, and Evangelos P Markatos. 2017. The long-standing privacy debate: Mobile websites vs mobile apps. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee.

[73] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. 2018. Exclusive: How the (Synced) Cookie Monster Breached My Encrypted VPN Session *(EuroSec'18)*.

[74] BI INDIA PARTNER. 2020. Freecharge's innovative ad that used accelerometer and gyroscope motion sensors helped the brand reach 4.5 million users. https://www.businessinsider.in/advertising/ad-tech/article/freecharges-innovative-ad-that-used-accelerometer-and-gyroscope-motion-sensors-helped-the-brand-reach-4-5-million-users/articleshow/78384923.cms. Accessed: 2021-04-26.

[75] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (Seoul, Korea) *(ASIACCS '12)*. Association for Computing Machinery, New York, NY, USA, 2 pages. https://doi.org/10.1145/2414456.2414498

[76] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. 71–72.

[77] Dan Ping, Xin Sun, and Bing Mao. 2015. TextLogger: Inferring Longer Inputs on Touch Screen Using Motion Sensors. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '15)*.

[78] Andrea Possemato and Yanick Fratantonio. 2020. Towards {HTTPS} Everywhere on Android: We Are Not There Yet. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 343–360.

[79] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. 2011. iSpy: Automatic Reconstruction of Typed Input from Compromising Reflections. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. ACM, New York, NY, USA, 527–536. https://doi.org/10.1145/2046707.2046769

[80] Ashis Kumar Ratha, Shibani Sahu, and Priya Meher. 2018. HTML5 in Web Development: A New Approach. (2018).

[81] 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 *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 603–620.

[82] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. 2010. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks (TOSN)* 6, 2 (2010), 13.

[83] Tomas Reimers/Github. 2017. Axolotl Machine Learning Framework. https://github.com/tomasreimers/axolotl. Accessed: 2020-10-28.

[84] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. A longitudinal study of pii leaks across android app versions. In *Network and Distributed System Security Symposium*.

[85] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *MobiSys '16*.

[86] Yanzhi Ren, Yingying Chen, Mooi Choo Chuah, and Jie Yang. 2013. Smartphone based user verification leveraging gait recognition for mobile healthcare systems. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*. IEEE, 149–157.

[87] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, Serge Egelman, et al. 2018. "Won't somebody think of the children?" examining COPPA compliance at scale. In *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*.

[88] Rich Tibbett, Tim Volodine, Steve Block, Andrei Popescu. 2018. Device orientation event. https://www.w3.org/TR/orientation-event/. Accessed: 2018-07-13.

[89] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. 2018. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 25–46.

[90] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and defending against third-party tracking on the web. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 155–168.

[91] rovo89. 2018. Xposed framework. https://repo.xposed.info.

[92] Android SDK. 2021. SYSTEM_ALERT_WINDOW permission. https://developer.android.com/reference/android/Manifest.permission#SYSTEM_ALERT_WINDOW. Accessed: 2021-02-10.

[93] Android SDK. 2021. Window Layout - FLAG_SECURE. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE. Accessed: 2021-02-10.

[94] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. Adsplit: Separating smartphone advertising from applications. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 553–567.

[95] Anastasia Shuba, Anh Le, Minas Gjoka, Janus Varmarken, Simon Langhoff, and Athina Markopoulou. 2015. Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices. In *Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, & for the Students*.

[96] Prabhsimran Singh. 2020. Install Burpsuite's or any CA certificate to system store in Android 10 and 11. https://pswalia2u.medium.com/install-burpsuites-or-any-ca-certificate-to-system-store-in-android-10-and-11-38e508a5541a.

[97] Raphael Spreitzer. 2014. Pin skimming: Exploiting the ambient-light sensor in mobile devices. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 51–62.

[98] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[99] Terry Stancheva. 2021. 50+ App Revenue Statistics - Mobile Is Changing the Game. https://techjury.net/blog/app-revenue-statistics/#gref.

[100] statcounter. 2021. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2021-07-13.

[101] Keen Sung, JianYi Huang, Mark D Corner, and Brian N Levine. 2020. Re-identification of mobile devices using real-time bidding advertising networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–13.

[102] Mohammad Tahaei and Kami Vaniea. 2021. "Developers Are Responsible": What Ad Networks Tell Developers About Privacy. (2021).

[103] NEXD team. 2020. The Nexd Perspective: Gyroscope ads engage your audience. https://www.nexd.com/blog/using-gyroscope-ads-to-better-engage-your-audience/. Accessed: 2020-10-28.

[104] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. 2015. A classification of web browser fingerprinting techniques. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*. IEEE, 1–5.

[105] Tom Van Goethem, Wout Scheepers, Davy Preuveneers, and Wouter Joosen. 2016. Accelerometer-based device fingerprinting for multi-factor mobile authentication. In *International Symposium on Engineering Secure Software and Systems*. Springer, 106–121.

[106] Rick Waldron. 2019. Generic Sensor API. https://www.w3.org/TR/generic-sensor/. Accessed: 2021-02-10.
[107] Wardell Bagby. 2021. Sensor Disabler - This Xposed module allows you to modify and disable various sensors on your device. https://github.com/wardellbagby/Sensor-Disabler. Accessed: 2021-08-10.
[108] Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. [n. d.]. RouteDetector: Sensor-based positioning system that exploits spatio-temporal regularity of human mobility. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*.
[109] Robert Williams. 2020. Facebook's ad revenue rises 25% to record $20.7B. https://www.mobilemarketer.com/news/facebooks-ad-revenue-rises-25-to-record-207b/571362/. Accessed: 2020-10-28.
[110] xda developers. 2021. A Magic Mask to alter System Systemless-ly. https://www.xda-developers.com/how-to-install-magisk/.
[111] Zhi Xu, Kun Bai, and Sencun Zhu. 2012. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*.
[112] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. 2017. Precisely and scalably vetting javascript bridge in android hybrid apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 143–166.
[113] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. 2017. DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. 351–360.
[114] Jiexin Zhang, Alastair Beresford, and Ian Sheret. [n. d.]. Sensorid: Sensor calibration fingerprinting for smartphones. In *2019 IEEE Symposium on Security and Privacy (SP)*.
[115] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. 2014. Acoustic fingerprinting revisited: Generate stable device id stealthily with inaudible sound. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 429–440.
[116] John Zulueta, Andrea Piscitello, Mladen Rasic, Rebecca Easter, Pallavi Babu, Scott A Langenecker, Melvin McInnis, Olusola Ajilore, Peter C Nelson, Kelly Ryan, et al. 2018. Predicting Mood Disturbance Severity with Mobile Phone Keystroke Metadata: A BiAffect Digital Phenotyping Study. *Journal of medical Internet research* 20, 7 (2018).

A APPENDIX

A.1 Additional Technical Details

In our analysis we used Nexus 5X devices running Google's AOSP version 7.1.1 and the latest version of Chrome. Our framework installs and analyzes each application individually (e.g., install app, analyze, clear app data and uninstall app). Moreover, we limit any other background app activities using the adb toolkit to avoid contamination from other apps. This is a common technique when dynamically analyzing Android apps (e.g., [8, 72, 85]).

Network Interception. We intercepted network traffic by using Mitmproxy's transparent proxy option. Since apps by default do not trust the user trust store unless explicitly stated in the network security configuration of the app, we installed Mitmproxy's certificate into Android's system store. Doing so requires mounting the system partition as writable, adding Mitmproxy's certificate and updating the file's permissions. This approach requires that the Android device is rooted; for Android versions 10 and 11 altering the system partition and inserting the Mitmproxy's certificate in the system store requires Magisk [110]. As these techniques are common, due to space constraints, we refer the reader to appropriate online tutorials (e.g., [96]). Furthermore, WebView for Android 7 - 9 is built into Chrome and the latest version of Chrome no longer allows certificates whose validity is too long (e.g., NET::ERR_CERT_VALIDITY_TOO_LONG). As such, we changed the DEFAULT_EXP_DUMMY_CERT in Mitmproxy's `certs.py` file accordingly and recompiled Mitmproxy.

Certificate pinning: Even though apps' core functionality can implement certificate pinning to better protect network communication with their backend servers, we empirically found that our methodology for monitoring and intercepting WebViews' ad-related network traffic was effective as certificate pinning is inherently unsuitable for ad-network deployments. This is due to the complexity of the ad ecosystem and the various entities that take part during the process of rendering an advertisement, which make it difficult (if not impossible) to list all the domains that an embedded ad library should be able to reach (i.e., the list of domains is not known in advance). In fact, recent work [78] found that many embedded ad libraries tend to weaken the app's network security policies (e.g., asking developers to allow cleartext network communication).

HTML5 WebAPIs. Table 7 provides a complete list of the HTML5 WebAPIs monitored by our system. This list is based on the functions that access and retrieve mobile-specific sensor data through the official mobile HTML5 WebAPI [40], as well as prior work on mobile sensors attacks and device tracking (e.g., [25, 34, 69]).

A.2 Ad Campaign - Ethical Considerations

This straightforward exploratory experiment aimed to provide an initial indication of whether any countermeasures exist against ads accessing sensor measurements. While this experiment did not collect any user or device data, it is important to detail the ethical considerations behind our experimental design and set up. When framing our experiment within the guidelines and conceptual framework provided by the Menlo and Belmont reports, the main dimension that is pertinent[1] in our case is that of *beneficence*, which emphasizes that subjects should not be harmed and that any ethical research should strive to maximize the potential benefits while minimizing probable harms. During our design phase we assessed our experiment accordingly to ensure its ethical nature.

In more detail, our experiment involved an ad being delivered to users' devices. The harms that could potentially occur from such an experiment would stem from either the ad adversely affecting the user's device or the ad exfiltrating personal data or other data that could be used to identify the user (e.g., device identifiers like the Advertising ID). However, our experiment did not incur any such harm and our ad did not adversely affect the users' devices in any way or introduce any long-term implications. Our ad used the appropriate API calls to read sensor data, yet did not store or exfiltrate any of that data nor did it attempt to infer user inputs or actions. Moreover, as users come across numerous ads during their everyday browsing activities, we believe that the act of showing them an ad doesn't incur any harm or result in an experience that deviates from their normal browsing experience.

As such, our experiment did not pose any harm to users, while at the same time we believe that the potential benefits of our research are substantial, as we have identified a novel attack vector and a series of serious flaws that pose an important privacy threat to users. We hope that our research will result in more attention from the wider research and developer communities and will ultimately lead to changes in the underlying ecosystems and additional safeguards being deployed for protecting users.

[1] The guideline of *respect for persons*, which revolves around informed consent, is not applicable in this scenario. Regarding the guideline of *justice*, all users were essentially treated equally and no additional burden was incurred by specific users. Additionally, any benefits that result from this research will be equally distributed to all users.

Table 7: Full list of WebAPIs monitored by our framework.

WebAPI	Information
Mobile-specific	
Sensor APIs - Accelerometer	Provides acceleration applied to the device along all three axes.
Sensor APIs - Gyroscope	Provides the angular velocity of the device along all three axes.
Sensor APIs - AbsoluteOrientationSensor	Describes the device's physical orientation regarding Earth's reference coordinate system.
Sensor APIs - RelativeOrientationSensor	Describes the device's physical orientation without regard to the Earth's reference coordinate system.
window.addEventListener(devicemotion)	Fired at a regular interval, indicating the amount of physical force of acceleration the device is receiving.
window.addEventListener(deviceorientation)	Fired when new data is available about the current orientation (compared to the Earth's coordinate frame).
window.addEventListener(deviceorientationabsolute)	Event handler containing information about an absolute device orientation change.
window.addEventListener(deviceproximity)	Provides information about the distance of a nearby physical object.
window.addEventListener(userproximity)	Provides a rough approximation of the distance, expressed through a boolean.
window.addEventListener(devicelight)	Provides information from photo sensors or similar detectors about ambient light levels near the device.
window.addEventListener(orientationchange)	Fired when the orientation of the device has changed.
screenOrientation.addEventListener(change)	Event handler fired when the screen changes orientation.
screen.orientation.lock	Locks the orientation of the containing document to its default orientation.
screen.orientation.lockOrientation	Locks the screen into a specified orientation.
navigator.getBattery	Provides information about the system's battery.
navigator.vibrate	Pulses the vibration hardware on the device, if such hardware exists.
navigator.geolocation.watchPosition	Registers a handler function that will be called automatically each time the position of the device changes.
navigator.geolocation.getCurrentPosition	Get the current position of the device.
General	
XMLHttpRequest.send	The XMLHttpRequest method send() sends a request to the server.
XMLHttpRequest.response	The XMLHttpRequest response property returns the response's body content.
Date.prototype.getTimezoneOffset	Returns the time zone difference, in minutes, from current locale (host system settings) to UTC.
HTMLCanvasElement.toDataURL	Returns a URI containing a representation of the image in the format specified by the type parameter.
HTMLCanvasElement.getContext	Returns an object that provides methods and properties for drawing on the canvas.
WebGLRenderingContext	Interface to OpenGL ES 2.0 graphics rendering context for the drawing surface of a <canvas> element.
Storage.setItem	When passed a key name and value, will add (or update) that key to the given Storage object.
Storage.getItem	When passed a key name, will return that key's value, or null if the key does not exist.
Storage.removeItem	When passed a key name, will remove that key from the given Storage object if it exists.
Storage.key	When passed a number n, returns the name of the nth key in a given Storage object.
document.createElement(canvas)	The HTML5 <canvas> tag is used to draw graphics, on the fly, with JavaScript.
document.createElement(webgl)	A different context of <canvas> element.