

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCES AND ENGINEERING

Android's Security and Privacy Journey through the Lens of Access Control Policies

Michalis Diamantaris

PhD Dissertation

Presented

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

Heraklion, January 2023

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Android's Security and Privacy Journey through the Lens of Access Control Policies

PhD Dissertation Presented

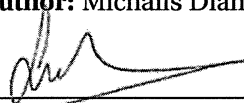
by **Michalis Diamantaris**

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

APPROVED BY:



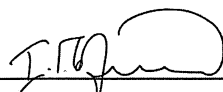
Author: Michalis Diamantaris



Supervisor: Sotiris Ioannidis, Associate Professor, Technical University of Crete

EVANGELOS MARKATOS Digitally signed by EVANGELOS MARKATOS
Date: 2023.02.01 13:49:54 +02'00'

Committee Member: Evangelos P. Markatos, Professor, University of Crete



Committee Member: Jason Polakis, Associate Professor, University of Illinois at Chicago

KONSTANTINOS MAGOUTIS
01.02.2023 20:19


Committee Member: Kostas Magoutis, Associate Professor, University of Crete

Polyvios Pratikakis Digitally signed by Polyvios Pratikakis
Date: 2023.02.07 14:36:04 +02'00'

Committee Member: Polyvios Pratikakis, Associate Professor, University of Crete



Committee Member: Michalis Polychronakis, Associate Professor, Stony Brook University



Committee Member: Alexandros Kapravelos, Associate Professor, North Carolina State University

Department Chairman: Argyros Antonis, Professor, University of Crete

Heraklion, January 2023

Στους γονείς μου

και

στη νονά μου

Σας ευχαριστώ για όλα

Acknowledgments

I would like to express my deepest gratitude to my advisor Professor Sotiris Ioannidis for his support and guidance during all these years, for always being there to discuss my ideas while providing directions and advises not only on my research but also on matters of personal interest. I would also like to express my deepest gratitude to Professor Evangelos Markatos for the fruitful discussions, the insightful advices and the valuable feedback.

I am particularly grateful to Professor Jason Polakis, for believing in me and helping me in *every* possible way since the very first day. Jason you are a very good friend and a great mentor. The experience of working with you and the skills learned are invaluable, and I am deeply grateful for that.

I would also like to thank, Kostas Drakonakis, Stefanos Fafalios and Panagiotis Ilias for all the brainstorming sessions (a.k.a drinking sessions) we had and for your emotional support when it was needed the most.

My best regards to all my co-workers and in the Distributed Computing Systems Lab, for their support and friendship, for the brainstorming sessions and the discussions we had, for the long hours we spent together in the lab during deadlines, and their contribution (in one way or the other) in the completion of the dissertation: Serafeim Moustakas, Elias P. Papadopoulos, Panagiotis Papadopoulos, Evangelos Ladakis, Dimitirs Deyannis, Giorgos Christou, Nick Christoulakis, Giorgos Tsirantonakis, Konstantinos Plelis, Eva Papadogiannaki, Eirini Degkleri, Kostas Solomos, Konstantinos Kleftogiorgos, Michalis Pachtalakis, Giorgos Vasiliadis, Antonis Krithinakis, Christos Papachristos, Stamatis Volanis, Thanasis Petsas, Aris Tzermias and anyone else I might be forgetting...

Finally, I would like to express my deepest and most sincere gratitude to my parents, my father Elias, my mother Rozita, my sister Marina and my brother Mario for their love and support throughout the years. I wouldn't have made it this far without you.

This work was partially supported by Institute of Computer Science, Foundation of Research and Technology Hellas and the European Union's Horizon 2020 Research and Innovation program, under grant agreements, No 700378 (CIPSEC), No 786890 (THREAT-ARREST), No 833683 (CyberSANE) and No 883275 (HEIR).

Abstract

The popularity of the Android operating system and the personalized nature of modern smartphones have gained a lot of attention. Smartphone devices offer a rich set of functionality that has been empowered by the application ecosystem. Android is dominated by free apps and developers earn their revenue by embedding advertisements. While this concept may appear beneficial to the user, as it does not induce a cost for enjoying a plethora of apps, it suffers from the inherent privacy risks of the embedded third-party libraries. As was foreseeable, applications are a black box with hidden inner workings and have become a treasure trove of sensitive user data and personally identifiable information.

In Android device resources are guarded by permissions and while Android has evolved over the last decade and moved towards a more fine-grained run time permission system, data privacy is still the major problem that mobile users face. Users can not differentiate between permission requests needed for the core functionality of the app and requests from third-parties, as they lack the contextual information that will enable them to make informed decisions. Additionally, mobile web browsing and apps' integration with web-based content, further aggravates the situation due to the semantic gap between access control policies in the operating system and the HTML5 WebAPIs.

In this dissertation using the permission management and enforcement system as our focal point, we explore how the Android operating system can be augmented to better protect users in real time. Specifically, we note that a fine-grained permission system should notify users of the origin of a permission request and explicitly state if it is needed by the app's core functionality or an integrated third-party library. We explore in depth the security and privacy issues that arise, due to improper access control, when mobile device characteristics are combined with the powerful features of the HTML5. Furthermore, we introduce a novel attack vector that misuses the advertising ecosystem and combines flaws in Android's isolation and permission management for delivering sophisticated and stealthy attacks that place even security-cautious users at risk. To mitigate these problems and better protect users, we implement solutions and propose a set of access control policies and guidelines.

Supervisor: Sotiris Ioannidis

Περίληψη

Η δημοτικότητα του Android και η προσωποποιημένη φύση των σύγχρονων smartphone έχουν ενσωματωθεί στην καθημερινότητα των ανθρώπων. Οι συγκεκριμένες συσκευές προσφέρουν ένα ευρύ φάσμα λειτουργιών, εμπλουτισμένο από μια πληθώρα εφαρμογών. Το εν λόγω οικοσύστημα κυριαρχείται από δωρεάν εφαρμογές και οι προγραμματιστές κερδίζουν έσοδά με την ενσωμάτωση διαφημίσεων. Ενώ μια τέτοια ιδέα μπορεί να παρουσιάζεται ως επωφελής για τον χρήστη, καθώς δεν συνεπάγεται κάποιο κόστος, ωστόσο, πάσχει από τους κινδύνους ασφάλειας και ιδιωτικότητας που σχετίζονται με τις ενσωματωμένες βιβλιοθήκες. Όπως ήταν προβλέψιμο, οι εφαρμογές είναι ένα μαύρο κουτί με κρυφές λειτουργίες και έχουν γίνει ένας θησαυρός ευαίσθητων δεδομένων και προσωπικών πληροφοριών.

Στο Android τα δεδομένα προστατεύονται από άδειες σχετικά με τη χρήση τους και, ενώ το λειτουργικό σύστημα έχει εξελιχθεί την τελευταία δεκαετία, εφαρμόζοντας ένα πιο λεπτομερές σύστημα πολιτικών ελέγχου πρόσβασης, η εξασφάλιση της ιδιωτικότητας εξακολουθεί να είναι το κύριο πρόβλημα. Οι χρήστες δεν μπορούν να διακρίνουν μεταξύ των αδειών που απαιτούνται για τις βασικές λειτουργίες μιας εφαρμογής και των αδειών που προκύπτουν ως αναγκαίες από τρίτα μέρη, καθώς δεν διαθέτουν τις σχετικές πληροφορίες που θα τους επιτρέψουν να λάβουν τεκμηριωμένες αποφάσεις. Επιπλέον, η περιήγηση στον ιστό μέσω των smartphones καθώς και η ενσωμάτωση περιεχομένου από τον ιστό στις εφαρμογές, επιδεινώνει περαιτέρω την κατάσταση, λόγω του χάσματος στη διαμόρφωση των πολιτικών ελέγχου πρόσβασης από πλευράς λειτουργικού συστήματος και από πλευράς HTML5.

Σε αυτήν τη διατριβή, εστιάζοντας στον τρόπο που χρησιμοποιείται το σύστημα διαχείρισης και επιβολής αδειών, εξετάζουμε πώς το λειτουργικό σύστημα Android μπορεί να επαυξηθεί για την καλύτερη προστασία της ιδιωτικότητας των χρηστών. Συγκεκριμένα, σημειώνουμε ότι ένα λεπτομερές σύστημα αδειών θα πρέπει να ειδοποιεί τους χρήστες για την προέλευση ενός αιτήματος άδειας και να δηλώνει ρητά εάν αυτή απαιτείται για τις βασικές λειτουργίες της εφαρμογής. Διερευνούμε σε βάθος τα ζητήματα ασφάλειας και απορρήτου που προκύπτουν, λόγω αδυναμιών στους ελέγχους πρόσβασης, όταν τα χαρακτηριστικά μιας φορητής συσκευής συνδυάζονται με τις δυνατότητες της HTML5. Επιπλέον, παρουσιάζουμε μία νέα δίοδο κατάχρησης του οικοσυστήματος των διαφημίσεων που, συνδυαστικά με τα ελαττώματα του Android, θέτει τελικά σε κίνδυνο ακόμη και τους χρήστες που είναι προσεκτικοί ως προς την ασφάλεια των δεδομένων τους. Με ζητούμενο την προστασία της ιδιω-

τικότητας, υλοποιούμε λύσεις και προτείνουμε ένα σύνολο οδηγιών και πολιτικών ελέγχου πρόσβασης.

Επόπτης: Σωτήρης Ιωαννίδης

Contents

Acknowledgments	vii
Abstract	iii
Περίληψη (Abstract in Greek)	v
Table of Contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Thesis Statement and Contributions	3
1.2 Organization of Dissertation	5
1.3 Referred Publications	6
1.4 Honors and Awards	7
2 Real-time App Analysis for Augmenting the Android Permission System	9
2.1 Background and Motivation	11
2.2 Reaper Design and Implementation	12
2.2.1 UI Harvester	12
2.2.2 Permission Harvester	15
2.2.3 Stack Analyzer	17
2.3 Dataset & Experimental Setup	18
2.4 Performance Evaluation	19
2.5 Permission Analysis	22
2.6 Discussion and Limitations	28
3 End Host Protection	31
3.1 One Flew Over the Tracker’s Nest	31
3.2 Implementation	31
3.3 Evaluation	33
3.3.1 Privacy performance	33
3.3.2 Latency overhead	33
3.3.3 Benefits from the use of antiTrackDroid	34
3.4 Discussion and Future Work	35
4 The Risks of Mobile Sensor-based Attacks	37
4.1 The Seven Deadly Sins of the HTML5 WebAPI	39
4.1.1 HTML5 WebAPI	39
4.1.2 Attack Taxonomy	40

4.2	Methodology and System Design	44
4.3	Data Collection and Analysis	46
4.3.1	In-Depth Analysis and Case Studies	49
4.3.2	WebAPI Request Origin	56
4.3.3	Transience of Web Measurements	58
4.3.4	Analysis Summary	59
4.4	Discussion	62
5	Misusing Mobile Sensors for Stealthy Data Exfiltration	67
5.1	Background	69
5.2	Motivation and Exploration	70
5.3	This Sneaky Piggy Went to the Android Ad Market	72
5.3.1	Threat Model	72
5.3.2	Intra & Inter-Application Attacks	73
5.4	System Design and Implementation	77
5.4.1	Additional Technical Details	79
5.5	Large Scale Measurement Study	80
5.5.1	Dataset & Experimental Setup	81
5.5.2	Intra-app Data Exfiltration	82
5.5.3	Inter-app Data Exfiltration	84
5.6	Input Inference	88
5.7	Discussion	90
5.8	Limitations and Future Work	93
6	Related Work	97
6.1	Android Permissions	97
6.1.1	Permission Analysis	98
6.2	Privacy Leakage	99
6.2.1	Third-party Libraries	99
6.2.2	Identifying Third-party Libraries	102
6.2.3	Privacy Leakage between Apps and Mobile Browsers	103
6.2.4	HTML5 WebAPI and Mobile Sensors	105
6.2.5	WebViews and Advertising	107
6.3	Protection Mechanisms	108
6.3.1	Ad-blockers and Network Monitoring	108
6.3.2	App and Third-party Library Compartmentalization	109
6.3.3	Fine-grained Access Control	109
6.3.4	Sensor Blocking	110
6.4	Dynamic Analysis and Exploration	110
7	Conclusion	113
7.1	Summary	113

7.2 Directions for Future Work	114
Bibliography	117

List of Figures

2.1	Overview of Reaper’s architecture.	13
2.2	Application Hook is PermissionHarvester’s <i>core hooking mechanism</i> that monitors PPCs and inspects stacktraces for extracting their origin. Android Server Hook is only used for validating the permission mappings.	15
2.3	Performance overhead of PermissionHarvester, including the overhead for the hook.	19
2.4	Performance overhead comparison between Reaper’s UIHarvester and UIAutomator.	20
2.5	Comparison of interactable object coverage between Reaper’s UIHarvester and Android’s Monkey.	21
2.6	Per app use of permissions and PPC by libs.	23
2.7	Origin of every permission protected function.	23
2.8	Breakdown for the 30 most used PPCs.	24
2.9	Breakdown for the 30 most used permissions.	24
2.10	Distinct libraries issuing PPCs in each app, and breakdown of PPCs for third-party and core functionality.	25
2.11	Number of permissions used across the apps that include the 4 most used libraries.	25
2.12	PII leakage from the most popular third-party libraries (sorted in descending order) broken down to the corresponding function call used. Blue circles denote PII being accessed through permission-protected calls, while the red circles indicate PII access by functions that are not permission-protected. The size of the circle denotes the number of apps in each case.	26
2.13	Coarse-grained (left) and Fine-grained (right) classification of PPCs and PII accesses initiated by libraries.	27
3.1	Defense mechanism overview.	32
3.2	Number of leaked ID without and with antiTrackDroid for the 30 apps with the higher number of ID leaks.	33
3.3	Overall request forwarding time with and without antiTrackDroid.	34
3.4	Percentage of KBytes saved.	35
3.5	Watts saved due to the less tracking request sent.	35

4.1	Taxonomy of attacks demonstrated in prior studies that leverage data from mobile sensors.	40
4.2	Overview of our crawling system’s architecture. Our system components provide an end-to-end view of requests to access the mobile sensors. The red arrows denote communication “pathways” observed by our system. . . .	44
4.3	Number of domains using mobile-specific WebAPI calls. Gray bars indicate domains that only use APIs that do not require a permission, while black bars indicate domains also accessing permission-protected APIs.	48
4.4	CDF of websites requesting access to sensors (left) and the number of attacks that are feasible with the data they collect (right).	48
4.5	Most frequent accessed sensors for each domain category.	51
4.6	Most frequent (feasible) attacks for each domain category.	51
4.7	Pearson’s coefficient between the Alexa rank for every website in every domain category, correlated to the number of sensors accessed and the number of attacks that are feasible with the data they collect.	52
4.8	Breakdown of sensors accessed and corresponding attacks that could be deployed by banking domains.	53
4.9	Breakdown of sensors accessed and corresponding attacks that could be deployed by domains with adult content.	54
4.10	Number of accessed sensors and feasible attacks for websites flagged as malicious (top). Websites that were flagged by at least three AV engines (bottom) – the number on the bars shows how many AV engines flagged the domain.	55
4.11	Aggregate number of sensors accessed by websites, classified by their country code top-level domain.	56
4.12	Aggregate number of feasible attacks for websites, classified by their country code top-level domain.	56
5.1	Overview of our attack vector. A malicious actor publishes an ad campaign that accesses mobile sensors, for delivering sophisticated and stealthy attacks.	73
5.2	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.	78
5.3	Number (top) and ratio (bottom) of apps with Google’s interstitial ad placements, per ranking bin.	83

List of Tables

2.1	Issues that resulted in certain apps not being traversed during our experiments.	22
2.2	Examples of inconsistencies and missing entries in the permissions-to-API-calls reported by previous work.	28
4.1	Sensor based side-channel attacks.	43
4.2	Number of domains using mobile WebAPI calls.	47
4.3	Breakdown of sensor based attacks, the number of domains capable of deploying them and the percentage of webpages capable of performing the specific attack.	49
4.4	Domain classification for the top 20 categories sorted in descending order based on the average request access for sensors across websites accessing at least one mobile sensor.	50
4.5	Third-party scripts accessing mobile-specific WebAPI calls.	58
4.6	Domains exhibiting differences in the WebAPI calls reported by our system and Das et al. [96].	59
4.7	Domain classification sorted in descending order based on the average request access for sensors across websites accessing at least one mobile sensor.	60
4.8	Access control currently enforced for <i>motion sensors</i> in popular mobile browsers. The "Universal revocation", "Per-site revocation" and "Origin differentiation" columns indicate whether the browser supports this feature. Columns marked with an asterisk (*) indicate whether the browser allows access to motion sensors in those scenarios.	65
5.1	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.	74
5.2	Full list of WebAPIs monitored by our framework.	80
5.3	Number of apps containing in-app ads accessing WebAPIs, analyzed across different countries.	81

5.4	Top 10 most popular apps with the <code>SYSTEM_ALERT_WINDOW</code> 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.	83
5.5	Non-browser apps with in-app ads that listen to <code>devicemotion</code> and <code>deviceorientation</code> events. <i>Intra Vuln</i> denotes that the app either displays ads in sensitive Views (◐) or uses Google's interstitial ad placements (◑). If both occur they are marked with (●). <i>Inter Vuln</i> denotes apps with the <code>SYSTEM_ALERT_WINDOW</code> permission.	85
5.6	Browsers marked by Google Play with in-app ads that listen to <code>devicemotion</code> and <code>deviceorientation</code> events. CAM, MIC and GPS application permissions allow in-app ads to access additional sensors.	88
5.7	Inference accuracy of the classification models.	90

Chapter 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 transformed how users experience and interact with web services, and the number of smartphone users has increased dramatically to 6.64 billion smartphone users [17]. Currently, Android is powering over 70% of devices worldwide [21] and has become the most prevalent mobile operating system. The popularity and the open source nature of the operating system has also attracted a lot of developers, many of which promote their content through the official Google Play market, that features one of the largest collections of applications, with over 2.6 million apps [42].

The Android app ecosystem is primarily driven by advertisements, and users can enjoy free applications, while developers earn their revenue by embedding advertisements. It is worth noting that mobile advertising has been proven more effective than Internet ads (e.g., up to 30 times [1]) and therefore mobile advertisements became the de facto source of revenue for app developers [49, 144]. Even major tech companies heavily rely on mobile advertising, with Facebook earning 94% of its ad revenue from mobile devices [274].

Over the last decade, Android has attracted much scrutiny from the security community, which has invested significant effort in better understanding and improving the Android ecosystem. At the heart of the Android OS lies the permission system, which is of critical importance, as it controls which Android API calls an app can issue. By extension, this system component decides which user data or device information and resources an app can access. The limitations of Android's permission system and potential modifications to it have been explored extensively [65, 157, 230, 251, 264], resulting in the permission system evolving significantly from its original design.

The risks that arise from the permission management system are further exacerbated by the dominating role that third-party libraries play in the Android app ecosystem. Libraries facilitate the app development process and even provide a steady revenue stream for developers through advertisements [175], without the need to charge users for the app itself. While this concept may appear beneficial to the users, as they are able to download

apps for free, it suffers from the inherent privacy risks of services being free, since the users end up "paying" with their personal data. The prevalence of third-party libraries has serious implications, as it can result in extensive leakage of personal information [24, 25, 30, 166, 213], since apps request more permissions than they need [247]. Indeed, applications often ask for permissions that reveal private information, including the user's location and persistent identifiers, such as the device's IMSI, that can accurately track users across the web while this information is *never* used for the core functionality of the app [108]. These permissions are mainly used by advertising and tracking libraries, with Google being the most prevalent one (74%), followed by Facebook (6%) [176]. According to Kaspersky [43], privacy leakage in mobile computing has long been one of the major security threats that users face. The personal nature of modern smartphones has rendered them a treasure trove of sensitive user data and personally identifiable information (PII), which is regularly collected and exfiltrated by Android apps. (e.g., [91, 157, 159, 200, 219, 229]). Unfortunately, most unsuspecting users desperately approve any permission requests, since many applications deny functionality unless these permissions are granted.

Furthermore, the mobile app personalization, when combined with powerful mobile HTML5 features and hardware components (e.g., mobile sensors), introduces significant security and privacy threats. For example, a plethora of prior studies have demonstrated that data obtained from mobile sensors can be used for identifying and tracking users across the web [55, 56, 80, 97–100, 106, 117, 134, 142, 143, 177, 187, 216, 220, 289, 291], inferring physical activities [134, 140, 174, 187, 216], and in more severe scenarios, the users' touch screen input [83, 138, 174, 198, 243, 280]. Typically these attacks assume that attackers are able to obtain sensor data through a malicious app installed on the device. However, in practice, modern browsers can mediate data exchange between websites and sensor data through the HTML5 WebAPI. This leads to a different threat model and an increased attack surface, as it removes the constraint of users having to install a malicious app; simply visiting a website can expose users to these attacks. To make matters worse, in the modern app ecosystem where most apps integrate third-party libraries for fetching and rendering third-party web content, attackers can simply use ads to stealthily reach millions of devices for delivering mobile sensor-based attacks.

There is a continuous effort by the research community to explore and mitigate threats in the area of mobile security and privacy, as well as educate users and make them aware of these privacy issues. Evidently, these efforts pushed Android towards designing and implementing more privacy-oriented policies and better access control mechanisms, showing considerable improvements over the flawed [145] original permission system. Even though Android has been moving toward a more fine-grained permission control system with each major revision, the problem is far from solved. Apps' core functionality still co-exists with third party library code and mitigations, such as the compartmentalization of libraries [231, 240, 285], can not be adopted as they do not offer the necessary granularity

for handling the intricacies in a way that would prevent third-parties from accessing user data without breaking the useful functionality that libraries offer to developers.

1.1 Thesis Statement and Contributions

Android made significant improvements over the years by integrating more security features and better access control policies. Despite of these improvements, serious design flaws in the underlying operating system, combined with novel features and the absence of sufficient access control, introduce new opportunities for misuse. Given the widespread presence of smartphones and the massive amount of sensitive information they store, it is imperative to stringently mediate access to user data. Currently, the proliferation and prevalence of third-party libraries render them a significant privacy risk, as users currently lack the required resources for deciding whether a specific permission request is actually intended for the app itself or is requested by possibly dangerous third-party libraries. Additionally, apps' integration with web-based content and the use of HTML5 APIs, combined with flaws in Android's isolation, life cycle management, and access control mechanisms, expand the attack surface and magnify the impact and coverage of mobile-based attacks.

This dissertation identifies several issues in the Android ecosystem that stem from the lack of understanding of the hidden mechanisms of the permission system. We identified that there is a dire need for better documentation of Androids' inner workings, as such scenarios lead to the confusion of both developers and users. By drawing on the successes of past efforts, while overcoming many of their limitations, we aim to enhance Android's access control mechanisms and better guide users into making informed decisions regarding their personal data. To this end, we also propose, discuss and implement solutions, enabling more fine-grained control of user data and argue that complex policies require careful design, and in certain cases a strong collaboration between Android and the World Wide Web Consortium. In short, in this dissertation we aim to show that:

The absence of sufficient access control mechanisms in Android, poorly-conceived OS design choices, and Android's inherent relation with novel features of the mobile web, introduce new security flaws and opportunities for misuse with severe ramifications that place even security-cautious users at risk.

Towards supporting the above thesis statement, in this dissertation we make the following contributions:

- Users cannot differentiate between permission requests needed for the core functionality of the app and requests from third-party libraries, and they can not make informed decisions regarding which permissions should be granted to each app. Motivated by this rationale, we argue that a fine-grained access control permission sys-

tem should notify users of the origin of a permission request and explicitly state if it is needed by the app's core functionality or an integrated third-party library. To bridge this significant gap we present Reaper, a system for dynamically analyzing apps and inferring the origin of permission-protected calls (PPCs) or non-permission-protected sensitive PII, through inline hooking that enables passive monitoring of the internals of the Android operating system. Our system can augment Android's run time permission system by enriching the contextual information provided to users. We experimentally evaluate our system and find that the overhead introduced is minimal, rendering it suitable for analyzing apps at a large-scale or integrating in user devices. We use Reaper to explore the interaction between libraries and Android's permission system in depth. We provide a fine-grained analysis of PPCs and PII access by third-party libraries in the wild. Our findings shed light on the alarming extent to which libraries dominate such calls, and motivate the need for incorporating origin information in permission requests. We publicly released our source code and the datasets used at <https://www.reaper.gr>.

- We design antiTrackDroid, a novel anti-tracking mechanism for mobile devices able to preserve the privacy of the users by blocking many personal and device information leaks to any third parties. Similar to state-of-the-art browser ad-blockers, our approach blocks any possible request that may deliver data to third parties, which can be used either for user profiling or device fingerprinting. We implement our system as an integrated filtering module for Android. Our solution uses a mobile-based blacklist and does not require changes in the respective OS or any kind of external infrastructure (i.e. proxy). We experimentally evaluate our prototype and show that it is able to reduce the leaking identifiers of apps by 27.41% on average, when it imposes an insignificant latency of less than 1 millisecond per request.
- Since modern browsers provide access to a device's underlying mobile sensors, a plethora of sensor-based attacks, previously limited to mobile apps, can "migrate" to the mobile web. For that reason, we conduct a large-scale, end-to-end study of websites, targeting mobile-specific sensors, across 183K of the most popular websites, according to Alexa, and our findings reveal the extent to which websites target smartphone users. We use a novel crawling infrastructure consisting of smartphone devices, which prevents potential evasive behavior from domains that can infer the presence of a virtualized execution environment (e.g., through canvas fingerprinting). We provide a taxonomy of previously reported sensor-based attacks and reframe them within the modern mobile ecosystem where WebAPI and WebView are widely supported and attacks are not constrained to users that install a malicious app. Guided by our taxonomy, we conduct a qualitative and quantitative analysis of our collected data and provide a comprehensive assessment of the risks presented by the mobile WebAPI. Due to the severity of the attacks enabled

by mobile sensors, we provide a set of guidelines for access control policies that should be adopted and standardized across browsers to better protect users and also allow them more control over their data. Our dataset is publicly available at <https://www.cs.uic.edu/~webapi> to further facilitate research on the security and privacy risks that users face due to mobile sensor data being accessible to websites.

- We introduce a novel attack vector that misuses 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 shared our code at <https://www.cs.uic.edu/~adsensors/>. We highlight that improper sensor access control combined with various system design flows creates an emerging threat that should not be taken lightly. 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.

1.2 Organization of Dissertation

This chapter introduced the problem of privacy loss due to the absence of sufficient access control in the Android operating system, having laid the ground behind the work presented in this dissertation, as well as summarized its key contributions. The rest of this dissertation is organized as follows:

Chapter 2 presents Reaper, a novel dynamic analysis system that traces the permissions requested by apps in real time and distinguishes those requested by the app's core functionality from those requested by third-party libraries linked with the app. We implement a sophisticated UI automator and conduct an extensive evaluation of our system's performance and find that Reaper introduces negligible overhead, rendering it suitable both for end users and for deployment as part of an official app vetting process.

Chapter 3 presents an end host protection that prevents privacy leaks from third-party trackers and enables users to access an online service through a mobile app or a mobile browser without risking their privacy.

Chapter 4 presents a comprehensive evaluation of the multifaceted threat that mobile web browsing poses to users, by conducting a large-scale study of mobile-specific HTML5 WebAPI calls across more than 183K of the most popular websites. We built a novel testing infrastructure consisting of actual smartphones, on top of a dynamic Android app analysis framework, allowing us to conduct an end-to-end exploration.

Chapter 5 introduces 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 in two distinct attack scenarios.

Chapter 6 presents the relevant work that is directly related to the approaches and mechanisms proposed in this dissertation.

Chapter 7 provides an overview of the contributions of this dissertation.

1.3 Referred Publications

Parts of the work for this dissertation have been published in international refereed conferences and journals.

- M. Diamantaris, S. Moustakas, L. Sun, S. Ioannidis and J. Polakis. 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)*, November 2021, Virtual Event, Republic of Korea.
- M. Diamantaris, F. Marcantoni, S. Ioannidis and J. Polakis. The Seven Deadly Sins of the HTML5 WebAPI: A Large-scale Study on the Risks of Mobile Sensor-based Attacks. In *ACM Transactions on Privacy and Security (TOPS)*, Vol. 23, No. 4, Article 19. *Publication date: June 2020.*
- F. Marcantoni, M. Diamantaris, S. Ioannidis and J. Polakis. A Large-scale Study on the Risks of the HTML5 WebAPI for Mobile Sensor-based Attacks. In *Proceedings of the 30th International World Wide Web Conference (WWW)*, May 2019, San Francisco, California, USA.
- M. Diamantaris, E. Papadopoulos, E. Markatos, S. Ioannidis and J. Polakis. REAPER: Real-time App Analysis for Augmenting the Android Permission System. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2019, Dallas, Texas, USA.
- E. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis and E. Markatos. The Long-Standing Privacy Debate: Mobile Websites Vs Mobile Apps. In *Proceedings of the 26th International World Wide Web Conference (WWW)*, April 2017, Perth, Australia.

1.4 Honors and Awards

- Reward from Google's Android security team, March 2021. *Remote inference of sensitive touch input using in-app ads that access motion sensors.*
- Best Paper Honorable Mention, WWW 2017. *The Long-Standing Privacy Debate: Mobile Websites Vs Mobile Apps.*

Chapter 2

Real-time App Analysis for Augmenting the Android Permission System

Modern smartphones have become a treasure trove of sensitive user data and personally identifiable information (PII) that is regularly collected and exfiltrated by Android applications (“apps”) [91, 157, 159, 200, 219, 229]. At the same time, the limitations of Android’s permission system have been explored extensively and various modifications have been proposed [78, 157, 230, 251, 264]. The privacy risks that arise from permission management are further exacerbated by the dominating role that third-party libraries have achieved in the Android app ecosystem by providing a revenue stream for developers [175]. On average, 41% of an app’s code is contributed by common libraries [160]. The prevalence of libraries has serious implications, as they incentivize apps to request more permissions than needed [247] and extensively leak personal information [24, 25, 30, 166, 213].

Android has been moving toward a more fine-grained permission control system with each major revision, showing considerable improvements over the original design where users were presented with confusing blocks of information at installation time [145]. Following the introduction of the new permission system in Android 6, users can now accept or reject a permission request at run time, or revoke permissions at any time through the system’s settings. However, recent work [112] demonstrated that users still do not fully grasp how permissions work and found that they are more likely to deny a permission request when given a detailed description of their personal information that will be accessed and uploaded (e.g., their actual phone number). Lin et al. [162] found that providing users with information on why a resource is being used can alleviate their privacy concerns, while in a different context Wang et al. [267] found that users perceive permissions differently when they are related to an application’s core functionality.

Even though the new approach empowers users by enabling a more precise granting of permissions, apps remain a black box with hidden inner workings, thus preventing users from fully benefiting from its potential; as users cannot differentiate between permission requests needed for the core functionality of the app and requests from third-party li-

10 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

libraries, they can not make informed decisions regarding which permissions should be granted to each app. Motivated by this rationale, we argue that a fine-grained access control permission system should notify users of the origin of a permission request and explicitly state if it is needed by the app's core functionality or an integrated third-party library.

To bridge this significant gap we present Reaper, a system for dynamically analyzing apps and inferring the origin of permission-protected calls (PPCs) through inline hooking that enables passive monitoring of the internals of the Android operating system. This requires tackling several challenges, each of which is addressed by one of the main components of our system. First, a dynamic analysis framework requires an efficient tool for traversing the graph of each app with sufficient coverage. We develop UIHarvester, an automation tool that utilizes hooks in the Android rendering process for identifying interactive elements and their properties, for traversing the app's graph without a priori knowledge of the app's functionality or visual characteristics. UIHarvester introduces negligible overhead that is 30-38 times smaller than that of Android's UI Automator, and improves coverage by $\sim 26\%$ compared to the tool that achieved the highest coverage in a comparative study [89].

PermissionHarvester is responsible for the main functionality of Reaper, as it hooks PPCs at run time and extracts the current stacktrace. Since the permissions required by functions are not the same across all Android versions, with newer versions not requiring permissions for certain calls that access PII, our tool automatically recognizes the OS version and adjusts its functionality accordingly. Even though PPCs protect device resources, common users do not have complete knowledge of Android's documentation and internals and are mainly concerned with apps accessing personally identifiable information. As such, PermissionHarvester also monitors PII access regardless of whether the call is protected by a permission or not. Extracted stacktraces are processed for identifying the origin of calls that are protected by permissions or access PII. Our approach is not affected by encryption techniques that may attempt to hide the presence of third-party libraries and the exfiltration of PII. Furthermore, Reaper does not require any modifications to the OS and does not depend on any sort of instrumentation, thus, introducing minimal performance overhead (we only require root access). Our system can be incorporated as part of the Android Open Source Project for enriching the contextual information shown to users.

To explore the potential benefits of our system, we use Reaper to analyze over 5K popular Android apps, and find several alarming results regarding the extent of third-party libraries' use of permissions and permission-protected calls. Indicatively, our study reveals that for 90% of the apps third-parties initiate more permission protected calls than the core app itself. We find that on average 65% of used permissions are needed by third-party libraries, and 34% of the apps never issue PPCs from their core code as the requested permissions originate solely from library code. To make matters worse, when it comes to *dangerous permissions* 48-59% of the requests originate from third-party libraries. For

permission-protected calls that reach PII, in 59% of the apps third-party libraries use the `getRunningAppProcesses()`, which can lead to precise user identification [48]. We also find many libraries accessing PII from non-permission protected calls. Finally, we explore how the origin information is augmented by accounting for the library type. We find that at least 37.3% of PPCs and 28.6% of PII accesses that originate from libraries are exclusively intended for functionality related to ads, tracking and analytics and could safely be denied by users without preventing the apps' intended functionality.

2.1 Background and Motivation

The incorporation of third-party libraries allows app developers to take advantage of useful existing functionality and also tap into an alternative revenue stream without the need to charge users for the app itself. While this may appear beneficial to end users as they are able to obtain apps seemingly for free, it suffers from the inherent privacy risks of the prevailing paradigm of services being free because users are the product [13] and “pay” with their personal data [175]. Not only have such libraries become prevalent (49% of apps contain at least one ad library [204]), but the risks they present [129] increase through time as they ask for increasingly more dangerous permissions [81]. This necessitates the deployment of functionality that can differentiate between permissions required by the actual app and those requested by third-party libraries. Tracing permission requests back to third-party libraries allows for enriching the contextual information presented to users.

While libraries can offer useful functionality to app developers, they may also surreptitiously add (potentially dangerous) permissions without the developer being explicitly informed. As such, users cannot rely on developers' intentions or safe practices for ensuring appropriate access to their data. Indeed, Android supports the merging of multiple manifest files, as each APK file can contain only one manifest file. While this functionality is meant to facilitate the inclusion of external libraries, it also allows third-party libraries to silently include permissions without the developer's approval. To make matters worse, developers have to explicitly and proactively include specific commands (i.e., `tools:node="remove"`) in their manifest to prevent libraries from including specific permissions.

To verify that this occurs in practice, we conduct an exploratory experiment with popular libraries. We create a test app and separately integrate each library; after compilation we extract the final manifest file to see which permissions have been included by the libraries without any form of notification. First, we look at one of the most prevalent libraries [14], namely Google Play Services. Google offers multiple libraries and we test Firebase and GMS (which incorporate functionality for analytics, ads etc.) and find that they add six and eight permissions respectively. We investigate whether libraries also merge dangerous permissions, and find that Instabug and Paypal silently include three

12 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

(read,write access to External Storage and Record Audio) and one (Camera) dangerous permissions respectively. This simple experiment highlights how Android offers functionality that can be misused by third-party libraries to silently obtain access to permissions that can affect the user's privacy or the device's normal operation. It is not meant to be exhaustive and many more libraries may be exhibiting such behavior in practice.

2.2 Reaper Design and Implementation

Reaper's primary goal is to distinguish which permissions are needed by the core functionality of the app and which by integrated third-party libraries. Many advertising and tracking companies freely provide preconfigured libraries and online tutorials on how to integrate them. Moreover, previous work [129] found that advertising libraries are prone to downloading code over HTTPS and dynamically loading and executing this code using the `DexClassLoader` class. Even though dynamic code loading offers useful functionality to developers, it can also be used to evade static analysis [208]. Furthermore apps may also hide their functionality through common obfuscation and encryption techniques [15, 91].

We leverage the hooking mechanism of Xposed [37] to build a dynamic analysis system that is designed to overcome the aforementioned obstacles. We require root access but do not rely on any OS modification, such as changing and recompiling the AOSP image, allowing us to apply it to any stock Android version. Figure 2.1 shows an overview of Reaper, which consists of three components: (i) `UIHarvester` (Section 2.2.1) a sophisticated UI automation tool for exercising apps, (ii) `PermissionHarvester` (Section 2.2.2) for hooking and monitoring the stacktrace of functions that lead to permission checks, and (iii) `StackAnalyzer` (Section 2.2.3) for analyzing stacktraces and inferring if they originate from a third-party library and of what type. If our system is adopted as part of an official app vetting process, or used by other researchers for dynamically analyzing apps, then all three components should be used, as shown in the gray box. If Reaper's functionality is integrated in the OS for augmenting the permission system, then only two of those components are required, as shown in the dotted line.

2.2.1 UI Harvester

A significant challenge when performing dynamic analysis on mobile apps is the traversal of the app's graph through the simulation of user interactions, without any a priori knowledge of the interactive content that will be displayed in the app. Previous work [54, 68, 85, 135, 168, 211, 212, 249, 255, 275, 288, 292], has explored the dynamic traversal of an application from different perspectives, such as achieving high traversing coverage or identifying malicious behavior. Unfortunately apart from requiring static analysis of the apk [54, 68, 135, 211, 249, 255, 275, 288], they may require some form of app instrumen-

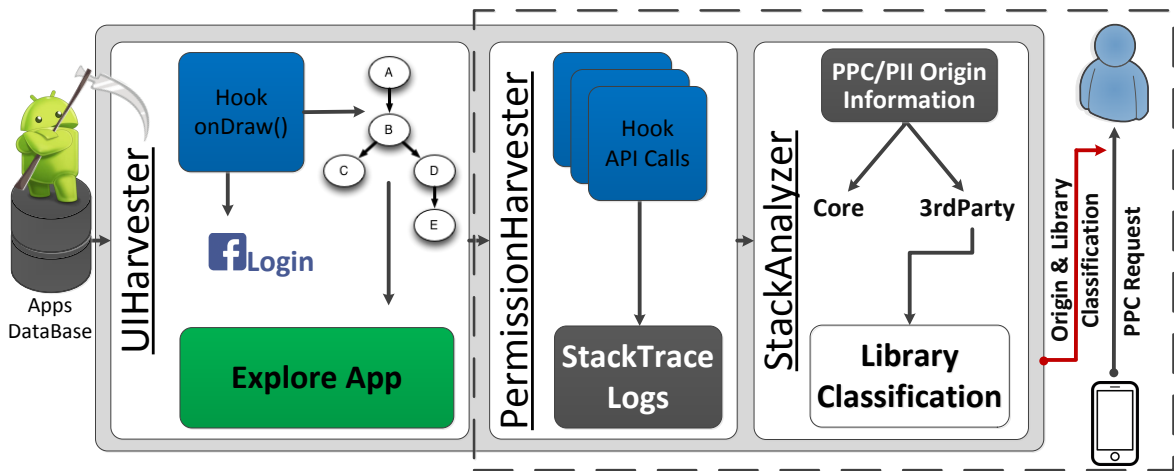


Figure 2.1: Overview of Reaper's architecture.

tation [68, 135, 211], or OS code modification [168, 212, 288], or are pinned to a specific Android version [54, 85, 288, 292].

UI Automator [31] is a useful tool available from the Android SDK that offers functionality similar to what we require; it can dump the interactable objects of the display and provide additional information about them. However, UI Automator presents two major disadvantages that render it unsuitable for our needs. First, if the app uses the `WindowManager.LayoutParams.FLAG_SECURE` option, UI Automator has to respect this specific flag and cannot output information about the objects being displayed. This flag is a security feature that treats the contents of the window as "secure" and prevents taking screenshots or being viewed on non secure displays [228]. This flag is not uncommon and is used to secure apps (e.g., PayPal) from side channel attacks, and can be used by apps or third-party libraries that want to evade dynamic analysis. Second, the performance overhead introduced is significant, rendering UI Automator unsuitable for a large scale analysis (details in Section 2.4).

To overcome the aforementioned restrictions, we developed a **plug & play** prototype that simulates user interactions, which fulfils the following design constraints:

- No requirement for a priori information on the content that will be displayed in the app.
- No requirement for decompilation or static analysis of the apk file, or access to the app's source code.
- No requirement for code modification (app or OS).
- No inefficient and ineffective random input generation approaches.
- Support for a wide range of Android versions.

Harvesting UI elements. Android renders the contents of the display based on a specific procedure, where each activity receiving focus provides the root node of the layout hierarchy and draws its layout. Drawing starts at the root node of the layout tree and is

14 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

traversed in a top-down order. Android also provides the `View` class, which is the basic building block for UI elements. While traversing the tree each `View` is rendered in the appropriate region. Rendering begins with a measure pass and continues with a layout pass. The former, is responsible for the dimension specifications of each `View`, while in the latter each parent positions all the children based on the measurements obtained during the measure pass. After this two-step process, the `onDraw()` function of the `Canvas` is called for rendering the contents of a `View` object. Whenever something changes on the display, `View` notifies the system and, depending on the changed properties, either the `invalidate()` or the `requestLayout()` functions are called. The former calls the `onDraw()` for the specified object while the latter instantiates the procedure from the beginning. Since `onDraw()` is called last before the actual rendering, we hook it to capture the displayed elements.

Identifying interactable objects. Having access to a `View` object enables us to use every method of the `View` class [7] from the Android SDK. This allows us to detect what type of elements are contained in the `View` object (e.g., `TextView`, `Button`, `Image`), as well as the corresponding metadata such as the text displayed, the resource-id, the index, and horizontal/vertical scrolling. To identify whether the elements of the `View` object are user interactable, we use the `getImportantForAccessibility()`, `hasWindowFocus()` and `isShown()` methods on each object. Moreover, we also find the position and the coordinates of each object by using the methods `getLocationInWindow()` and `getLocationOnScreen()`.

Traversing applications. `UIHarvester` hooks into the `onDraw()` function and exports all the information to `logcat`. Using a Python parser we extract this information and use all the interactable objects for performing a breadth-first traversal of the app. Since we know the type of each element as well as its coordinates, we utilize the Android Debug Bridge for performing the appropriate actions (e.g., tap, swipe, keyevents, etc.). Every time a new `View` is drawn on the display (e.g., after a button is pressed), `UIHarvester` exports its elements. By obtaining all the `Views` that have been drawn on the display, we can recreate the app's UI graph.

Login. Many apps provide a login option for a more personalized experience, while others require users to login before using the app. Not being able to handle such cases significantly limits the coverage and usefulness of any UI automator. As such, we implement an automated login feature that leverages Facebook's SSO.

Setting a threshold. Due to the dynamic nature and content of Android apps, it is possible for UI automators to get stuck in an infinite recursion of state transitions for certain apps. A simple example is the "back to menu" button. We handle this case by checking whether the elements (and their properties) have already been encountered in the exact same order. A case that can not be handled by our approach is when an activity renders content that is downloaded from the web and changes between transitions. As such, we need to impose a threshold for terminating the traversal of these apps. Since our goal is to execute as many permission-protected functions as possible, we set a rule to stop the

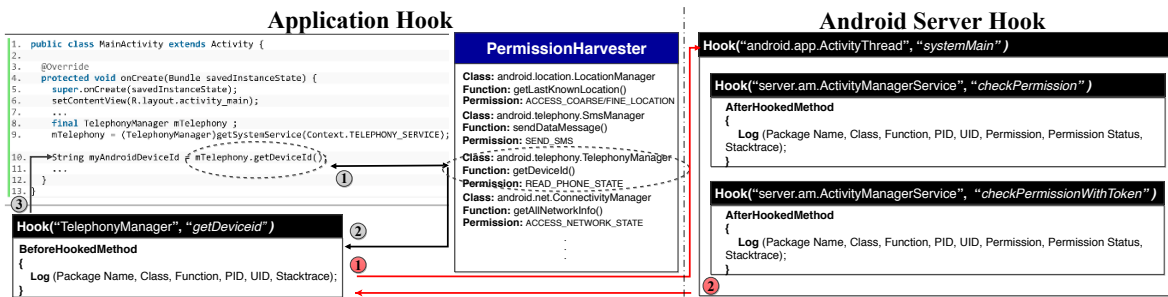


Figure 2.2: **Application Hook** is `PermissionHarvester`'s *core hooking mechanism* that monitors PPCs and inspects stacktraces for extracting their origin. **Android Server Hook** is only used for validating the permission mappings.

traversal when five minutes pass from when the last permission request occurred. While this could potentially result in a loss of certain requests, tracking and advertising libraries often perform their functionality either at launch time or after the user logs in [22, 189].

2.2.2 Permission Harvester

This is the core component of Reaper and is responsible for monitoring function calls and logging the current stacktrace for subsequent analysis. Since blindly hooking into every function call would result in an increase of the overhead without providing any additional information, we first need to identify the functions that lead to a permission request from the Android Server.

Permission mappings. The constant evolution of Android's permission system has led to many changes as well as compatibility and security issues regarding how each permission works [287]. Due to the differences between API versions, functions may lead to other permissions or may no longer be permission-protected across versions. For instance, the `getScanResults()` function from the `net.wifi.WifiManager` class only needs the `ACCESS_WIFI_STATE` permission up to API 22. Since API 23, this method also requires access to the device's location through `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. The Stowaway project [116] used static analysis in Android 2.2 to determine an app's API calls, and provided a map that identifies what permissions are needed for each API call. Recently PScout [66] and AXPLORER [71], statically analyzed the Android Framework, extended the mappings for newer versions and corrected previous uncertainties. When comparing the mappings of PScout and AXPLORER, we find various differences in their results; in API 22 AXPLORER registers the function `getWifiState()` in `net.wifi.WifiManager` with the `ACCESS_WIFI_STATE` permission. On the contrary, PScout registers the same function with the `DUMP` permission. As such, it is important to dynamically validate the permission mappings as we discuss below.

Mappings selection. Our system identifies the OS version and adjusts the permission

16 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

mappings, using AXPLORER's results for APIs 16 (Android v. 4.1) to 25 (v. 7.1) by hooking the appropriate functions. We excluded API 20 as AXPLORER does not provide mappings for this version. To facilitate our system's description we will refer to an example permission-protected call and the induced hooks as illustrated in Figure 2.2. By monitoring the PPCs ①, we can identify the corresponding permission through AXPLORER's list and the origin of the function call through the stacktrace of the current thread. The Java stacktrace holds every execution until a Binder transaction occurs, and also reveals the path and exact Java file (inside the apk) from which the call originated ②.

Validating permission mappings. To dynamically validate the mappings from previous work, we need to hook the appropriate functions of the Android Server. The `checkPermission()` and `checkPermissionWithToken()` functions (found in the class `ActivityManagerService`) grant or deny access to resources according to the app's permissions. Prior to API 22 access to these two functions is feasible by directly hooking them. Since API 22, a different entry point is needed for reaching them. To reach the methods and classes of the Android framework we hook the `systemMain()` function of the `app.ActivityThread` class. Within that hook we can monitor the permissions that each app and process request at run time by encapsulating the hooks for these functions.

Handling asynchronous calls. In Android different resources are handled by different System Services. For an app to access such information ①, a new thread of the appropriate service manager is created and this newly created thread calls the validation check mechanism ②. During this process, Android Binder is responsible for passing messages between entities, using the `onTransact()` and `execTransact()` functions. Even though the functions involved in permission validation are asynchronous calls, we know a priori the functions that will lead to a permission request ① and can call them sequentially and map each function call with the appropriate permission check that occurs on the Android Server ②. To this end we created a mock application that executes in a sequential manner all the permission-protected calls of the Android SDK.

In practice asynchronous callbacks are frequently used in Android, and a library can register its functionality, or part of it, as a callback. Even though the registered callback executes in a separate thread, the stacktrace of this newly created thread contains the origin of the embedded executed code. Since PermissionHarvester monitors the execution of PPCs independently of asynchronous calls, the StackAnalyzer component can identify the true origin of the PPC. We illustrate this process with an example of a library registering a PPC in an asynchronous callback. Listing 2.1 presents a callback that executes the `getLastKnownLocation()` function needed by the "com. appodeal" library. This library creates a subclass of an `AsyncTask` and overrides the method `doInBackground()`. Even though the code that is placed in this method is executed in a different thread, thus obscuring whether the core app or the library registered the callback, we can still successfully identify whether the executed code belongs to a third-party library.

```
0 java.lang.Thread.getStackTrace(Thread.java:580)'
1 android.location.LocationManager.getLastKnownLocation()'
2 com.appodeal.ads.an.e(SourceFile:243)'
3 com.appodeal.ads.d.b.<init>(SourceFile:180)'
4 com.appodeal.ads.d.i.a(SourceFile:295)'
5 com.appodeal.ads.d.i.a(SourceFile:105)'
6 com.appodeal.ads.d.i.doInBackground(SourceFile:37)'
7 android.os.AsyncTask$2.call(AsyncTask.java:292)'
...
13 java.lang.Thread.run(Thread.java:818)'
```

Listing 2.1: Example stacktrace for `getLastKnownLocation()`. The code that initiated the PPC through an asynchronous call belongs to "com.appodeal" package, which corresponds to the Appodeal third-party library.

Non-permission-protected PII leaks. It is important to note that not all PII is protected by permissions, and library developers may take extra measures to hide the presence of PII leaks and the surreptitious exfiltration of data (i.e., obfuscation, encryption and dynamic code loading). As PII can enable user tracking, it is crucial to identify the origin of such requests. Recent work [200] released an extensive list with such device characteristics that are leaked. We manually map those characteristics with their appropriate function calls and find that 8 such functions are not permission-protected. By extending Reaper to support these calls, our system is *able to identify the origin of PII leaks regardless of the call being protected by a permission or not*. While this is not part of our work's main focus, we include this information to further highlight the invasive behavior of third-party libraries in our study in Section 2.5.

Our approach can reveal privacy leakage without the need to perform deep packet inspection, thus, bypassing the obstacle of attempting to identify data exfiltrated in an obfuscated form, which has stifled previous work on PII leakage. For instance, in our experiments we found two apps¹ that integrate the XavirAd library, which downloads a dex file from a remote server, collects PII and sends them encrypted over the network [15].

2.2.3 Stack Analyzer

Apart from being useful for debugging, stacktraces can be used during run time execution since they contain essential information about the current thread. We opted for this approach as it provides a straightforward solution for identifying the origin of a function. The stacktrace contains a path to the source file and has four fields of interest: the package name, the class from which the method was called, the actual method and the file name of the source code. StackAnalyzer processes the stacktraces of important calls and checks if the package name of a known third-party library exists in the path of the stacktrace's function call. Even though library code can be obfuscated (e.g., classes, functions,

¹com.sevideo.slideshow.videoeditor, com.fourvideo.videoshow.videoslide

18 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

etc.), by default library package names remain intact since developers need to know which library to link in their app during the build process. To verify that stack inspection is effective in practice, we manually examined the package name of all the stacktraces collected from our experiments (see Section 2.3) and found that only 1.14% have an obfuscated package name, preventing us from identifying their origin. This is further corroborated by Wermkeet al. [272], who conducted an obfuscation detection analysis on 1.7 million apps from Google Play and found that even for obfuscated libraries larger scopes remain identifiable in package names (e.g., `com.google.ads.*`).

Third-party library package names. Li et al. [160] conducted a large scale analysis of 1.5 million apps from Google Play, in order to identify common Android libraries. Even though their approach does not handle obfuscated code, they identified 1,353 third-party libraries. LibScout [70] bypassed the limitations of obfuscated code by using a variant of Merkle trees and performing profile-matching between known third-party libraries and the contents of the apk file being tested. Since the results provided by LibScout are bound by the dataset they are trained with, it is possible for LibScout to miss some of the libraries that are integrated in the application. Indeed, during our experiments we came across such an example: AppsFlyer [8] a well known mobile tracking library. StackAnalyzer uses the combined results of both systems to create a coherent list of package names and to identify at runtime whether the stacktrace belongs to code originating from a third-party library.

Library classification. In practice, developers may use code from a third-party library that is integral to the app’s functionality. By classifying the type of the library from which the permission request originates, we obtain more detailed information regarding the nature of the call and whether it can be attributed to code that is necessary for the app’s intended functionality; e.g., by differentiating calls from an ad library to those from an app-development library. By disambiguating the origin of the calls Reaper further augments the contextual information presented to users and guides them towards granting “useful” permissions. Specifically, our system uses information from two sources [6, 160] to ascertain the category of the library from which each third-party call is initiated at runtime and provide that information to the user.

2.3 Dataset & Experimental Setup

We downloaded free apps from Google Play using Raccoon [27] and performed the majority of experiments using emulators. We opted for an emulator for the ability to deploy multiple virtual machines and analyze a large amount of apps. While third-party libraries may be able to infer the presence of a virtualized execution environment and alter their behavior, previous work on app analysis has also relied on emulators [241, 242]. To make the environment look more like an actual device we installed the Google Play services and

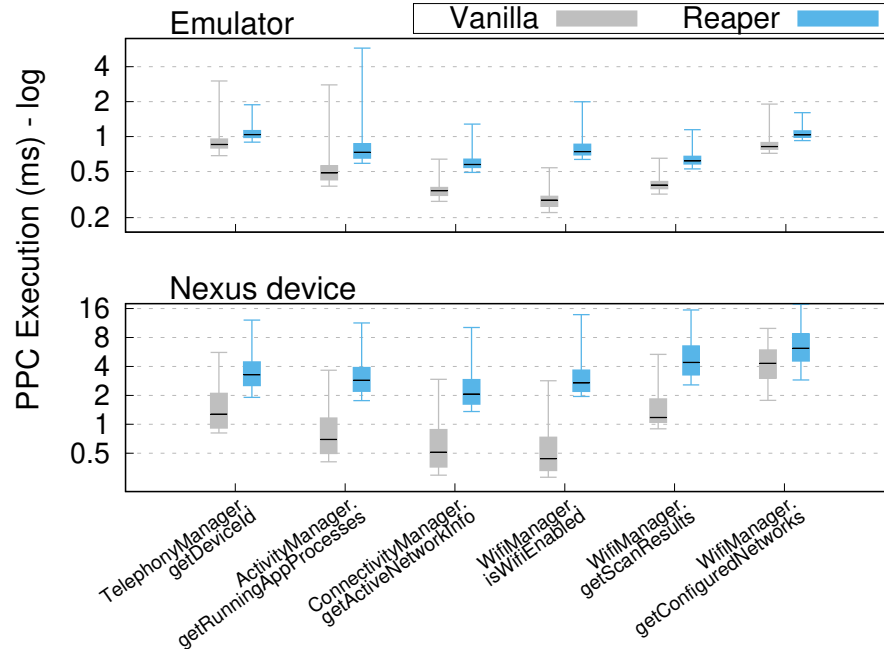


Figure 2.3: Performance overhead of PermissionHarvester, including the overhead for the hook.

signed in with a legitimate Google account. We conduct the experiments in Android API 22, as it is the API with the most accurate permission mappings available – AXPLORER’s mappings for API 23 are incomplete [10]. Overall, we selected the top 300 apps (or as many as were available) from each category, and downloaded a total of 5457 from 38 categories.

2.4 Performance Evaluation

Here we evaluate our system’s performance and measure the overhead introduced by each of the main components. We also compare UIHarvester to popular tools and demonstrate the advantages of our approach. We perform experiments using both an emulator and a Google Nexus 6 device, running the AOSP image with API 22.

PermissionHarvester overhead. The same code handles every PPC. Using a mock app that individually issues six PPCs from different managers, we measure the time needed for each PPC with and without PermissionHarvester present. In the vast majority of cases the function calls tested had an execution time of less than 1ms and 4ms for the emulator and the real device respectively. Since `System.currentTimeMillis()` does not produce readings of less than 1ms, we used the `System.nanoTime()` to extract a more accurate representation of the execution time. Figure 2.3 presents the results from 50K executions of the app. We observe that even though the same code applies to every hooked PPC, the induced

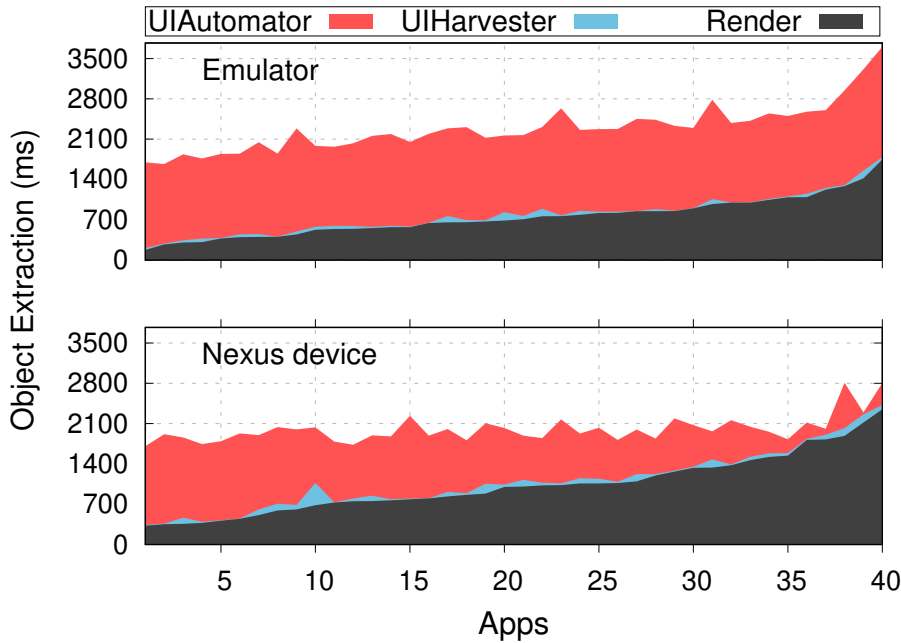


Figure 2.4: Performance overhead comparison between Reaper’s UIHarvester and UIAutomator.

penalty varies between 0.18-0.45ms for the emulator and 1.93-3.77ms for the Nexus device. The reason for this is that each PPC can result in stacktraces of different sizes. While `System.nanoTime()` is significantly more accurate than `System.currentTimeMillis()`, it is a relatively expensive call. It depends on the underlying architecture and can take up to 100 CPU cycles while measuring with millisecond precision takes only 5-6 CPU cycles. Apart from being more expensive, it also exhibits deviation in its execution time, which is reflected in the larger deviation of `getRunningProcess()` and `getDeviceID()`. Overall, the overhead for the actual hook is 0.0075ms [38] and the remaining overhead is due to the system call required for logging the stacktrace.

UIHarvester overhead. We measure the induced overhead of UIHarvester, by checking the extra time needed to render the contents of the display. We use the “Displayed” value from logcat, which represents the time elapsed between launching an activity and drawing its contents. For this experiment we selected 40 apps of different sizes and varying loading times, and measured the time needed to launch the main activity with and without UIHarvester.

As shown in Figure 2.4 the penalty in the emulator is between 0.3%-21% with an average of 6.55%, and depends on the number of elements drawn in the display. In the device the penalty is 0.16%-56.59% with an average penalty of 7.8%. In the worst case, the overhead to render the contents of a heavy View is 140ms for the emulator and 386.1ms for

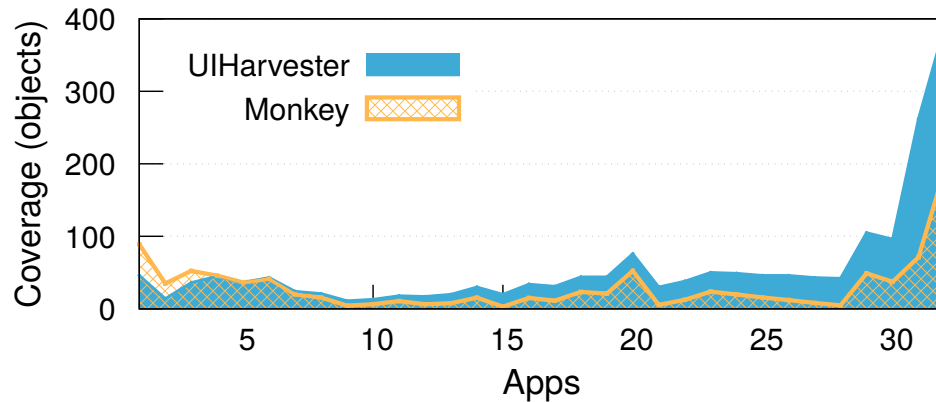


Figure 2.5: Comparison of interactable object coverage between Reaper’s UIHarvester and Android’s Monkey.

the device, which is acceptable for fully automated dynamic analysis. We also compare to the time needed to extract information about the display using UIAutomator, which offers similar functionality. On average UIHarvester only requires 40.19ms for the emulator and 67.29ms for the device. UIAutomator takes over 1,546ms and 2,001ms to extract the elements respectively, resulting in a 30 to 38-fold increase. Thus our tool offers superior performance while being more effective for this study.

UIHarvester Coverage. While GUI exploration and coverage in Android can be measured using different techniques (line coverage, activities, crashes, etc.) we opted for counting the interactive elements since it can be applied to both open and closed-source apps without the need for instrumentation. Choudhary et al. [89] evaluated six input generation techniques and compared them to Android’s Monkey. They found that the Monkey fuzzer was the best option, achieving a 40% coverage. To evaluate UIHarvester’s coverage, we obtained the same set of apps from [89] and compared the interactive objects found by UIHarvester and Monkey; we tested the 32 apps that remain functional. Since Monkey can only perform random clicks and does not count the interactive elements, we used the technique employed in UIHarvester to extract them. For a direct comparison, we set a timeout of 5 minutes and configured the time required to generate input events to be consistent between both tools. In Figure 2.5 we plot the number of objects found by Monkey (averaged over three runs) and the objects found by UIHarvester. Overall, UIHarvester improves coverage by 25.98%.

Compatibility between versions. A common limitation of analysis tools is being pinned to a specific Android version. By designing Reaper to have minimum dependencies, we maintain compatibility across APIs. We verified this by analyzing ten backwards-compatible apps on the four most common Android versions [5] and found that all Reaper components remained fully functional.

22 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

Table 2.1: Issues that resulted in certain apps not being traversed during our experiments.

Apps Without Interaction - <i>Addon (41), Launcher (31), Plugin (9), Theme (5), Widget (3)</i>	89
Manual Login Required	45
Installation Failure	37
Device Specific	15
Emulator Detection	1
Root Detection	1
Malfunction / Crash	156
Total	344

2.5 Permission Analysis

We study 5457 apps in order to understand the use of PPC and access of PII by third-party libraries in practice. Our study dynamically examines the origin of such calls, enabling a fine-grained exploration of the corresponding privacy risks.

Apps without PPC. In our experiments 315 apps did not issue a PPC call during their analysis. Furthermore, we were not able to traverse an additional 344 apps and obtain a PPC stacktrace. Table 2.1 breaks down the numbers for the issues that resulted in this; 89 apps could not be traversed due to their type, as they do not contain launchable activities and there is no direct interaction. Out of the remaining, 45 required a manual login, 37 failed during installation, 15 apps were for a specific device brand or only available for certain CPUs/GPUs and 2 apps did not execute because of the device’s environment. Also, 156 apps malfunctioned at launch time. To understand whether Reaper affects these 156 apps, we tested them without our framework, and observed that in both cases the apps remained non-functional. When executed without the Xposed framework, 155 apps continued to crash. While one app appears to be detecting Xposed, this can be trivially bypassed by renaming the Xposed package. Thus, practically, our experimental environment only prevented one app from running. To analyze apps that perform emulation or root detection, Reaper can also be used with a real device and the root requirement can be hidden using known root-hiding techniques [18]. Interestingly, certain apps that can not be traversed due to their type still perform PPCs (at launch time).

Third-party library use. In Figure 2.6 we explore the use of PPCs and their corresponding permissions by libraries. We observe that for 521 apps PPCs are only used by the app’s core functionality, while for 1,642 apps every PPC originates from third-party libraries. While there is varying behavior in the remaining 2,635 apps, there is significant use of PPC throughout. Overall, 65.22% of the permissions requested are not from the apps’ core code, but are requested by the libraries. These results verify our intuition that PPCs and their underlying permissions are heavily used by third-party libraries, with 34%

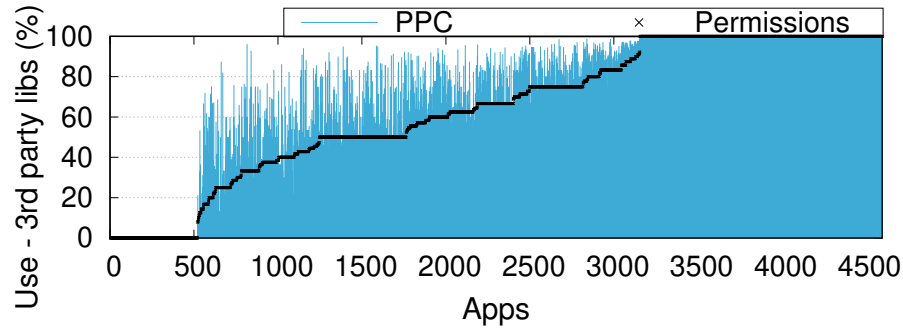


Figure 2.6: Per app use of permissions and PPC by libs.

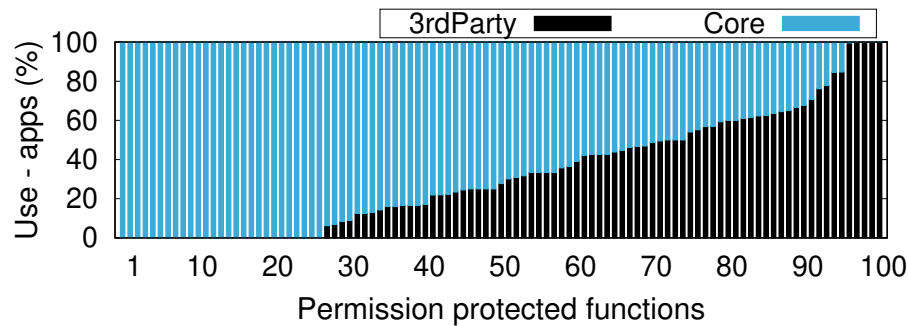


Figure 2.7: Origin of every permission protected function.

of these apps never calling them from their core code. This highlights the benefit of adopting the functionality offered by Reaper for informing users about the origin of permission requests and enabling more fine-grained control.

Function and permission origin. To better understand the origin of each function, we explore their use across all apps. As shown in Figure 2.7, use of permission-protected functions by libraries remains high and certain functions *are never used by core functionality*. For instance, one such function that also accesses PII, is `getSubscriberId()` which returns the device’s IMSI.

In Figure 2.8 we plot the 30 most used functions. We find that these typically are calls that return device specific information, such as Device-IDs, Network-Info, SSIDs, Location, Apps-installed, which are considered PII and are used by advertising and tracking companies. We observe that the `getSubscriberId()` function which is also included in the top 30, requires the `READ_PHONE_STATE` permission which is one of the permissions considered dangerous by the Android developer guide. In Figure 2.9, we plot the 30 most used permissions. We manually mapped these permissions to their protection level from the official Android source code [26] and found that third-party libraries also use permissions that fall in different protection levels such as signature, privileged, installer, development and dangerous. The use of the four dangerous permissions (i.e., `ACCESS_COARSE_LO-`

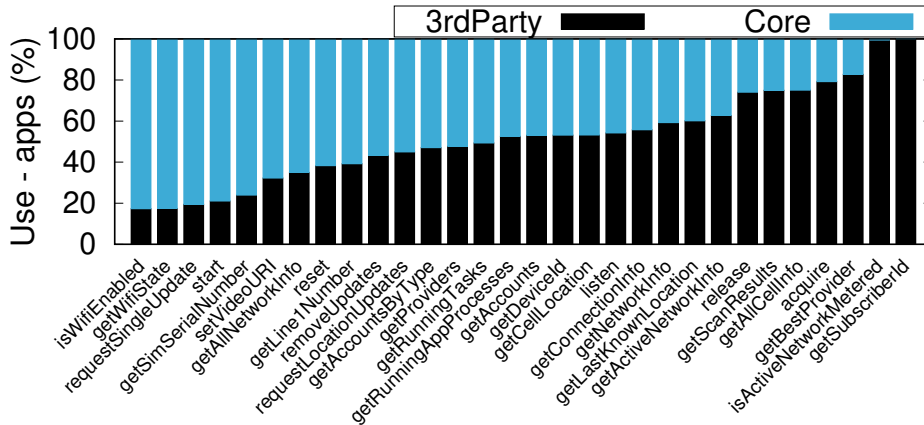


Figure 2.8: Breakdown for the 30 most used PPCs.

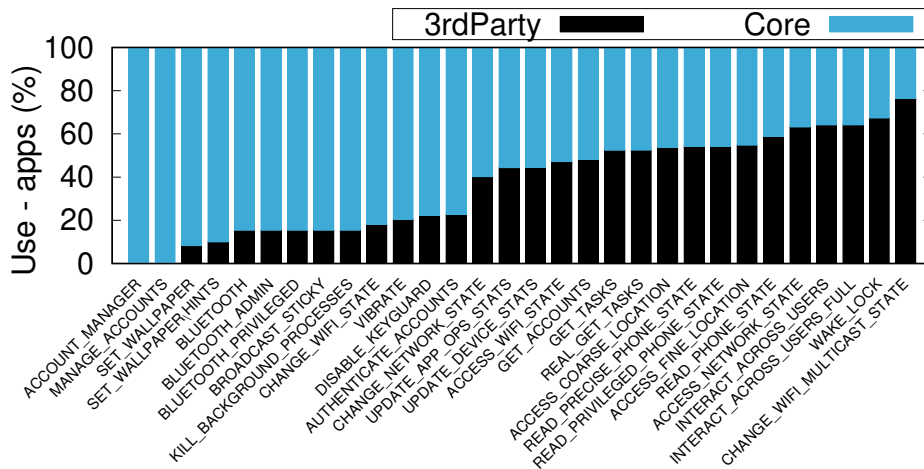


Figure 2.9: Breakdown for the 30 most used permissions.

CATION, ACCESS_FINE_LOCATION, GET_PHONE_STATE, READ_ACCOUNTS) ranges from 48% to 59% for third-party libraries. This means that for these apps *when users are presented with a dangerous permission request at run time, roughly half the time the permission does not originate from core code.*

Third-party library integration. To understand how many third-party libraries are used inside apps, we calculate the fraction of distinct third-party libraries using PPCs, as well as the total fraction of PPCs attributed to 3rd parties or core functionality. In Figure 2.10 (left) we observe that 30% of our dataset contains at least two distinct third-party libraries that initiate PPCs. Moreover, as can be seen in Figure 2.10 (right), for 90% of the apps third parties initiate more PPCs than the app’s core code.

Permission variation. We found cases where a library uses a different number of permissions across apps. In Figure 2.11, we select four of the most used libraries and plot the

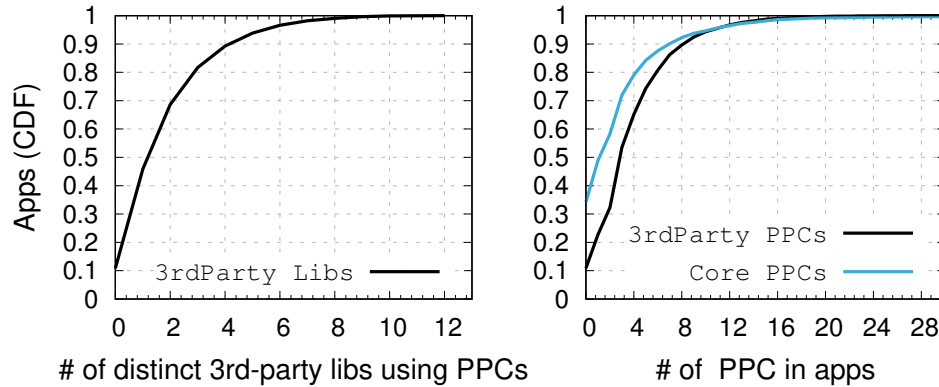


Figure 2.10: Distinct libraries issuing PPCs in each app, and breakdown of PPCs for third-party and core functionality.

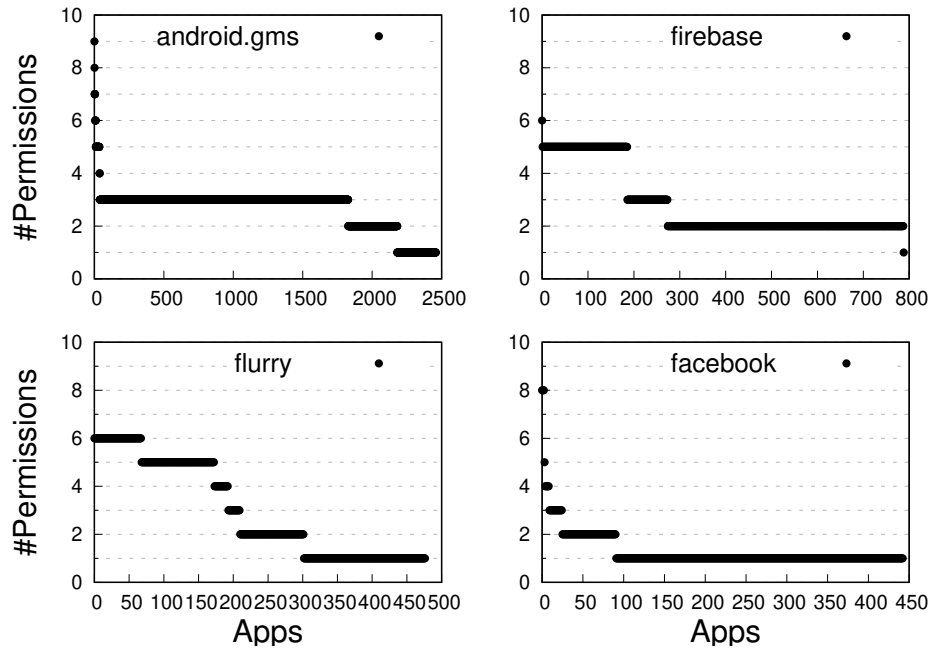


Figure 2.11: Number of permissions used across the apps that include the 4 most used libraries.

number of permissions used across all apps. One possible reason for this could be because UIHarvester was not able to reach a level of coverage that would trigger all the permission requests. However this will not always be the case since apps contain different versions of the same library, which may offer different functionality. Furthermore, libraries may also adjust according to the number of permissions granted [9].

PII access. While not our main focus, an important aspect of our analysis is exploring the extent of third-party libraries accessing PII; the origin information provided by

26 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

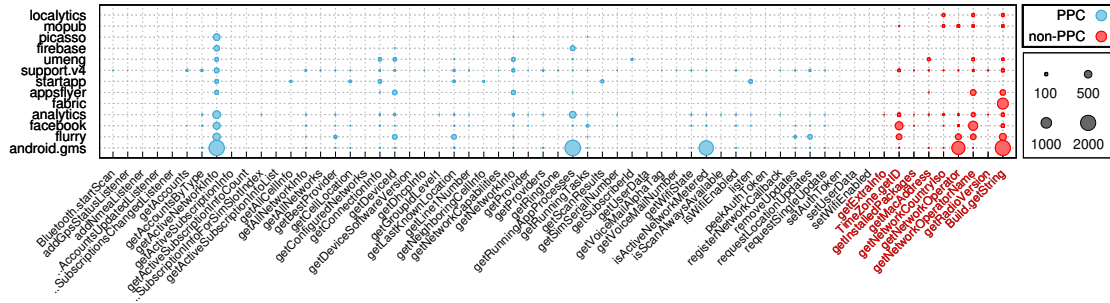


Figure 2.12: PII leakage from the most popular third-party libraries (sorted in descending order) broken down to the corresponding function call used. Blue circles denote PII being accessed through permission-protected calls, while the red circles indicate PII access by functions that are not permission-protected. The size of the circle denotes the number of apps in each case.

Reaper results in a more fine-grained and precise view of PII leakage when compared to prior studies that explore apps’ behaviors as a whole. We map function calls to PII based on the identifiers provided by prior work [200, 219] and the Android SDK documentation, and analyze the information provided by Reaper. Figure 2.12 shows all the functions that access PII, whether through a permission-protected call (blue circles) or not (red circles). The size of the circle denotes the number of apps that contain the respective library and issue the corresponding function call. We find that third-party libraries access the non-protected calls more frequently than the permission-protected calls. As users can be fingerprinted from the information returned by these functions, it is troubling that Android does not enforce a permission requirement. *We argue that all calls that lead to PII should be permission-protected, allowing users to manage what information can be accessed by apps and third parties.* The `getRunningAppProcesses()` function can also be called without the `GET_TASKS` permission. It returns a list of running processes and is being used by 3,218 (59%) apps. This is also a significant privacy threat, as previous work has shown that tracking companies can potentially identify users when as few as 4 apps are known [48]. Despite being discontinued for APIs ≥ 23 , it remains active for older versions, with 66.4% of devices running APIs ≥ 23 [5].

Library classification. To further enrich the origin information, Reaper classifies the type of libraries initiating each monitored call, which allows our system to further disambiguate the origin of calls. In Figure 2.13 (left) we present the coarse-grained classification of the type of library from which each call originated. Libraries that have multiple labels are counted in all respective categories. For a subset of the libraries we also obtain fine-grained information regarding the functionality that they offer, which we present in Figure 2.13 (right). While we provide a coarse-grained classification of all the calls initiated by libraries in our dataset, we are not able to obtain fine-grained labels for all the

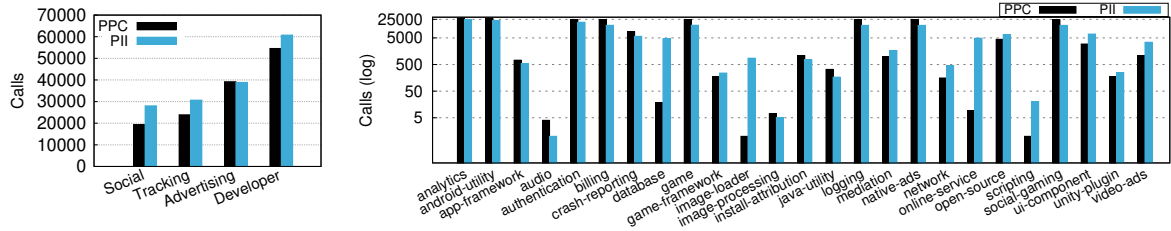


Figure 2.13: Coarse-grained (left) and Fine-grained (right) classification of PPCs and PII accesses initiated by libraries.

libraries identified in our experiments. Specifically, we obtain labels for 84 out of the 234 libraries that issue PPC calls, and for 75 of the 203 third-party libraries that access PII. As can be seen in Figure 2.13 (right), analytics-based libraries are responsible for the most PPC calls and PII accesses, while ad-related libraries issue more calls when the different subcategories are aggregated. It is also evident that specific libraries that ease the app development process are very common.

Using the coarse labels, we find that 15,610 PPCs and 21,322 PII accesses originate from libraries that are *exclusively* labelled as developer libraries, indicating that they are needed for the app’s core functionality and should likely be granted. On the other hand, 1,287 PPCs and 845 PII accesses originate from tracking or ad libraries, and can be safely denied. Furthermore, for libraries with multiple coarse-grained labels, we leverage the fine-grained labels and find that an additional 9,129 PPCs and 5,240 PII calls can be safely denied as they are used exclusively for analytics and advertising. Three of the most used libraries (facebook, google.gms and firebase) cannot be excluded using fine-grained labels as they cover a wide spectrum of functionality and contain numerous labels; however, for all three we can infer which aspect of their functionality is used in each call as the respective package name (e.g., `com.google.android.gms.ads`) explicitly denotes it (obviously, this approach cannot be applied to untrusted or unknown libraries). As such, the stacktraces allow us to identify an additional 10,424 PPCs and 11,580 PII calls than can be denied as they are used for analytics, ads, and tracking.

Overall, out of the 55,859 distinct PPC calls Reaper would enable users to safely deny 20,840 (37.3%) PPCs without preventing apps from leveraging third-party code for core functionality. Similarly, out of 61,602 PII accesses users could safely deny 17,665 (28.6%). Thus, apart from augmenting the permission system by providing rich contextual information, Reaper can further help users by providing concrete recommendations to accept or deny a considerable number of “straightforward” permission requests. The information provided by our system can also be used to expand access control tools like XPrivacy, allowing for more fine-grained control of user data, as it can selectively block invasive calls originating from third-parties while allowing such calls required for core functionality; we

28 Chapter 2. Real-time App Analysis for Augmenting the Android Permission System

Table 2.2: Examples of inconsistencies and missing entries in the permissions-to-API-calls reported by previous work.

	TSM.divideMessage()	TGSM.divideMessage()	CCW.sendStickyBroadcast()	CMCW.sendStickyBroadcast()
PSCOUT	DUMP	DUMP	DUMP	no permission required
AXPLORER	READ_PHONE_STATE	READ_PHONE_STATE	no permission required	no permission required
Reaper	no permission required	no permission required	BROADCAST_STICKY	BROADCAST_STICKY

TSM: telephony.SmsManager, TGSM: telephony.gsm.SmsManager, CCW: content.ContextWrapper, CMCW: content.MutableContextWrapper

consider this part of our future work. For the remaining calls, displaying the library’s type and the specific permission requested can significantly improve the existing permission system and better guide users into making informed decisions based on the app’s intended functionality.

Permission mapping inconsistencies. While Reaper relies on permission mappings provided by prior work, our system can be used to dynamically validate those statically generated mappings. Thus, while not part of our study’s main focus, we conduct an exploratory study as more accurate mappings will further improve the main functionality of our system; we opt for API 22, since it is the most recent version with the most accurate permission mappings. We created a mock application that sequentially executes all the permission-protected calls of the Android SDK, and verified the permission of each function call based on the permission check occurring in the Android Server. Table 2.2 shows some of the inconsistencies and missing entries that we have found. The `divideMessage()` function exists in two different classes, and PScout and AXPLORER report different permissions for this function. Using Reaper we found that this function does not need a permission. To further verify this result, we triggered this function without declaring any permission in the app’s manifest file and observed the same functionality without any warnings, errors, or crashes. We also manually investigated certain functions that were not mentioned in either study, and found that prior work missed `sendStickyBroadcast()`, which requires the `BROADCAST_STICKY` permission.

Interestingly, we find that functions being permission-protected also depends on the arguments provided. For example, the function `getprovider()` from the `LocationManager` class is permission-protected when provided with `GPS_PROVIDER` but does not need a permission for `KEY_LOCATION_CHANGED`. We argue that there is dire need for better documentation of the internals of Android permissions, as such scenarios can further confuse developers.

2.6 Discussion and Limitations

Defining origin. Reaper distinguishes core from third-party functionality based on the origin of the executed code. We compiled a list of libraries using data from [28, 70] for

identifying their origin. If a library is not in the list, the corresponding stacktrace will not be flagged as third party functionality. Similarly, our library classification relies on external resources [6, 28]. As lists of libraries are extended and become more complete, Reaper’s coverage will also increase. Moreover, core functionality could potentially be misclassified if a function has the same name with that of a known library. We investigated our dataset and did not find such instances.

Call mappings. Reaper maintains a permissions-to-function-calls mapping that contains the functions that should be monitored. While we have used Reaper to validate these mappings, expanding the mappings reported by prior work is out of our scope; thus, our system will not monitor PPCs missing from that list.

Native code. Android apps are written in Java or in native code. Xposed is able to hook functions written in Java, as well as native code in cases of JNI. However, we cannot hook custom native code written by developers since it is not supported by Xposed.

Kernel permissions. Certain Android permissions are regulated by the kernel. Since Pscout and AXPLORER did not conduct a native code analysis and have not created mappings for such permissions, we have not included these permissions in our study.

Graph coverage. UIHarvester may miss displayed content when apps use wrappers or webviews, as will UI Automator.

Emulators. Since apps or libraries can identify that they are being executed in a virtual environment [206], our results may present a lower bound of the privacy risks posed by libraries.

Obfuscated package names. While PPCs that originate from obfuscated package names only account for 1.14% in our study, Reaper could incorporate a static analysis tool like LibScout to reverse the obfuscated package name back to its original form.

Chapter 3

End Host Protection

3.1 One Flew Over the Tracker’s Nest

In the previous chapter we identified that apps leak personally identifiable information. Thus, it is reasonable for privacy-aware users to prefer using web browsers instead of apps to access online services. However, this is not always possible, or desired [250]. As a result, the use of mobile apps is, in many cases, unavoidable. To provide these users with better privacy guarantees, we propose *antiTrackDroid*: an anti-tracking mechanism able to preserve the privacy of the users by blocking many personal and device information leaks to any third parties. Specifically, antiTrackDroid is a module which filters all outgoing requests and blocks the ones delivering tracking information.

The core design principles of antiTrackDroid include the ability to operate (i) for all apps, and (ii) without the need for any additional infrastructure (e.g. VPN, Proxy, etc.). To meet these principles, antiTrackDroid leverages Xposed [224]: a popular Android framework, which allows system-level changes at runtime without requiring installation of any custom ROM or modifications to the application. By using Xposed, antiTrackDroid is able to intercept every outgoing request and check if the destination’s domain name exist in a blacklist of mobile trackers. In case of match (i.e. the destination is blacklisted), the outgoing request is blocked. Figure 3.1 summarizes the design of our approach.

3.2 Implementation

To assess the effectiveness and feasibility of our approach, we implemented a prototype of antiTrackDroid for Android. Our system consists of the following main components:

1. The *Filtering module*, which implements the `IXposedHookLoadPackage` and filters the tracking requests based on the Xposed module.
2. An *Android Activity* (hereafter named Launcher), with a graphical user interface to allow the users configure the Filtering module.

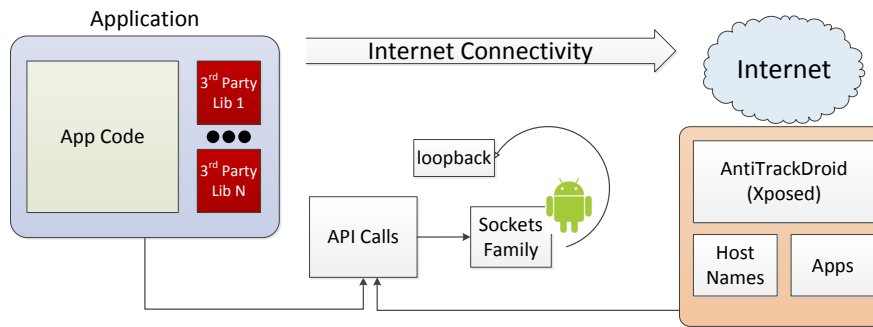


Figure 3.1: Defense mechanism overview.

3. The *AppList Updater*, which listens for newly added or removed packages and updates the list of applications being monitored, using a Broadcast Receiver.

Launcher Activity. Launcher acts as an interface between the Filtering module and the user. It contains a menu allowing the user to (i) load a different blacklist or exclude an application from the filtering procedure. Launcher is also responsible of maintaining two different data structures: a HashSet with the tracking domain names loaded from the blacklist, and a HashSet with the applications being monitored. By using HashSets, antiTrackDroid is able to perform look-ups with $O(1)$ complexity reducing significantly the per request latency overhead.

Filtering Module. Mobile applications send data over HTTP/HTTPS requests by using TCP sockets. Therefore, Filtering module dynamically hooks on the constructor of the TCP socket opened by the applications residing in the HashSet of monitored apps. In addition, it re-writes the destination IP address with localhost in case of a blocked request. This loop-back interface is a virtual network interface that does not correspond to any actual hardware, so any packets transmitted to it, will not generate any hardware interrupts. By redirecting to loop-back, antiTrackDroid avoids possible crashes of apps caused by aborted connections.

AppList Updater. Since users may install or remove applications at any time, our system must be able to update the list of monitored apps. In Android, every time an app is added or removed in the system, a broadcast message is sent through the PackageManager component, which can reach any app in the device. By using a Broadcast Receiver [59], the AppList updater, running as a background service, can listen such messages and update the list of monitored apps.

Blacklist of Trackers. To determine if a request is a tracker or not in the *Filtering Module*, we use the popular mobile-based blacklist of AdAway [147], which we extended by including the tracking domains we collected manually during our privacy leak analysis [201]. Our blacklist of antiTrackDroid currently contains 66k entries in total. Recall that in Launcher Activity, the user is free to change the used blacklist by loading one of her choice.

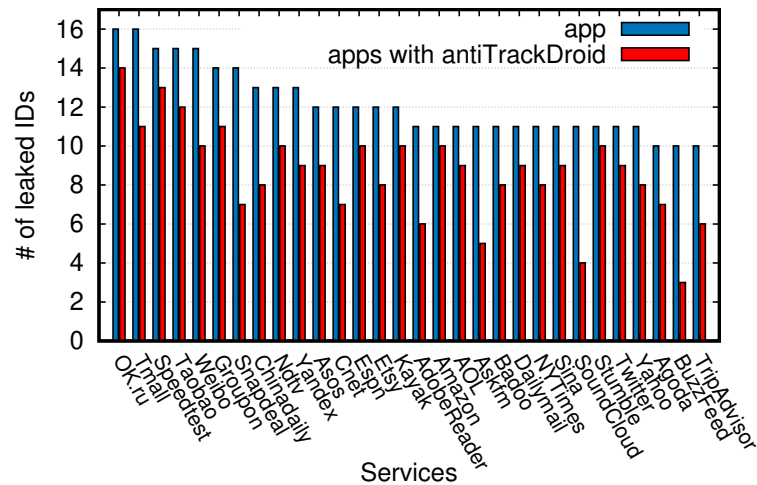


Figure 3.2: Number of leaked ID without and with antiTrackDroid for the 30 apps with the higher number of ID leaks.

3.3 Evaluation

In this section, we evaluate the effectiveness and performance of our antiTrackDroid, and we explore its benefits.

3.3.1 Privacy performance

To evaluate the privacy preservation of antiTrackDroid, we inspect the identifiers leaked to the network with and without the use of antiTrackDroid. In Figure 3.2, we see the number of leaked IDs with and without antiTrackDroid for the 30 more leaking apps. Our results show that antiTrackDroid is able to reduce the number of leaked identifiers by 27.41% on average. Note that since our approach blocks the majority of third party trackers, the rest of the leaking IDs exist due to requests destined to the developer’s first party domains and content providers (e.g. CDNs). Blocking such requests would cause degradation of the user experience or even fatal error to the application.

3.3.2 Latency overhead

Although antiTrackDroid significantly improves privacy, it may have an impact on the overall latency of the apps as well. Indeed, antiTrackDroid may increase latency because it includes an extra check with the blacklist. On the other hand, it may significantly reduce the latency imposed by blacklisted tracker requests as these requests will be blocked and the app will not have to suffer their latency. To measure the impact on latency, we created an Android app, with which we can send arbitrary number of requests to a server of ours.

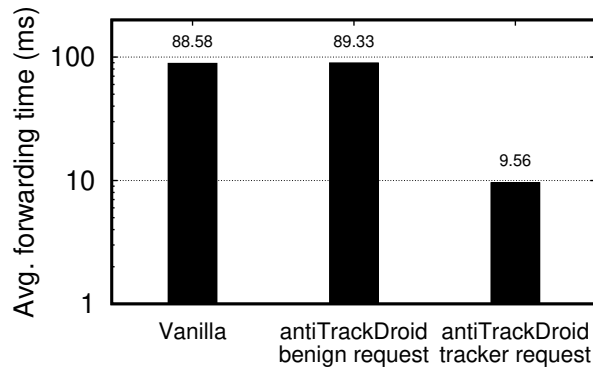


Figure 3.3: Overall request forwarding time with and without antiTrackDroid.

Thus, we create 1000 requests carrying 15KB of data each, and we send these requests to the server sequentially after a short time interval. We run this experiment 3 times: (i) once with antiTrackDroid switched off, (ii) once with antiTrackDroid enabled and the domain not included in the blacklist (benign request), and (iii) one more with antiTrackDroid enabled and the domain of the server blacklisted (Tracker request).

Figure 3.3 shows that the vanilla (no antiTrackDroid) request (see 1st bar) takes about 100 ms. When we switch on antiTrackDroid (see 2nd bar), the latency is practically the same. Indeed, a few lookups in a blacklist do not add any overhead noticeable in the 100 ms range (less than 1 ms). Finally, when we switch on antiTrackDroid and make an access to a tracker (see 3rd bar), the latency drops to less than 10 ms as the request is blocked. We are happy to see that antiTrackDroid, not only improves privacy, but it also improves performance.

3.3.3 Benefits from the use of antiTrackDroid

Besides preserving the user's privacy, the blocking functionality of our approach improves also the performance of apps in the user's data-plan and battery.

Bytes transferred. By blocking the tracking related requests antiTrackDroid is able to save a significant amount of data, an aspect of great importance when it comes to mobile users with specific data plan. To determine the different volume of data transferred to/from the apps in a device running antiTrackDroid, we conduct the following experiment: we run all apps in our device as previously, but instead of blocking the requests we calculate the outgoing bytes of requests and the incoming bytes of the associate responses. In addition, we measure the overall traffic of the app and finally calculate the portion of traffic marked as tracker-related. Figure 3.4, presents the results, where we see that antiTrackDroid reduces the volume of transferred bytes by 8% for the 50% of the apps.

Energy cost There are several studies [133, 180] attempting to measure the energy cost imposed by the ad-related content to a user's device. It is apparent that every connection

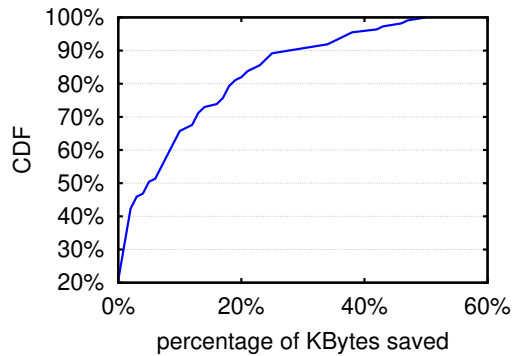


Figure 3.4: Percentage of KBytes saved.

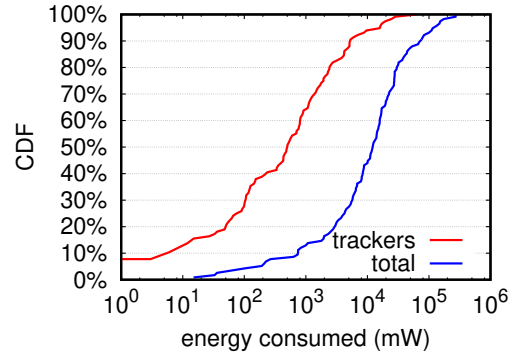


Figure 3.5: Watts saved due to the less tracking request sent.

an app opens with a network entity, it imposes an overhead to the overall energy consumption of the device [191]. As a consequence, by reducing the requests an app sends or receives, along with their transferred data, antiTrackDroid is able to reduce the energy cost of the application as well. Measuring the energy consumption in a mobile device is a challenging task. In order to estimate our gain with antiTrackDroid, we perform a simulation, based on the energy readings of Appscope [283]. Figure 3.5 presents the distributions of the per-app power consumption for (i) the total and (ii) the tracker-related transferred bytes. From our simulation we find that there is a significant reduction of about 7,5% for the 50% of the applications.

3.4 Discussion and Future Work

Interestingly, the antiTrackDroid can be combined with the functionality offered by Reaper (i.e., origin information) in order to create an end host firewall independent of domain names which is faster, more lightweight, and more effective than previous blocking approaches. This firewall operates by checking the origin of the system call that initiated a connection based on a list of third-party library package names. The Android documentation states that every app package name has to be unique. For usability and easier integration, the same principle applies for third-party libraries. Since each third-party library has its own package name the proposed blacklist will contain one entry per third-party library, in contrast to a domain blacklist where multiple domain names exist for the same campaign. Using package names as identifiers not only reduces the size of the blacklist but also removes the need for pattern matching with regular expressions. The proposed solution allows to efficiently use an $O(1)$ complexity algorithm with constant access time (e.g. HashSet) while maintaining the size of the blacklist manageable. Finally, this protection mechanism does not depend on a VPN or a proxy but operates directly on the user's de-

vice through in line monitoring of the appropriate API calls. The inherent advantage of this approach is the ability to process outgoing connections in less than 1 ms [200]. Furthermore, by identifying sensitive data being accessed when the actual function call is issued, our system is not affected by data subsequently being encrypted or obfuscated prior to an exfiltration attempt. We consider incorporating origin information in antiTrackDroid and evaluating this tool in the wild as future work, since it will enable more effective user protection mechanisms at a more granular level without preventing desired application functionality (e.g., access control tools that prevent PII leaks).

Chapter 4

The Risks of Mobile Sensor-based Attacks

Mobile phone usage has been on a constant rise, and smartphone devices have reached a near-ubiquitous presence in many countries. According to reports, almost all mobile phone owners in the United States own a smartphone [156]. Amongst the rich set of functionality offered by such devices web browsing remains popular. Smartphone usage has gained such traction, that in 2016 more internet traffic originated from mobile devices than desktop computers worldwide [126] and 56% of the traffic to top-sites in the United States was from mobile devices [246]. This upward trend has been influenced by many factors, including improvements in the speed of mobile internet connections, mobile browsers offering more features, and improvements in smartphone hardware [115].

On the other hand, apart from the obvious usability benefits, smartphone devices have also introduced a plethora of privacy risks. In this post-Snowden era [131] users are becoming increasingly aware of privacy issues including online tracking and internet surveillance, and employ private browsing among other techniques to remain anonymous online (despite overestimating the protection it actually offers [277]). Nonetheless, private browsing offers some protection against users being tracked across different sessions [276]. However, adversaries can still track users through browser or device fingerprints [194]. This consists of collecting characteristics of the device environment and the browser itself, making it possible to identify which device is navigating a given webpage [154]. Prior work has also shown that manufacturing imperfections in sensors' hardware render them fingerprintable [106].

Websites can access mobile sensor data through the HTML5 WebAPI, which is supported by major browsers (with certain variations). The WebAPI is not limited to mobile devices, and offers a rich set of capabilities to modern websites. Accordingly, Snyder et al. [238] presented a cost-benefit analysis of the functionality of the WebAPI using a small set of websites, by correlating WebAPI calls to vulnerabilities reported in CVE reports and relevant academic attacks (including tracking, cross-origin information stealing [257], and timing attacks [132]). As that study focused on desktop browsing and was limited in scale, Das et al. recently explored the pervasiveness of mobile device fingerprinting [99]. How-

ever, this is only one of the threats that the WebAPI poses to mobile users and no comprehensive large-scale exploration currently exists. In practice a plethora of attacks that were previously limited to mobile apps can “migrate” to the mobile web, as modern browsers provide access to a device’s underlying mobile sensors.

In this chapter we present a *quantitative* and *qualitative* large-scale study of mobile-specific WebAPI calls made by websites in the wild. We build a unique crawling infrastructure that uses real Android devices and perform an end-to-end analysis of WebAPI requests. In more detail, apart from injecting a script in websites that allows us to hook all WebAPI calls, we leverage a dynamic real-time app-analysis system that allows us to trace the internal behavior of the Android OS when calls occur, ensuring the fidelity of our measurements. Using our crawling infrastructure, we measure the prevalence of mobile-specific WebAPI calls across 183,571 of the most popular websites during March-September 2018. Our experiments capture the true scale of this phenomenon, as we detect 5,313 unique domains accessing at least one mobile WebAPI call; 35.89% of those also result in sensors being accessed by third-party scripts that originate from 11 second-level domains. To better understand the implications and potential threat that users face, we survey prior literature on attacks from malicious apps that leverage data from mobile sensors. Based on this diverse yet representative selection of papers we create a taxonomy of attacks that can be potentially carried out by modern websites by obtaining seven categories of sensor data. We then break down the different attacks based on their sensor requirements and conduct an in-depth analysis of our dataset, and find that 3,008 websites request access to sensor data needed for at least one attack. While we refer to attacks, these capabilities include privacy-invasive behavior (e.g., inferring a user’s sensitive demographic information for personalizing an ad) that can be carried out by, otherwise, legitimate websites.

Our extensive analysis reveals that popular websites, as well as websites from popular categories (e.g., e-banking), tend to access more mobile sensors which consequently leads to the feasibility of conducting more sensor-based attacks. Interestingly at least one domain from *every* category in our dataset could potentially infer the user’s input, which is also the most frequently feasible attack across all categories and can be used to steal sensitive information (e.g., credit card information and pin number). Moreover websites do not need to access a lot of different sensors to perform these attacks; readings from the motion and orientation sensors, which do *not* require any permission at the operating system level, can lead to 9 and 8 different attacks respectively. We argue that with different browsers enforcing different access policies, as do the plethora of apps that support Web-View, there is dire need for a standardized, fine-grained universal mechanism that allows users to control access to *all* types of mobile sensor data.

4.1 The Seven Deadly Sins of the HTML5 WebAPI

In this section we provide an overview of mobile-specific calls supported by the HTML5 WebAPI and then present a taxonomy of attacks presented in prior work that rely on data provided by mobile sensors.

4.1.1 HTML5 WebAPI

Browsers have evolved significantly from their original design, both in terms of the functionality they provide as well as their underlying complexity. At the same time, the advent of smartphones and their almost ubiquitous presence has enabled a wide range of previously-infeasible functionality due to connectivity on-the-go and the information returned by embedded mobile sensors. As such, many capabilities previously restricted to native apps are now available to websites. While the WebAPI introduces obvious usability benefits to end users as it can improve the overall experience, it also poses a significant privacy and security risk. Apart from enabling certain forms of user tracking (e.g., through the discontinued Battery API [197]) other sensor-based attacks that were previously restricted to mobile apps can now be deployed over the web. To better explore this threat, we focus on all mobile-specific HTML5 WebAPI calls, and subsequently explore the attacks that they enable. In detail, our study focuses on the following WebAPI calls

DeviceMotionEvent.acceleration [62]: This call provides web developers with information from the accelerometer sensor about the speed of changes in the device's position, returning values expressed in m/s^2 for all three X, Y, Z axes.

DeviceMotionEvent.rotationRate [182]: This call returns information from the gyroscope sensor about the rate at which the device's orientation changes along the three orientation axis (alpha, beta, gamma). This value is expressed in degrees per second.

DeviceOrientationEvent [222]: This event is fired when the accelerometer detects a change to the device's orientation (i.e., from landscape to portrait and vice versa).

DeviceProximityEvent [63]: This allows websites to obtain information about the distance of a nearby physical object using the device's proximity sensor. The value is returned in centimeters. For instance, this information can be used for energy conservation by turning a device's screen off when the user is talking on the phone.

DeviceLightEvent [60]: Websites can obtain information about changes in the device's environment by indicating changes in the intensity of the light as measured by the ambient light sensor, which is expressed in lux units.

Geolocation [58]: This set of API calls allows web developers to retrieve the geographical position of a smartphone device in real time. This is done at two levels of granularity: fine which relies on readings for the device's GPS, or coarse which relies on information of the WiFi network the device is connected to. Specifically the `getCurrentPosition()` method instantly retrieves the position of the device, while the `watchPosition()` method

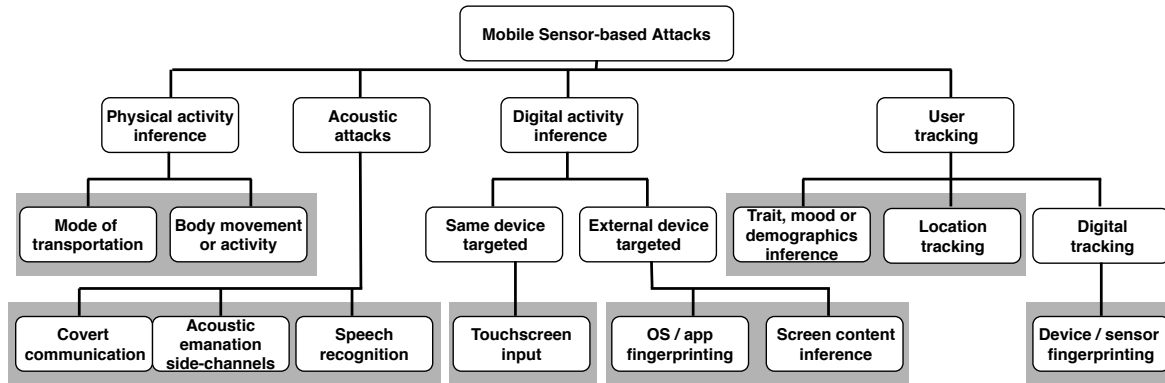


Figure 4.1: Taxonomy of attacks demonstrated in prior studies that leverage data from mobile sensors.

returns the position of the device using a new a process that continuously polls for the current location.

getUserMedia [95]: This set of calls provides access to the device’s camera and microphone sensors.

vibrate() [61]: This allows websites to control the smartphone’s vibration engine, and supports different vibration patterns of different durations to be sent to the device.

4.1.2 Attack Taxonomy

A plethora of research papers have demonstrated mobile-based attacks that employ sensor data. While a considerable number of attacks present similar characteristics, e.g., demonstrating different techniques for inferring a user’s touchscreen input or fingerprinting the user’s device, a wide range of different attacks have been proposed. Here we introduce a taxonomy of attacks compiled from the literature that captures the vast potential of how the seven different categories of mobile sensor data can be misused by adversaries. Typically these attacks assume that attackers are able to obtain sensor data through a malicious app installed on the device. However, in practice, modern browsers can mediate data exchange between websites and sensor data through the HTML5 WebAPI. This leads to a different threat model and an increased attack surface, as it removes the constraint of users having to install a malicious app; simply visiting a website can expose users to these attacks. While this might affect the accuracy of certain attacks (e.g., due to a website being able to obtain sensor data for a shorter amount of time compared to an app) it remains an important and largely unexplored risk for mobile users.

In Figure 4.1 we present our taxonomy which aims to highlight the variety of attacks enabled by sensor-data, while simultaneously obscuring the type of sensor used for each attack. We do not include explicit sensor information in our taxonomy, as prior attacks

often obtain the same objective while using different combinations of sensors (as can be seen in Table 4.1). At the same time we opt for a relatively fine-grained first level, and specifically consider acoustic attacks as a separate class due to their unique and diverse nature, instead of including them as sub-classes of physical and digital activity inference attacks. Next we briefly describe the four main classes from our taxonomy's first level and refer to some of the presented attacks.

Physical activity inference. Numerous studies [134, 140, 174, 187, 216] have demonstrated that mobile sensors can be used to infer information about personal everyday activities. For example it is possible to infer whether the user is walking, running or their mode of transportation, by leveraging the Motion and GPS sensors [216].

Acoustic attacks. [57, 102, 123, 124, 171, 172, 177, 226] showed that access to Accelerometer, Gyroscope or the Vibration API can be used to infer users' credit card numbers by listening for specific frequencies [226] or what a user is typing on a physical keyboard [172], and bypassing dynamic analysis systems and antivirus products through covert channel attacks [171].

Digital activity inference. This class includes a wide range of attacks, with prior work [79, 83, 86, 117, 138, 174, 198, 243, 280] showing that sensor information (including the Accelerometer and Gyroscope) can be used to predict what the user is typing on the smartphone's touchscreen (e.g., [174, 198]). This is possible because typing leads to changes in the position of the screen, its orientation and the device's motion. In a different study, the Light sensor was used to identify the content of an external display and even classify users' digital activities into different categories with an 85% accuracy [86].

User tracking. Identifying and tracking users across the web has garnered much attention [55, 56, 80, 97–100, 106, 117, 134, 142, 143, 177, 187, 216, 220, 289, 291]. This can be conducted in different ways, from coarse-grained location tracking that does not require any user-permission (using just the Accelerometer or Gyroscope) [134, 187], to fine-grained device fingerprinting using rich and high-resolution data from smartphone sensors (e.g., [56, 106]). In this category we also include alternative attacks that could track users by inferring demographic information (e.g., age [100], gender [177] and fingerprints [117]), physical traits such as their gait [143, 220], or information about their mental state or mood [291].

Deconstructing sensor attacks. Table 4.1 lists the different sensor-based attacks previously described in the studies that guided our taxonomy. We classify previous attack papers based on the taxonomy introduced in Figure 4.1. Subsequently, we break down all the attacks presented in those papers based on the type of sensor data needed to carry out the attacks. If an attack can be carried out using a single sensor, that sensor is denoted with ●. For attacks that require multiple sensors to succeed we mark the sensors with ◐. For sensors that are not required, but can be used to improve accuracy we use ○. For example the technique in [216] that infers the body movement or activity of a user requires access to the motion and GPS sensors. On the other hand, [187] only requires the orienta-

tion sensor, but using the Motion or Magnetometer sensor can further improve the attack. Even though access to the magnetometer is not currently supported by Firefox [20], which we used, we include it for completeness.

Table 4.1: Sensor based side-channel attacks.

Ref.#	Attack	Motion	Orientation	Light	Vibration	Magnetometer	Media (Camera, Mic)	GPS
[216] [134, 136, 140] [192] [187] [271, 282] [248]	Mode of transportation	●	-	-	-	-	- -	●
[216] [174, 282] [152] [248]	Body movement or activity	●	●	-	-	-	- -	●
[139, 216] [134] [187] [190] [271] [192]	Location Tracking	●	●	-	-	-	- -	●
[172] [123, 124, 279, 290]	Acoustic emanation side-channels	●	-	-	-	-	- ●	-
[177] [57]	Speech recognition	●	●	-	-	-	- -	-
[83, 138, 174, 179, 207, 280] [198], [67], [84] [84] [243] [117, 210] [232] [186] [235]	Touchscreen input	●	●	●	-	-	● -	-
[79]	OS/app fingerprinting	-	-	-	-	●	- -	-
[86]	Screen content inference	-	-	●	-	-	- -	-
[100, 220, 291] [82] [177] [117] [121, 143] [153]	Physical trait or demographics inference (e.g., age, sex, mood, fingerprints, gait)	●	●	-	-	-	● -	-
[98, 99, 142, 282] [80] [56] [106] [97, 289] [55]	Device/sensor fingerprinting	●	●	-	-	-	● -	-
[171] [226] [102] [284]	Covert communication side-channels	-	-	-	●	-	- ●	-

Sensors marked with (●) are sufficient for performing the specific attack. When a combination of multiple sensors is required to perform the attack, they are marked with (●). We denote optional sensors with (○) (e.g., that data is optional and enhances the accuracy of the attack). When papers present multiple attacks, combinations of all of the above may be present in the table. Grey columns denote sensor data that should require explicit user permission according to the W3C.

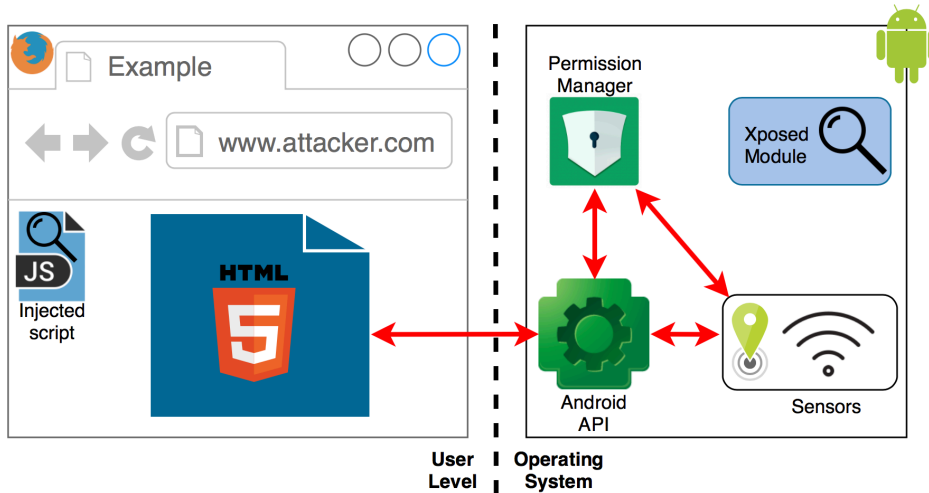


Figure 4.2: Overview of our crawling system’s architecture. Our system components provide an end-to-end view of requests to access the mobile sensors. The red arrows denote communication “pathways” observed by our system.

4.2 Methodology and System Design

In this section we present our system design and experimental methodology. We give an overview of our system’s architecture, and provide implementation details about the in-line hooking methods for intercepting both JavaScript and Android system call functions.

System architecture. Our system employs a transparent proxy server that intercepts network traffic by using `mitmproxy` [93]. We configured all the Android devices used in our experiments with `mitm`’s certificate in order to intercept both HTTP and HTTPS traffic. As can be seen in Figure 4.2, the proxy server injects a JavaScript component that hooks and monitors JavaScript calls to mobile-specific WebAPIs. However, our aim is to obtain an in-depth view of sensor data access. In general, browsers are responsible for mediating access between high-level JavaScript function calls and low-level Android API calls. Understanding how this mechanism works for every browser would be time consuming and in many cases infeasible due to proprietary code. As such, we have opted for a generic and browser-agnostic approach, where we intercept Android system calls using a custom module for the Xposed framework [224] that (i) detects and hooks requests to sensor-specific Android API calls and (ii) identifies which of these API calls are permission-protected through the list of permission protected calls found in [11] (such requests are handled by the Permission Manager). By intercepting these low-level function calls we are able to validate that the JavaScript interception was successful and calls requesting sensor data were correctly logged.

Mobile HTML5 Functions. We identified the functions that retrieve mobile-specific data through the official mobile HTML5 WebAPI [118], which lists all the available fea-

tures. We consider as *mobile-specific* any calls that obtain information originating from an integrated sensor of a mobile device. The HTML5 WebAPI calls interact with the webpage using either a direct one-time communication (i.e., Vibration and Media capture) or through an event listener, since some sensors (i.e., Motion, Orientation, Proximity, and Ambient Light) continuously fire events in order to provide up-to-date readings in real time. For Geolocation one call exists for each category. The full list of identified mobile HTML5 functions maps to seven mobile sensors, as shown in Table 4.1.

JavaScript Calls Interception. We build our component for hooking JavaScript methods upon the `javascript-hooker` Node.js module [77]. This script allows us to hook any JavaScript function called in a webpage, and passes as arguments the actual object, the method to hook and the function that will be called before the execution of the hooked method. The script creates a wrapper around the functions and substitutes the original method. In our experiments we do not overwrite the original function but only need to identify whether a function is called. Thus, whenever a mobile HTML5 WebAPI is called the JavaScript modules creates a log entry for further analysis and executes the original function. Since `javascript-hooker` also takes the arguments of the original function, we can also intercept the arguments of the `addEventListener` and check for events of interest. Our code is directly injected in the head of the document (if there is one) or the page body otherwise.

In order to listen to events and associate a function to a specific target we need to intercept the setter property. Even though this is possible using `Object.defineProperty()` the original value will be lost and the webpage may not function as expected. Therefore, we follow the approach employed by Chameleon [125] and overwrite the `getter` property of each event prototype instead of substituting the setter relative to the target property. As such, every time the property is read, our custom function is called. While certain sensor data may normally remain the same during navigation (e.g., properties related to display characteristics), remaining constant might be considered “suspicious” for other sensors. For instance, when an actual human uses the device, small changes in the gyroscope readings would be expected. As such, to make our crawling more realistic, our system intercepts the values returned by certain sensors and slightly modifies their value while ensuring that the result remains “valid”; For instance, a gyroscope orientation reading is a decimal number between -365 and $+365$. In general, data retrieved through events, is handled in two different ways: listening to `addEventListener` on the target object while checking if the argument matches the desired event and defining new getters for the properties of the event’s prototype.

Identifying the JavaScript source. Apart from logging WebAPI calls we also want to identify the origin of the JavaScript files being executed. This information is important in order to identify if the script belongs to a first-party domain or a third-party domain. We register the source of the URL by utilizing the `stack` property of the `Error` object. Our

hooking script implements a mechanism that creates an Error object and reads its stack property.

Android API call interception. Each mobile HTML5 WebAPI is associated with a low-level Android API call. In order to validate the results of the JavaScript interception and to identify which ones require a permission, we use the PermissionHarvester [109] module that hooks every Android permission protected API call and logs the current stacktrace. Since access to some of the sensors does not require an Android permission, we also manually identified and hooked the functions that give access to non-permission-protected sensor data. Android applications (including the device's browser) cannot directly read the current value of a sensor and are required to register a listener in order to consequently read the captured events. Each sensor can be obtained by calling the `getDefaultSensor()` method of the `android.hardware.SensorManager` class. The listener is declared by specifying the name of the sensor with the `getDefaultSensor()` function, and a `Sensor` instance is created. Finally, the listener is registered by calling the `registerListener()` method. Our module intercepts both of these function calls.

Experimental setup. Among popular browsers for Android, Google Chrome and Mozilla Firefox have better compatibility for HTML5 WebAPIs [118]. Since Chrome relies heavily on Google Play Services for the Android internals, while Firefox more clearly leverages the official and better-documented Android API, we opted for the latter. In our experiments we use Mozilla Firefox (v.59.0.1) as our browser on three Android Google Nexus 5X and a OnePlus One device, all running AOSP 7.1.2. We controlled the devices using custom scripts and the Android Debug Bridge. We first evaluated the effectiveness of our methodology by creating a dummy website that executes all possible mobile HTML5 WebAPIs. Since browsers require a valid certificate in order to call certain APIs (i.e., they are only served over HTTPS) we used a self-signed certificate. Before conducting our actual large-scale study, we confirmed that our approach can successfully intercept and monitor access to the devices' sensors. Our system also simulates brief user interaction through random gestures (swipes and taps) with websites so as to elicit functionality from websites expecting some user activity. Gestures are issued for approximately 30 seconds on average for each website, while an extra module monitors for potential redirections to different domains (e.g., due to clicking on an ad) which are rolled back so the original website can continue to be processed.

4.3 Data Collection and Analysis

In this section we present our findings from our large scale study on the use of mobile-specific WebAPI calls in the wild. Our crawling list included the 200K most popular websites according to Alexa, as returned on 03/24/2018, to be processed by our crawling infrastructure comprising of four Android smartphones. Our system was unable to access or

Table 4.2: Number of domains using mobile WebAPI calls.

WebAPI	#Domains	WebAPI	#Domains
Device orientation	2,199	Ambient light sensor	152
Geolocation	1,688	Proximity sensor	142
Device motion	1,360	Vibration	84
Screen orientation change	645	Media capture	12
Total 6,282			

complete the crawling process for 16,199 (8%) of the domains in our list (e.g., 503-timeout, 502-Bad Gateway, or DNS errors), and omitted 230 domains flagged as malicious by the Google SafeBrowsing API. Our crawling experiments, from US-based IP addresses, took place between 03/24/2018-09/03/2018 and 11/11/2018-11/22/2018.

In Table 4.2 we can see the prevalence of the mobile-specific WebAPI calls logged by our system among the 183,571 domains processed by our crawling infrastructure. We logged 5,313 (2.89%) websites using at least one of the targeted APIs, while 807 request access to sensor data using more than one of the API calls. The most prevalently accessed data is from the acceleration and orientation sensors which do not require the user’s permission, as well as geolocation data which requires permission in major browsers. While the Geolocation API can also return information for desktop computers (using “information about nearby wireless access points and the IP address” [23]), we consider it mobile-specific due to smartphones’ integrated GPS receivers which provide real-time location information. While geolocating users based on landline IP addresses is considerably accurate [269], that is not the case for mobile IP addresses [73, 260]. It is important to note that the Media capture and Geolocation APIs should explicitly request permissions from the user; while this is enforced in major browsers, it is not always the case with other browsers (e.g., for Geolocation [148]). For the remaining WebAPI calls, users will be unaware that such information is being retrieved by the website even for major browsers, which occurs in 4,582 (2.49%) of the websites we processed.

As can be seen in Figure 4.3 the use of mobile-specific WebAPI calls is not uniform across our dataset. Indeed, the highest concentration is found in the top 5K websites with 250 domains, which is more than double the 122 domains in the last chunk 195K-200K. Moreover, all the chunks above the 150 domain-threshold are found within the top 60K. Overall, most chunks contain between 100 and 150 domains requesting access to mobile-specific API calls. Domains also access more *permission-free* WebAPIs (gray bars) independently of their rank, indicating the importance of this sensor data. As discussed later on, this type of information can be used for a plethora of attacks, and users should have the ability to explicitly grant permission for them.

As Figure 4.4 (left) shows the majority of websites issue request access to a single sensor

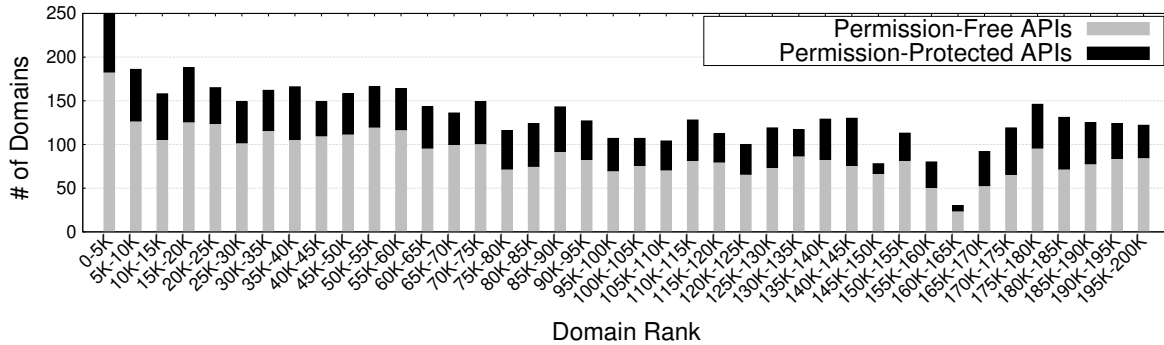


Figure 4.3: Number of domains using mobile-specific WebAPI calls. Gray bars indicate domains that only use APIs that do not require a permission, while black bars indicate domains also accessing permission-protected APIs.

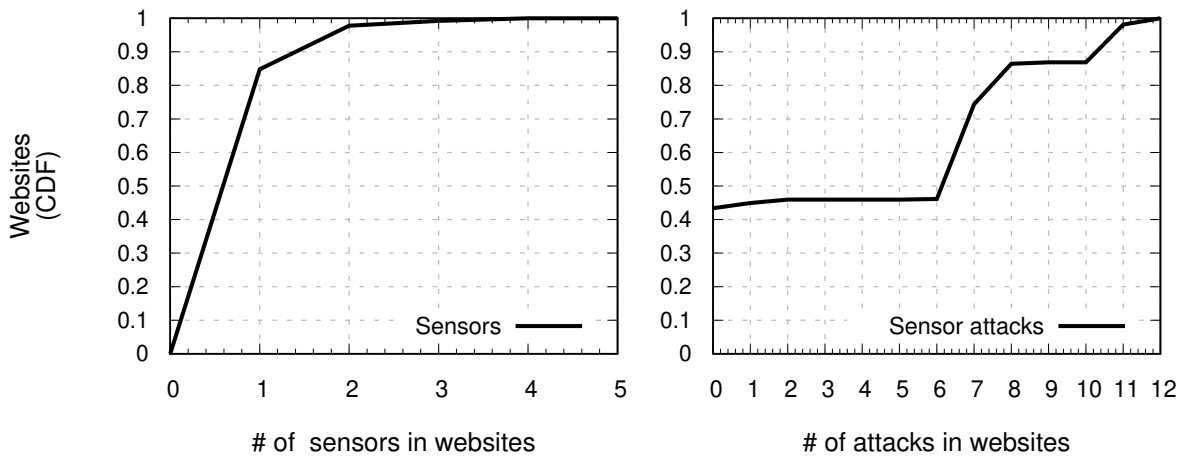


Figure 4.4: CDF of websites requesting access to sensors (left) and the number of attacks that are feasible with the data they collect (right).

through the WebAPI, while 15.1% of the domains we processed target at least two different types of sensor data. As shown in Table 4.1, only accessing the Motion sensor can lead to six different attacks, while a combination of two sensors (Motion and Orientation) leads to eight attacks. Furthermore, as can be seen in Figure 4.4 (right) 56.6% of the domains that issue mobile-specific WebAPI calls are able to perform at least one attack.

Sensor-based attacks. Next, we continue our analysis of the dataset collected by our system by framing it within our taxonomy based on representative prior work. It is important to note that in our analysis we do not take into account or argue for (or against) the *plausibility* of the attacks presented in previous studies. Instead, our goal is to measure the potential risk that mobile users face due to web browsing by identifying websites that request access to specific sensor data and could *potentially* misuse them in an invasive or malicious manner.

Table 4.3: Breakdown of sensor based attacks, the number of domains capable of deploying them and the percentage of webpages capable of performing the specific attack.

ID	Mobile Sensor-based Attack	#Domains	Percentage
1	Mode of transportation	2,861	53.85%
2	Body movement or activity	720	13.55%
3	Location tracking	2,861	53.85%
4	Acoustic emanation side-channels	1,372	25.82%
5	Speech recognition	2,199	41.39%
6	Touchscreen input	2,926	55.07%
7	Screen content inference	152	2.86%
8	Inferring user's age	1,360	25.60%
9	Inferring user's sex	2,199	41.39%
10	Inferring user's fingerprints	12	0.23%
11	Inferring user's gait	2,861	53.85%
12	Inferring user's mood	1,372	25.82%
13	Device/sensor fingerprinting	2,873	54.07%
14	Covert channels	96	1.81%

Table 4.3 breaks down the number of domains for each attack. We observe that the most common attacks across websites that access WebAPIs are touchscreen input (55.07%), device/sensor fingerprinting and trait, mood or demographic inference (54.07%), location tracking and mode of transportation (53.85%) and speech recognition (41.39%). As can be seen in Table 4.3, the most commonly feasible attack enabled by collected mobile sensor data is the inference of the user's touch input. This would allow a malicious domain to exfiltrate extremely sensitive information, such as the user's credit card number, her login credentials, or even private chat messages. The attack surface grows considerably due to third-party scripts accessing WebAPIs (see Section 4.3.2), since they create a hidden covert channel and can exfiltrate sensitive data even if the user is browsing a legitimate webpage. We argue that any information gained from sensors poses a risk for users and an access control policy should be enforced, either through some form of run-time permissions [29] or using a mechanism similar to GDPR [12] where users are informed and have to explicitly give their consent.

4.3.1 In-Depth Analysis and Case Studies

Here we continue our analysis and present a series of case studies. We first classify every domain that accesses at least one WebAPI call using McAfee's real time database [173]. In Table 4.4 we provide the top 20 categories (sorted in descending order) based on the average number of access requests for sensor data across all the websites of each category. In Table 4.7 we provide a full list of the classification performed. The first column denotes the classification label, while the second, third and fourth columns denote the number of do-

Table 4.4: Domain classification for the top 20 categories sorted in descending order based on the average request access for sensors across websites accessing at least one mobile sensor.

Label	# Domains	# Sensors	# Attacks	% Sensors/Total Domains
Business	662	743	2246	13.98%
Online Shopping	427	539	2316	10.14%
Marketing/Merchandising	417	459	1380	8.64%
Entertainment	348	439	2118	8.26%
Travel	322	392	1492	7.38%
General News	327	385	1640	7.25%
Education/Reference	293	321	1380	6.04%
Internet Services	301	318	1169	5.99%
Finance/Banking	239	307	955	5.78%
Fashion/Beauty	169	249	1218	4.69%
Blogs/Wiki	201	236	1272	4.44%
Public Information	201	226	388	4.25%
Software/Hardware	200	214	569	4.03%
Pornography	136	196	983	3.69%
Potential Illegal Software	117	181	990	3.41%
Health	128	170	480	3.20%
Games	115	143	865	2.69%
Restaurants	126	139	192	2.62%
Sports	113	135	685	2.54%
Real Estate	114	122	252	2.30%

mains, the aggregated number of sensors accessed and the aggregated number of feasible attacks for each category respectively. The last column shows the average access requests for sensors across websites that access at least one mobile sensor. We observe that domains that have a higher request access sensor rate fall into 10 major categories. Domains that fall into these categories typically show a lot of advertisements as well as retargeted ads [76], since users with that kind of browsing history appear to be heavily targeted by advertisers [239]. Based on the plethora of techniques that can be used for the device/sensor fingerprinting attack (see Table 4.1), we believe that mobile sensors can be used as another channel for tracking users even across sessions. We also observe that the three categories with the highest aggregate number of accessed sensors and attacks are Business, Online Shopping and Entertainment; these categories typically generate more ad revenue than other categories [41]. Since significant effort and deliberate design dictate the rules of digital advertising, the fact that these categories have the highest aggregate numbers of accessed sensors is not coincidental. Indeed ads can influence how we perceive our surroundings, which is highly applicable in beauty products [259]. For instance, a prevalent concept spread by the media relates to how people perceive beauty and attractiveness [74]. Moreover Barford et al. [75] showed that users with a 'Beauty & Fitness' profile are highly

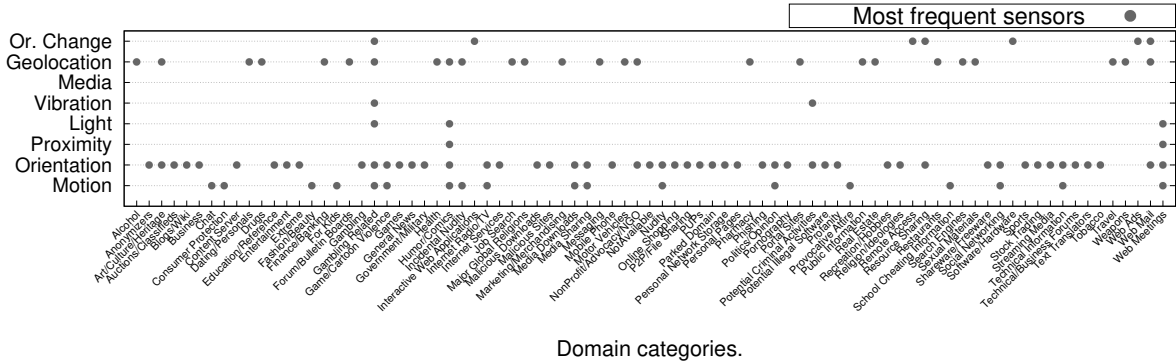


Figure 4.5: Most frequent accessed sensors for each domain category.

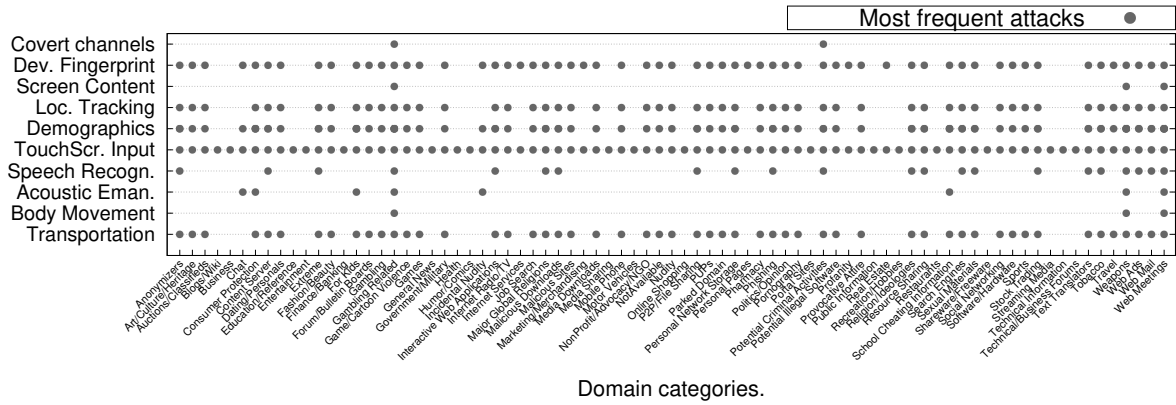


Figure 4.6: Most frequent (feasible) attacks for each domain category.

targeted by shopping-related ads.

Figure 4.5 and Figure 4.6 depict the most frequently accessed sensors and the most frequently feasible attacks for each domain category. In Figure 4.5 we observe that the majority of the categories will more often access three specific sensors: the motion, orientation, and device location sensor. We emphasize that the motion and the orientation sensors do not require any permission from the operating system and when used alone lead to 9 and 8 different attacks respectively. Moreover when these three sensors are combined they can result in 12 different attacks, including inferring the touchscreen input, device/sensor fingerprinting, and location tracking. Figure 4.6 shows that these three are the most frequent attacks across domain categories, along with the inference of demographic information. Interestingly, every category in our dataset, including those that the McAfee service could not classify (i.e., NotAvailable), can infer the user’s input across all categories. We argue that the information gained from this attack is extremely sensitive and can lead to more severe attacks.

Domain/Category popularity. To explore whether there is a correlation between the popularity of a given domain category and the number of sensors these domains tend to

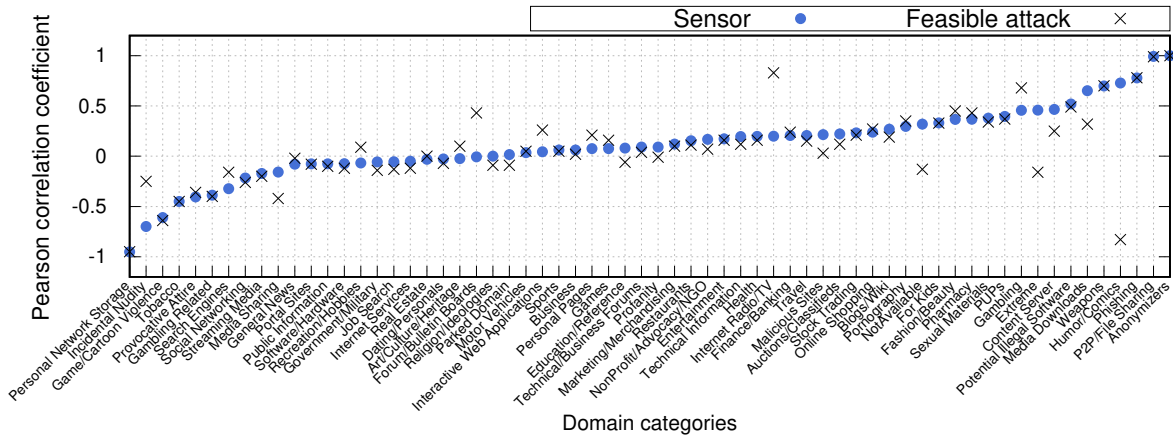


Figure 4.7: Pearson's coefficient between the Alexa rank for every website in every domain category, correlated to the number of sensors accessed and the number of attacks that are feasible with the data they collect.

access, we calculate Pearson's correlation coefficient. Figure 4.7 shows Pearson's correlation coefficient between the Alexa rank for every website in every domain category with the number of sensors accessed and with the number of attacks that are feasible with the data they collect. A score of 1 denotes perfect linear correlation between two variables, 0 denotes no correlation, and -1 shows total negative correlation. A positive correlation indicates that both variables increase or decrease together, while negative correlation indicates that as one variable increases, so the other decreases, and vice versa. As can be seen in Figure 4.7 the domain categories that have a positive value indicate that as the Alexa ranking is dropping so does the number of sensors accessed by websites in this category. The opposite is also true – more popular websites are more likely to access more mobile sensors. This is consistent across different popular categories such as Online Shopping, Finance/Banking, Entertainment, Pornography and Malicious Sites. Websites in these categories have a smaller Alexa ranking, which means that they are more popular and, as we discuss later on, these categories request a higher average access rate for sensors compared to other categories. Since the number of sensors accessed affects the number of feasible attacks, we deduce that *popular websites and domain categories are prone to access more mobile-specific sensors and such information enables a variety of techniques that can be used for a plethora of attacks*. We also observe that for some categories the r value is zero, indicating that for these categories there is no correlation between the number of sensors accessed and their popularity. For websites with a negative value we observe that as one variable increases (Alexa ranking), so the other decreases (sensors accessed). This is potentially due to these categories containing very few websites as shown in Table 4.7.

Banking sites. While device fingerprinting allows third parties to track users across the Web [194], fingerprints can be used as an additional factor for authentication [52]. As such,

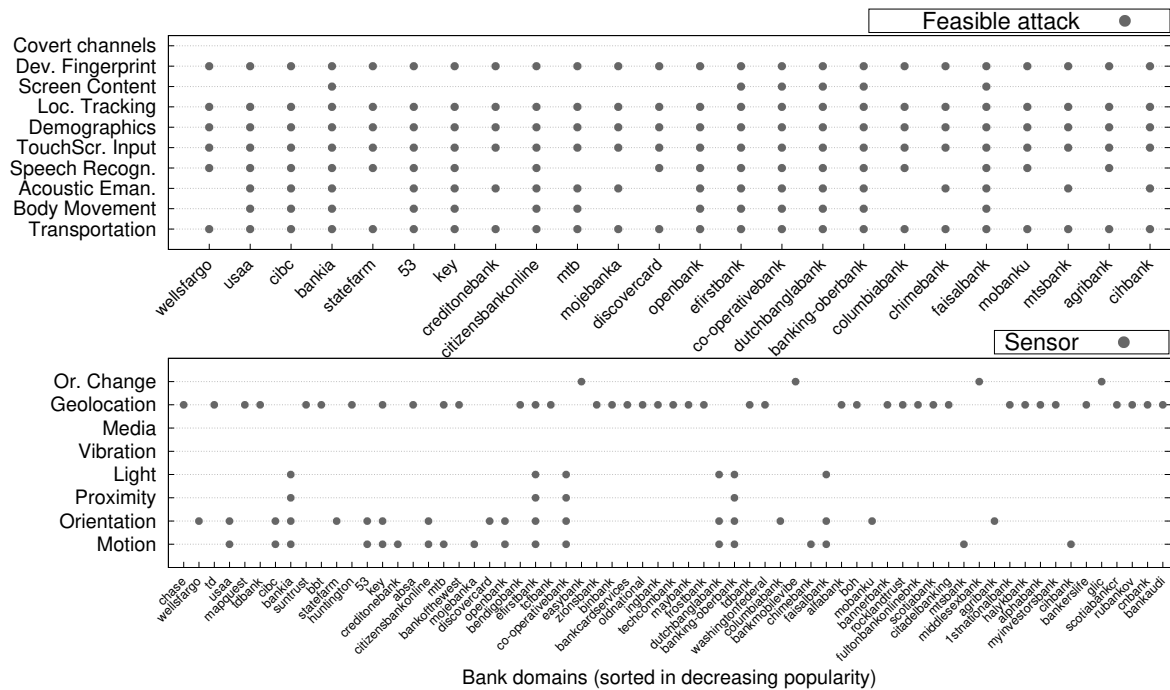


Figure 4.8: Breakdown of sensors accessed and corresponding attacks that could be deployed by banking domains.

banking websites are well-suited for deploying such a security mechanism [195] due to the significant implications of compromised accounts. As details of such practices are not typically disclosed, we further explore the prevalence of sensor-based information access across e-banking domains. We compiled a list of bank domains using online resources [4, 32] and cross-referenced it with our dataset.

As can be seen in Figure 4.8, we identified 65 banking domains that request access to data from at least one mobile sensor. Overall, banking domains request access for 1.38 sensors on average, which is higher than the average of 1.17 in other domains, indicating that banking websites are more likely to leverage the HTML5 WebAPI for accessing sensor data. We find that 24 of the bank domains obtain access to the sensor data necessary to conduct at least one of the attacks included in our taxonomy. Interestingly, all of those banks collect the sensor data leveraged in prior work for device fingerprinting, while 40 banks request access to the user’s geolocation which can also be used for enhancing the authentication process [52]. Furthermore, we also find that efirstbank.com actually requests access to more sensors than any other domain in our entire dataset. Overall, while accessing sensor data could be motivated by enhancing the authentication process, this practice raises privacy concerns as argued by privacy advocates [178].

Adult Content. Figure 4.9 shows the number of sensors accessed by websites that are classified by McAfee as “pornography”. We found that 136 domains access at least one mo-

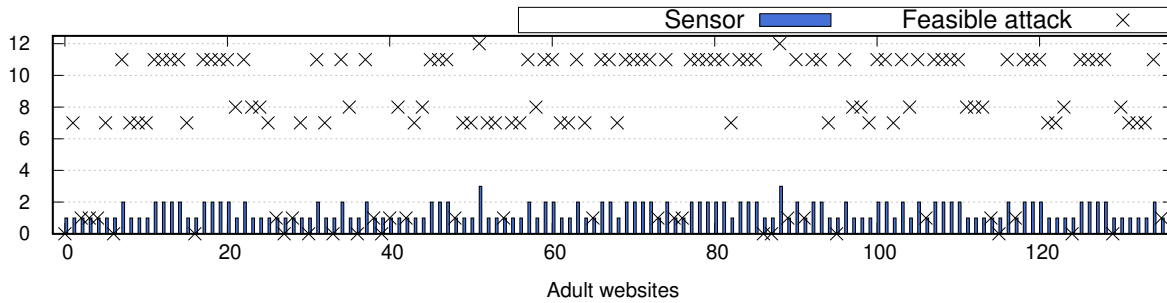


Figure 4.9: Breakdown of sensors accessed and corresponding attacks that could be deployed by domains with adult content.

mobile sensor. While the majority of them accesses one or two mobile sensors, which can result in 11 different attacks, in aggregate these domains access six out of the eight available sensors (orientation, position, motion, orientation_change, light, vibration). Interestingly we found that 73 domains are capable of inferring the user’s age and mood and 87 are capable of inferring the user’s sex. Such information can be of great value for this specific category of websites since it can be used to recommend additional content, increasing the duration of users’ sessions while showing them targeted advertisements (which leads to increased revenue). The information gained from mobile sensors should not be ignored within this context, especially since third-party analytics and advertising services have the ability to track users across and outside the adult web [262].

Malicious domains. Even though our system checked Google’s SafeBrowsing API before visiting a domain, it is possible that visited domains could be flagged as malicious later on, or by different blacklists. As such, we submitted all the domains that issued WebAPI requests to VirusTotal. Figure 4.10 presents the websites flagged as malicious (sorted by their rank), the number of accessed sensors per website and feasible attacks. Out of those, 149 domains were flagged by one AV engine and 17 domains were flagged by two. We can see that higher ranking malicious domains are more likely to access more sensors which results in a higher number of feasible attacks. We found 11 websites being flagged by at least 3 AV engines. The label on top of the bars in Figure 4.10 (bottom) represents the number of AV engines identifying these domains as malicious or suspicious. Finally, we found two websites,¹ namely *goggle.com* and *yotube.com*, that are flagged by eight AV engines as malicious. Apart from likely examples of typosquatting [181, 268], these websites requested access to sensor data that could be used to perform one and eleven different attacks respectively.

Country code top-level domain. To get a better understanding of the target audience of the domains that access mobile sensors we plot the websites that access at least one

¹The VirusTotal community also confirms that these websites were recently used for malicious purposes, as discussed here [34] and here [35].

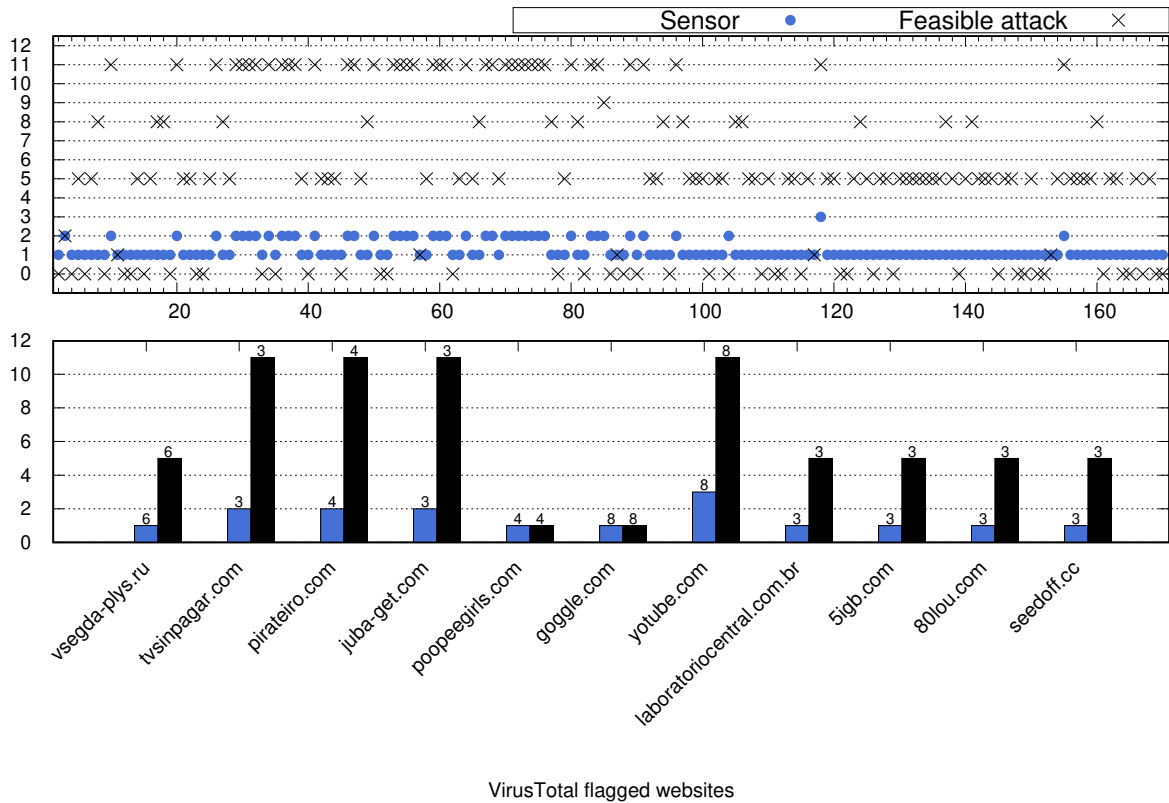


Figure 4.10: Number of accessed sensors and feasible attacks for websites flagged as malicious (top). Websites that were flagged by at least three AV engines (bottom) – the number on the bars shows how many AV engines flagged the domain.

mobile specific WebAPI based on the country code top-level domain. Figure 4.11 and Figure 4.12 show the aggregate number of sensors accessed and the aggregate number of feasible attacks. We observe that domains with a code top-level domain from Ukraine, Russia, China, Italy, Canada, Germany, India and Brazil tend to access more sensors than domains from other countries and also have a higher number of aggregate attacks. For example, websites from Ukraine, Russia and China have the ability to perform 909, 795 and 423 attacks respectively. Finally, we observe that some of the countries with the highest aggregate number of feasible attacks are among those that spend more money on digital advertising according to reports (e.g., [44, 45]), indicating that data captured from mobile sensors (e.g., demographics) could potentially be used for enhancing the efficiency of digital advertising. While analyzing how mobile sensor data is leveraged by the online ad ecosystem is out of the scope of this work, we consider an in-depth exploration of this phenomenon an interesting future direction.

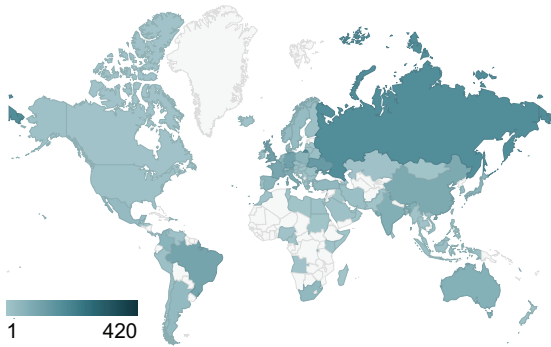


Figure 4.11: Aggregate number of sensors accessed by websites, classified by their country code top-level domain.

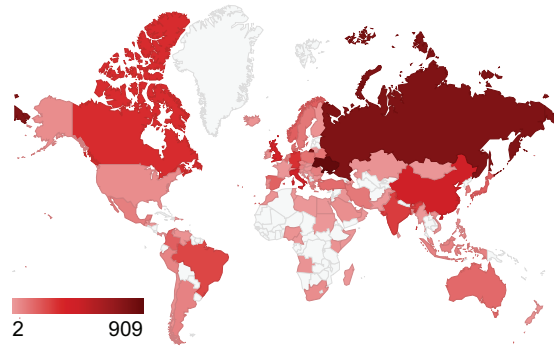


Figure 4.12: Aggregate number of feasible attacks for websites, classified by their country code top-level domain.

4.3.2 WebAPI Request Origin

Next we explore where the WebAPI requests originate from. Apart from exploring whether the request is first party (i.e., issued from a script hosted on the domain being processed) or third-party (i.e., issued from an external source but inside the original domain's page), our system also logs whether the request originates from an iframe. It is important to note that different browsers implement different policies regarding which sensors can be accessed for these three different types of origin. As such, we present statistics for all the websites that requested access, even if those requests were blocked by Firefox during our experiments.

Iframes. Our system collects all the calls executed by every element of a website, including iframes. In every log we record the source domain name of the element that is accessing sensor information. By comparing the URL of the address bar and the URL in the logfiles, we can identify whether WebAPIs are accessed by the DOM or by an iframe. Our analysis shows that 991 websites out of 5,313 contain iframes that use WebAPIs to access mobile specific information. We analyzed all iframes from our experiments and found that specific iframes are found in different websites. The two most frequent domains injected inside iframes exist in 389 webpages (or 39.3% of pages with iframes collecting data) and are related to online media players.

External sources. Among the websites that issue API calls for mobile specific information we found 40 scripts from external domains (either as a third-party scripts or inside an iframe) that collect data from 2461 websites 46.3%. We manually analyzed these scripts and found that they offer services for media-players and advertisements and they collect information about the orientation and motion of the device. In Table 4.5 we list the domains that appear in more than 50 websites and collect data from sensors. The first column is the origin of the script being executed. The second and third column show

how many websites and iframes host this script while the fourth column is the domain of the page inside the iframe. Given that these third-party domains are used in 35.89% of websites that access sensor data, we classified them based on the type of service they provide using *Cyren*². The last two columns show which sensors the script accessed and their corresponding attacks. We observe that most of these domains call the motion and orientation WebAPIs which enable a plethora of attacks. Moreover, domains classified as search engines and ad-networks gain access to characteristics that can track users across the web.

From Table 4.5 we can see that the domain *api.b2c.com* enables 12 different attacks. After investigating this domain³ through VirusTotal [33] we found that scripts served from this domain and Android apps that communicate with it are classified as intrusive adware and even malware by some antivirus vendors. Another domain, *c.adsco.re*, is flagged as malware by Cyren, even though it is not considered malicious by the Google SafeBrowsing API. We manually analyzed the content of the script that retrieves the data and found that apart from retrieving information about the Motion and the Orientation sensors it also exhibits behavior which is a strong indicator of device fingerprinting, such as creating and manipulating canvas elements [183] and reading different Navigator, Screen, Storage and Window properties. Interestingly the *adsco.re* domain states that it is used for traffic validation by Adscore, a bot detection service. In total, these two domains which are considered malicious by certain security lists, were found on 5.4% of all the sensor-accessing domains logged by our system, which again raises concerns regarding browser policies that allow third party domains to access sensor data without explicit user permission.

Android internals. Our crawling system allows an end-to-end analysis of sensor data access. Apart from providing high call-detection fidelity, since we can match requests logged by our injected JavaScript to actions at the operating system level, it also revealed sub-optimal browser behavior. We found that while Firefox prevents iframes from accessing sensor data, in practice Firefox simply “omits” returning the sensor data instead of blocking (i.e., ignoring) the actual request. Specifically, Firefox allows iframes to create event listeners, which then trigger the necessary WebAPI calls which then trigger the corresponding Android-level processing and permission checks for obtaining the sensor data; the data is then returned to the browser but not provided to the iframe.

Android WebView is based on the Chromium project and allows mobile apps to access and display web content. WebView usage is extremely widespread, found in 85% of the apps in the official Google Play Store in 2015 [184]. Due to its prevalence across Android apps, we also tested two popular WebView-based browsers, namely UC, and WebView (info.android1.webview), along with Facebook and Messenger, and found that they allow iframes to obtain data about motion and orientation. As such, even if users use Fire-

²<https://www.cyren.com/security-center/url-category-check>

³<https://www.virustotal.com/#/domain/api.b2c.com>

Table 4.5: Third-party scripts accessing mobile-specific WebAPI calls.

Script origin	#Sites	#iframes	iframe domain	Classification	Sensors	AttackID (see Table 4.3)
f.vimeocdn.com	275	275	player.vimeo.com	Streaming	O	1, 2, 3, 5, 6, 9, 11, 13
fast.wistia.com	467	3	fast.wistia.com	Technology	OC	-
fast.wistia.net	125	115	fast.wistia.net		OC	-
c.adsco.re	211	-	-	Malware	M,O	1 - 6, 8, 9, 11, 12, 13
g.alicdn.com	170	65	wanwang.aliyun.com m.aliyun.com	Shopping General	O,G	1, 2, 3, 5, 6, 9, 11, 13
aeu.alicdn.com	127	83	mbest.aliexpress.com	General	O	1, 2, 3, 5, 6, 9, 11, 13
api.b2c.com	76	-	-	General	M,O,PL	1 - 9, 11, 12, 13
cdn.admixer.net	169	-	-	Ads	M	1 - 4, 6, 8, 11, 12, 13
static.yieldmo.com	107	-	-		O	1, 2, 3, 5, 6, 9, 11, 13
secure-ds.serving-sys.com	51	35	googleads.g.doubleclick.net tpc.googlesyndication.com	Ads	M	1 - 4, 6, 8, 11, 12, 13
dlsnbr.baidu.com	77	69	pos.baidu.com	Search Engine	M	1 - 4, 6, 8, 11, 12, 13
client.perimeterx.net	73	-	-	Technology	M	1 - 4, 6, 8, 11, 12, 13

M: motion, G: geolocation, P: proximity, O: orientation, L: light, OC: orientation_change

fox or Chrome for web browsing, which currently block iframes from accessing any sensor data, opening a website within such popular apps that use WebView would expose them to attacks.

4.3.3 Transience of Web Measurements

Scheitle et al. [225] recently found that there is significant fluctuation in the websites contained in ranking lists used by academic studies, with Alexa being the most volatile list. As a result, similar measurement experiments that use an Alexa list from a different date could result in a significantly different view of the web ecosystem. To quantify and frame this effect within the dataset we have collected, we compare to the recently released dataset⁴ by Das et al. [96] which was part of their concurrent study on mobile sensor fingerprinting. While their collection set up was different (they used a modified version of OpenWPM as opposed to actual mobile devices) they also logged mobile sensor APIs used by popular websites. When comparing the domains that accessed mobile-specific WebAPI calls during our experiments to those in their dataset, we find only 403 overlapping domains – 7.9% of our detected websites. However, our system detected WebAPI calls in 2,252 domains that are in their two US-based datasets but with no calls logged during their experiments. Given that both of our experiments were conducted at similar times, including some overlap in May 2018, and used Alexa’s list (our version is from 03/24/2018 while their version is from 05/12/2018), this is a surprising result. As such, the two datasets together provide a more extensive coverage of the websites that use WebAPIs to access mobile sensors in the wild (we provide a detailed comparison to their study in Chapter 6).

Another important dimension that needs to be considered is that the modern web is

⁴<https://databank.illinois.edu/datasets/IDB-9213932#>

Table 4.6: Domains exhibiting differences in the WebAPI calls reported by our system and Das et al. [96].

Website	Our Dataset	[96]	Website	Our Dataset	[96]
allrecipes.com	O	L, M, O, P	britishairways.com	OC, O	M, O
99designs.com	M, OC	M	payback.de	M, P	P
gilt.com	M	M, O	newsela.com	O, P	P
joomshaper.com	O	M, O	udnfunlife.com	O	M, O
beefree.io	O	M, O	ceros.com	O	M, O
360cities.net	M, OC	M	99designs.co.uk	M, OC	M
skyscnr.com	M	O	zerator.com	O	M, O

highly dynamic and websites often introduce new functionality or may even remove existing functionality. To further explore how a view of the web can change through time, we compare the actual WebAPI calls reported for those 403 overlapping domains. While we find that for the vast majority (91.8%) of domains both datasets report the same calls across the two datasets, there are differences for 33 websites. In more detail, for those domains our system logged a total of 74 WebAPI calls, while the datasets from [96] contain 62 calls. This difference is partially due to that study targeting a *subset* of the calls that our study explores. However, in Table 4.6 we include the remaining domains where the two datasets report different sensor data being requested, which correspond to $\sim 3.47\%$ of the domains detected by both systems. While that number is not very large, it is non-negligible and highlights the dynamic and ever-evolving nature of the web.

A notable example is *allrecipes.com* which in our experiments only obtains the orientation data, while Das et al. reported that it accessed four different sensors. To further investigate this issue, we processed *allrecipes.com* again (10/31/2018) and verified our original findings. Motivated by prior work [158] that leveraged the Internet Archive for obtaining a retrospective view of web tracking, we processed one stored snapshot for each day of the crawling period of that specific dataset as reported by the authors (5/17/2018 - 5/21/2018) using a US-based IP address as well. Again we only identified requests for the device’s orientation. Subsequently, we identified the third-party JavaScript file (originating from *api.b2c.com*) that issued those requests in their dataset, and obtained a snapshot of it from the Internet Archive from 05/15/2018. After de-obfuscating it we verified that it indeed issues those requests.

4.3.4 Analysis Summary

In our analysis we provide a comprehensive exploration of the mobile sensors that websites access through the use of mobile HTML5 WebAPI calls, and analyze how this data can be used by websites in order to exfiltrate personal information about the user. To that end we have created a taxonomy of sensor-based attacks from prior studies and we present an

analysis by framing our collected data within that taxonomy. Our analysis shows that attacks that were previously limited to mobile apps can now migrate to the mobile web, and access to these sensors can lead to a plethora of different attacks such as capturing the user's input, identifying personal information and interests, as well as tracking the user across the web. We subsequently perform an in-depth analysis by classifying these websites into different categories based on the content they provide and analyze our dataset from multiple viewpoints. We find that popular websites and popular categories tend to access more mobile sensors, consequently leading to the feasibility of more sensor-based attacks. Furthermore, third-party scripts embedded inside webpages are also able to capture sensor information, thus creating a larger attack surface. Compared to similar studies [96], our study focuses on every mobile sensor-based WebAPI call and a direct comparison of the results shows that combining and comparing these datasets highlights the transience of the web ecosystem in practice. To shed more light on this phenomenon and to further facilitate research on the security and privacy risks that users face due to mobile sensor data being accessible to websites, we have made our dataset publicly available. Based on our results we argue that any information gained from sensors poses a risk for users and more effective access control policies should be enforced.

Table 4.7: Domain classification sorted in descending order based on the average request access for sensors across websites accessing at least one mobile sensor.

Label	# Domains	# Sensors	# Attacks	% Sensors/Total Domains
Business	662	743	2246	13.98%
Online Shopping	427	539	2316	10.14%
Marketing/Merchandising	417	459	1380	8.64%
Entertainment	348	439	2118	8.26%
Travel	322	392	1492	7.38%
General News	327	385	1640	7.25%
Education/Reference	293	321	1380	6.04%
Internet Services	301	318	1169	5.99%
Finance/Banking	239	307	955	5.78%
Fashion/Beauty	169	249	1218	4.69%
Blogs/Wiki	201	236	1272	4.44%
Public Information	201	226	388	4.25%
Software/Hardware	200	214	569	4.03%
Pornography	136	196	983	3.69%
Potential Illegal Software	117	181	990	3.41%
Health	128	170	480	3.20%
Games	115	143	865	2.69%
Restaurants	126	139	192	2.62%

Continued on next page

Table 4.7 – continued from previous page

Label	# Domains	# Sensors	# Attacks	% Sensors/Total Domains
Sports	113	135	685	2.54%
Real Estate	114	122	252	2.30%
Motor Vehicles	105	111	252	2.09%
Government/Military	64	96	364	1.81%
Portal Sites	81	88	304	1.66%
Recreation/Hobbies	63	71	300	1.34%
Forum/Bulletin Boards	64	68	240	1.28%
Job Search	64	68	164	1.28%
NonProfit/Advocacy/NGO	46	58	188	1.09%
Streaming Media	33	43	210	0.81%
Technical/Business Forums	34	43	204	0.81%
NotAvailable	35	39	222	0.73%
Auctions/Classifieds	28	31	150	0.58%
PUPs	26	30	176	0.56%
Gambling	15	23	97	0.43%
Parked Domain	19	23	85	0.43%
Pharmacy	19	23	71	0.43%
Search Engines	18	21	46	0.40%
Dating/Personals	17	20	70	0.38%
Media Sharing	13	17	70	0.32%
Interactive Web Applications	15	16	46	0.30%
Religion/Ideologies	13	16	65	0.30%
Provocative Attire	12	15	80	0.28%
Social Networking	13	15	82	0.28%
Art/Culture/Heritage	13	14	55	0.26%
Content Server	12	14	51	0.26%
Stock Trading	11	14	54	0.26%
Technical Information	12	13	62	0.24%
Personal Pages	10	12	77	0.23%
Internet Radio/TV	8	9	56	0.17%
Sexual Materials	8	9	18	0.17%
Weapons	7	9	12	0.17%
Major Global Religions	8	8	21	0.15%
Mobile Phone	8	8	44	0.15%
Incidental Nudity	6	7	24	0.13%
Malicious Sites	6	7	47	0.13%
Shareware/Freeware	7	7	39	0.13%
Alcohol	6	6	0	0.11%
Gambling Related	4	6	13	0.11%
Game/Cartoon Violence	4	6	37	0.11%
Media Downloads	5	6	27	0.11%

Continued on next page

Table 4.7 – continued from previous page

Label	# Domains	# Sensors	# Attacks	% Sensors/Total Domains
P2P/File Sharing	4	6	36	0.11%
Politics/Opinion	6	6	45	0.11%
Tobacco	5	6	39	0.11%
Chat	5	5	22	0.09%
Humor/Comics	3	5	17	0.09%
Nudity	5	5	30	0.09%
Phishing	3	5	25	0.09%
Profanity	4	5	30	0.09%
School Cheating Information	5	5	40	0.09%
Drugs	4	4	0	0.08%
Extreme	3	4	19	0.08%
For Kids	3	4	27	0.08%
Personal Network Storage	3	4	25	0.08%
Web Ads	4	4	7	0.08%
Web Meetings	1	4	12	0.08%
Anonymizers	2	3	18	0.06%
Web Mail	3	3	7	0.06%
Potential Criminal Activities	2	2	8	0.04%
Resource Sharing	2	2	7	0.04%
Consumer Protection	1	1	8	0.02%
Malicious Downloads	1	1	7	0.02%
Messaging	1	1	0	0.02%
Remote Access	1	1	0	0.02%
Text Translators	1	1	7	0.02%

4.4 Discussion

In this section we provide guidelines for establishing policies and defining the functionality that should be supported and standardized across browsers to better protect users.

Even though the research community has proposed access control mechanisms that regulate which application can access mobile sensors [72] or even provide apps with fake values [40], we believe that these approaches are not sufficient. Specifically, third-parties are able to circumvent such mechanisms either by being embedded in the application's source code or in certain cases by being part of the webpage in the form of third-party JavaScript. For instance, research has shown that browsers that rely on WebView do not enforce correct policies for the HTML5 Geolocation API [149]. Moreover, any approach that relies on completely disabling JavaScript or blocking all scripts will inadvertently lead

to poor usability and user experience. To identify such limitations and shortcomings, we performed an empirical analysis of the sensor access control options currently offered by major mobile browsers by examining the security options they provide through their user interface for different use cases. We also experimentally inferred the access control policies in place for different scenarios that pose additional threats to users (e.g., the ability of third-part scripts to access sensor data). Table 4.8, summarizes our findings. Based on browsers' existing designs, and the limitations we have identified we argue that there is dire need for a standardized, fine-grained universal mechanism that allows users to control access to all types of mobile sensor data. Based on the severe implications of information being gained by these sensors, we provide guidelines and propose a concrete list of access control strategies for different scenarios, that should be adopted across browsers:

- **Universal revocation.** The browser should provide user's with the ability to universally decline access to *all* mobile sensors, regardless of being protected by a specific Android permission. Currently, as shown in Table 4.8, for the motion sensors (accelerometer and gyroscope) only Google Chrome and Brave (which is based on Chromium) have implemented a feature allowing users to navigate to the browser's site settings under the advanced options in order to disable access to them. Unfortunately, even for these browsers users can not control how access is granted to other mobile sensors (e.g., Ambient Light). Moreover, users do not have the option to enable access for a specific sensor or a specific website (i.e., a whitelist-based approach where a user can explicitly allow a specific website to always access a given sensor).
- **Explicit permission requests - Permissions list.** The browser should enforce its own run-time permission system for *all* mobile sensors. This feature is already being used for the GPS sensor; whenever a website requires access to the device's location, the user can grant or deny the request. This feature not only informs users about the functionality of the website but also allows for a fine-grained access control mechanism where users have control over their data. Since explicitly asking the user for permission to access the sensor at each request can lead to poor usability and user experience, the browser should maintain a list of domains that the user has granted or denied access to, similar to Android's dangerous permissions for apps. This will allow users to revoke access to a specific sensor for a specific domain at any time. A permission list has been implemented in Google Chrome for regulating access to different elements of a specific website (Sound, Mic, Location, etc.) and can be found in the `Site settings` under the permission category. Unfortunately even in the latest version of the mobile Google Chrome (version 79), the permission list does not support or include mobile sensors. Interestingly the desktop version of Google Chrome includes motion sensors in the permission list found under the same `Site settings` category. Since motion sensors exists mostly in mobile devices it is likely that this feature may soon be implemented in the mobile version of Google Chrome. Due to the severity of the attacks enabled by mobile sensors we argue

that whenever a website requests access to any of the available mobile sensors the user should be informed with an explicit permission request.

- **Origin differentiation.** As shown by our experiments, the most commonly feasible sensor-based attack is the inference of the user's touch input which allows a malicious domain to exfiltrate extremely sensitive information, such as the user's credit card number. The attack surface grows considerably since third-party scripts can create a hidden covert channel and exfiltrate sensitive data even if the user is browsing a legitimate webpage. Even though only one of the browsers tested (UC) allows iframes to gain access to motion sensors, we found that they *all* allow third-party scripts to obtain data about the device's motion and orientation. We believe that the browser should inform the user whether the sensor request originates from the first party domain or from a script hosted on a third-party domain. This information about the origin will enable users to make better decisions about whether to grant permission or not. Indeed, prior work [109] highlighted the need for providing Android API permissions based on the origin of the request – a similar idea can be applied at the application layer for differentiating access control policies based on a script's origin. Moreover, studies have shown that users are more likely to deny a permission request when a detailed description of the data that will be accessed is given [112]. Understanding the purpose of why and how sensitive resources are used can have a major impact on their feelings and trust decisions [162].
- **Private browsing.** Private browsing was created as a privacy feature where the browser creates a temporary session isolated from the browser's main session and user data. In the current web ecosystem where ad platforms and trackers collect an abundance of user information that is added to user profiles so as to improve recommendations, private browsing modes allow users to browse the web without exposing the common identifiers (i.e., cookies) that are sent by browsers during normal operation. Unfortunately, data from smartphone sensors can be used to accurately fingerprint a mobile device and, as an extension, the user. Therefore we argue that browsers should deny access to all mobile sensors in private browsing mode by default, unless users explicitly change their settings to allow that. Through empirical analysis of the most popular mobile browsers, (including Google Chrome, Mozilla Firefox and Microsoft Edge) we have found that they *all* allow access to sensors while in private browsing mode. As shown in Table 4.8, even privacy-oriented browsers such as Brave and DuckDuckGo neglect to block sensor access in private browsing mode.

Table 4.8: Access control currently enforced for *motion sensors* in popular mobile browsers. The "Universal revocation", "Per-site revocation" and "Origin differentiation" columns indicate whether the browser supports this feature. Columns marked with an asterisk (*) indicate whether the browser allows access to motion sensors in those scenarios.

Browser	Version	Universal revo/tion	Per-site revo/tion	Origin differ/tion	3rd-party scripts*	iframes*	Private browsing*
Chrome	79.0.39	✓	✗	✗	✓	✗	✓
Firefox	68.4.2	✗	✗	✗	✓	✗	✓
Edge	44.11.2	✗	✗	✗	✓	✗	✓
Brave	1.5.3	✓	✗	✗	✓	✗	✓
Opera Mini	46.0.22	✗	✗	✗	✓	✗	✓
UC Browser	v12.12.6	✗	✗	✗	✓	✓	✓
DuckDuckGo	5.41.0	✗	✗	✗	✓	✗	✓
Dolphin	v12.1.5	✗	✗	✗	✓	✗	✓

Chapter 5

Misusing Mobile Sensors for Stealthy Data Exfiltration

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 [256]. 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 [55, 56, 80, 97–100, 106, 117, 134, 142, 143, 177, 187, 216, 220, 289, 291], inferring physical activities [134, 140, 174, 187, 216] and in more severe scenarios inferring users' touch screen input [83, 138, 174, 198, 243, 280]. Das et al. [96] also demonstrated that web scripts accessing mobile sensors allow for stateless tracking on the mobile web, while Marcantoni et al. [170] 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 [46], 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 [205], 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 chapter 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 [50] 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 [119], in certain cases (i.e., [105, 128]) 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.

5.1 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., [57, 98, 123, 172, 177, 187, 210, 271, 282, 284]) 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 [174], without requiring any permission from the operating system or the user. Researchers previously presented a taxonomy of these sensor-based attacks [107, 170], 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 [67, 83, 84, 138, 174, 179, 198, 207, 280]. 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` [62] `DeviceOrientationEvent` [222] APIs, while the `DeviceMotionEvent.rotationRate` [182] API gives access to the gyroscope sensor. Moreover, the Generic Sensor API [263] 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 [96, 107] 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 [127]. 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 mobile 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.

5.2 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 telecommu-

nication 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 UIC’s Institutional Review Board (IRB). 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.

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 pertinent¹ 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 appro-

¹The 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.

priate 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.

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.

5.3 This Sneaky Piggy Went to the Android Ad Market

In this section 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.

5.3.1 Threat Model

We demonstrate a new attack vector that abuses the mobile advertising ecosystem for delivering a mobile sensor-based attack which affects every Android device (71.93% of mobile users worldwide [21]). 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 [107], 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.

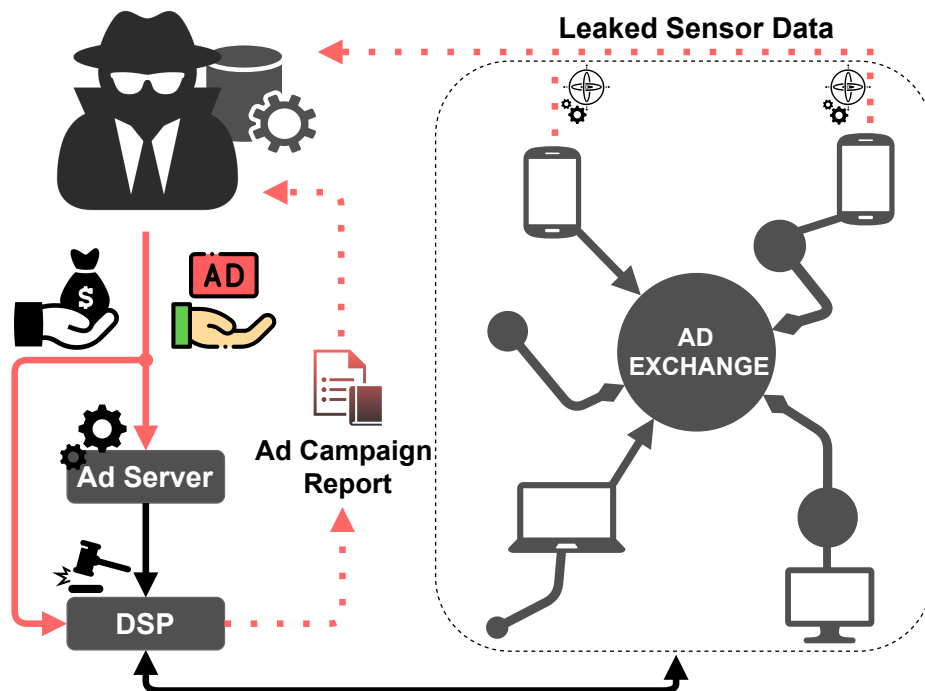


Figure 5.1: Overview of our attack vector. A malicious actor publishes an ad campaign that accesses mobile sensors, for delivering sophisticated and stealthy attacks.

Figure 5.1 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 [203], 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 §5.3.2.

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 5.1: 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			
<i>User Actions</i>	Device Lock	✓	✓	✓	✓
	UI Swipe	✓	✓	✓	✗ — ✓
	Swipe + Lock	✗	✗	✗	✗
	Force Stop	✗	✗	✗	✗

5.3.2 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 5.1 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 [103].

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 §2.5, 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 [245], 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 its 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 [227], 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 [128], 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 [105], 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 misled. 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" [228], 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 5.1, 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).

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.

5.4 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 5.2, 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 [224], 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

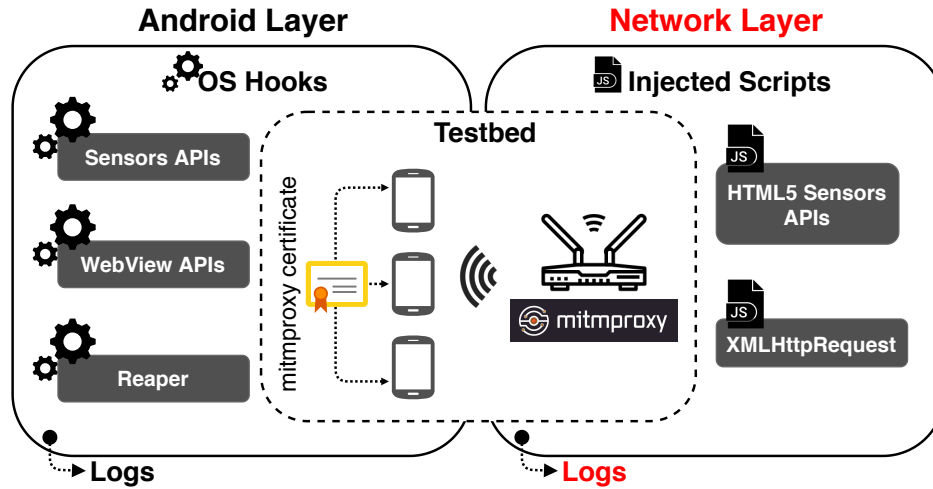


Figure 5.2: Overview of our framework’s infrastructure. The combined components of both layers provide an in-depth view of requests to access mobile sensors. Components in the Android layer (left) are responsible for monitoring system API calls, while components in the Network layer (right) monitor JavaScript calls and network traffic.

the `mitmproxy`’s root certificate that allows us to intercept HTTPS traffic. In Section 5.4.1, 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 [199], 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 Chromium and WebView APIs. Additionally, we leverage functionality from prior work [109] 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` [93] and injecting code for intercepting JavaScript calls. We used the `javascript-hooker` Node.js module [77] 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 [118]. We also monitor any calls of the `XMLHttpRequest` function, since in-app adver-

tisements 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 5.2 in Section 5.4.1, 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.

5.4.1 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., [51, 201, 219]).

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 [19]. As these techniques are common we refer the reader to appropriate

Table 5.2: 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(deviceorientation)	Fired at a regular interval, indicating the amount of physical force the device is receiving.
window.addEventListener(deviceorientation)	Fired when new data is available about the current orientation .
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(deviceambientlight)	Provides information from photo sensors 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 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.

online tutorials (e.g., [236]). 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 [209] 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 5.2 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 [118], as well as prior work on mobile sensors attacks and device tracking (e.g., [96, 107, 196]).

Table 5.3: 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

5.5 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.

5.5.1 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 [96, 107], we tried to download the corresponding mobile app if it exists in the official store. Overall, we downloaded 4,478 apps from Google Play using the Raccoon [27] 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 [109], our framework interacts with each application for five minutes using a breadth-first traversal strategy.

5.5.2 Intra-app Data Exfiltration

WebAPI Accesses. As can be seen in Table 5.3, 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 [196]. 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 our original threat model, we believe that such functions should be restricted since local storage can be used for re-identifying mobile devices [253].

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 5.3 shows the number of apps that contain Google’s interstitial ad placements based on their numbers of downloads. We observe that interstitial ad placements

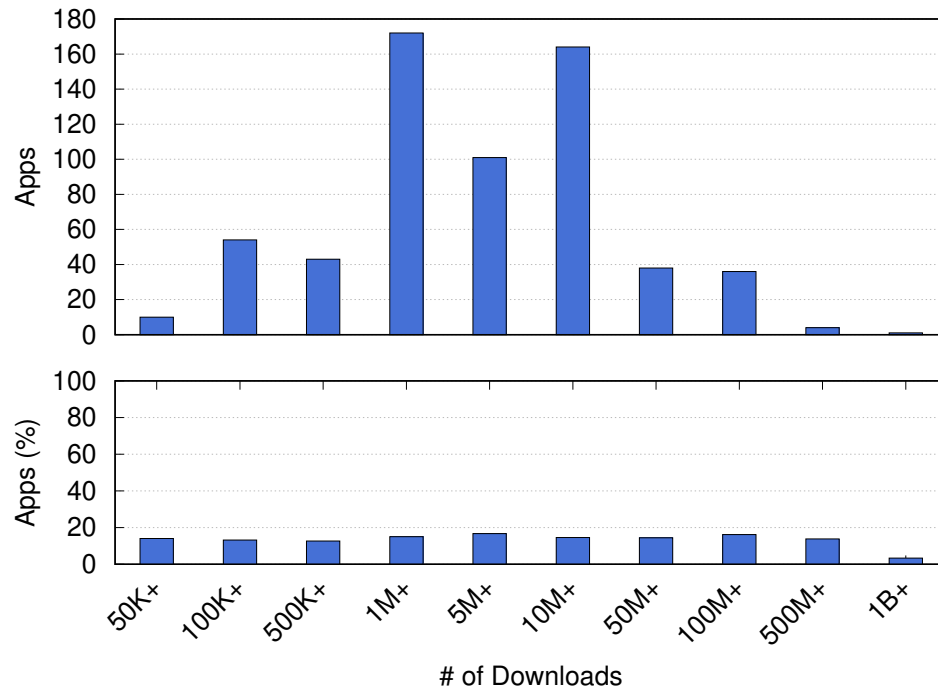


Figure 5.3: Number (top) and ratio (bottom) of apps with Google’s interstitial ad placements, per ranking bin.

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.

5.5.3 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 5.4 shows the 10 most popular apps that contain ads and hold this permission.

Table 5.4: 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	✓	✗	✓

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 5.5 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

Table 5.5: 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
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...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

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., [87, 94, 151]) and since ad libraries are mostly responsible for ad fraud activities [151], 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 [110], 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 5.6 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 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 5.6 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 [50] 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

Table 5.6: 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 & Orientation Events	Intra Vuln	Inter Vuln	CAM	MIC	GPS	Ads 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	✓	☐	✓	✗	✗	✓	✗

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.

5.6 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 [217]. 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.

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 [53] 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) [185] as the hidden layer activation func-

tion 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 [137]. 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) [90], 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 [244] 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 5.7. 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.

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 open-sourced our code to facilitate additional research, our

Table 5.7: 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%

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 [84]. In most studies [67, 84, 138, 174, 179] 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 [84]. Hodges et al. [138] 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. [179], 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.

5.7 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., [200, 219]) 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 [199]. Another mechanism for identifying leaks in Android apps is AGRIGENTO [92], 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` [270] 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. Below 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. Devel-

opers 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 [104]. 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.

5.8 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 below.

Element Coverage. Prior work [109] 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 [202]. 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 [203], 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 [203] 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.

Chapter 6

Related Work

6.1 Android Permissions

Android sandboxes apps by executing them in separate processes with distinct user IDs (UID) and assigning them private data directories on the filesystem. In order to achieve privilege separation Android enforces Permissions, i.e., privileges that an app is granted by the user. These permissions are assigned to the app's UID and are enforced at different points. Apps can directly interact with the kernel through system calls, such as editing files. Access control in the filesystem ensures that the apps' processes have the necessary permissions to issue particular syscalls. The filesystem access control consists of the traditional Linux Discretionary Access Control (DAC), and is complemented by SELinux and Mandatory Access Control (MAC). Apps can also interact through the Android API in a strictly controlled manner with highly privileged resources. Normally apps are prohibited to access those resources directly and access to these resources is given by system services. Apps that want to access the resources handled by a system service need to implement the API and perform the appropriate call. Whenever an app wants to access such information, a new thread of the appropriate service manager is created and this newly created thread calls the validation mechanism. During this process, Android Binder is responsible for passing messages between entities, using the `onTransact()` and `execTransact()` functions

Android permissions have changed throughout Android's life cycle. In the early versions of Android, users were presented with confusing blocks of information at installation time [145]. Following the introduction of the new permission system in Android 6, users can now accept or reject a permission request at run time, or revoke permissions at any time through the system's settings. However, recent work [112] demonstrated that users still do not fully grasp how permissions work and found that they are more likely to deny a permission request when given a detailed description of their personal information that will be accessed and uploaded (e.g., their actual phone number). Lin et al. [162] found that providing users with information on why a resource is being used can alleviate their privacy concerns, while in a different context Wang et al. [267] found that users perceive permissions differently when they are related to an application's core functionality.

6.1.1 Permission Analysis

The official Android SDK documentation does not provide a list of permission-protected functions which led the research community to statically analyze the Android framework in order to shed more light on the internals of the Android permission system. In [287] the authors conducted a study on the Android permission system, and highlighted the differences between API versions as well as possible issues that arise due to the constant evolution of the permission system. Stowaway [116] was the first project to statically map API calls to permissions (in Android 2.2) for detecting overprivileged applications. Their static analysis approach, feedback directed API fuzzing and unit testing, generates the required permissions for the framework API calls. PScout [66] statically created the permission mappings in four Android versions (2.2-4.0) by performing reachability analysis between API calls and permission checks. Wijesekera et al. [273] conducted a user study to understand how often apps require access to protected resources by instrumenting the Android platform. Kennedy et al. [146] explored the impact of statically removing permissions from apps in the older Android permission system. Wang et al. [265] employed text analysis and machine learning to infer how two specific permissions are used (ACCESS_FINE_LOCATION and READ_CONTACT_LIST) based on a manual labelling of 622 apps. They relied on the PScout mappings and reported an accuracy of 85% and 94% for the two permissions. AXPLORER [71] is one of the most recent studies in Android permission analysis and improves previous results and also publicly provides the permission mappings for API's 16 to 23 (excluding API 20). The authors identified the framework API methods that are exposed to applications as analysis entry points, and using forward control-flow slicing they mapped the framework entry points to the required permissions. AXPLORER also introduced the concept of permission locality and investigated which framework components enforce a particular permission. They showed that although framework services follow the principle of separation of duty, permissions can be checked at different and not closely related components. This indicates a violation of the separation of duty principle and complicates a comprehensive understanding of the permission enforcement. The permission locality is measured in terms of number of distinct classes that check a given permission. High permission locality means that a permission is checked at a single service, while a low permission locality means that a permission is checked at different services. Compared to PScout's results which includes only normal and dangerous permissions, AXPLORER additionally includes system and systemOrSignatures permissions and improves previous inconsistencies. Neither PScout nor AXPLORER conducted a native code analysis since this requires a dedicated analysis for the native code.

Even though static approaches do not have the coverage problem that dynamic analysis suffer from, they do have some limitations. First they assume that all permissions identified given a program dependence graph for a particular API are indeed required.

This is also true for AXPLORER's static analysis since it does not take into consideration the input values of API calls which may alter the behavior of the permission mechanism. As shown in [108] some functions may or may not be permission-protected based on the arguments provided. For example, the `getprovider()` function of the class `LocationManager` is permission protected when provided with the string `GPS PROVIDER` but does not need a permission for `KEY LOCATION CHANGED`. Second the permission enforcements are often conducted with other security features such as UID checks and User checks. To address these limitations, ARCADE [47] builds on top of previous results by statically analyzing the Android framework in a precise and path-sensitive API analysis. Their system identifies the public entry points of the framework system services and creates a control flow graph. This control flow graph is transformed into an access-control flow graph by removing nodes irrelevant to enforcing access control in order to reduce the complexity of the analysis. The protection map is constructed using a depth first traversal of the access-control flow graph starting from the framework entry point until the node where the permission access is granted. During this phase ARCADE extracts all the conditional expressions (e.g., AND, OR) of a specific path and represents them as a first-order logic formula.

6.2 Privacy Leakage

"Personally identifiable information (PII) is any data that could potentially identify a specific individual. Any information that can be used to distinguish one person from another and can be used for de-anonymizing anonymous data can be considered PII" [39].

An increasing body of work is about how personally identifiable information or other sensitive information is collected or leaked by Android applications and the integrated third-party libraries. In this section we present studies that show how third-party libraries surreptitiously exfiltrate personal data (Section 6.2.1), studies that identified the plethora of these libraries (Section 6.2.2), studies that measured privacy leakage while accessing the same service through the mobile app and a mobile browser (Section ??) and how mobile sensors can be used as another channel for data exfiltration (Section 6.2.4).

6.2.1 Third-party Libraries

Meng et al. [175], studied the privacy concerns that arise from in-app advertising. In contrast to web advertising where ads are displayed inside an `iframe` and are isolated from the hosting website, mobile advertisements run in the same process as the hosted app. Therefore app developers are able to infer personalised information about the user just by looking at their personalised advertisements. The authors, in order to understand the level of personalised data major ad libraries collect (e.g., Google AdMob), created 217 real user profiles and compared those profiles against the ads delivered to each user. Their re-

sults show that ad publishers can identify user demographic information up to 75% and parental status up to 66%, highlighting the need for additional protection against such privacy leakage. The information that can be leaked to app developers can be used to request ads from other higher paying ad networks, price discrimination or even be sold to other parties.

Son et al. [240] studied the isolation of different ad SDKs, and showed that the same-origin policy is not sufficient for protecting users' privacy from malicious ads. The same-origin policy prevents advertising code from reading the contents of cross-origin resources such as local files, but it does not prevent advertising code from embedding these files as image, audio, or video elements. By attempting to load a DOM element whose URI points to a local file, a malicious advertisement can identify if a local file with this name exists in the external device, even if it can't read its content. Using four different application scenarios, the authors showed that this technique can leak information such as the user's medication, the gender preference, the browsing history and the user's social graph.

In a similar study, Demetriou et al. [101] showed that ad libraries have the potential for increased data collection through side-channels and proposed a system that combines natural language processing and machine learning for discovering what information is exposed to advertising parties. Their framework, Pluto, analyzes the risk associated with an ad library using four attack channels: a) permissions granted to their host apps; b) reading files generated at runtime and stored in the host apps' protected storage; c) observing user input into their host apps; and d) unprotected APIs such as `getInstalledApplications()`.

Leontiadis et al. [157] conducted a large scale app analysis by extracting information from the apps' XML file and identified that many applications (both free and paid) request at least one permission that could reveal personal information. The authors argue that even though the Android app ecosystem is primarily driven by advertisements that rely on accurately profiling the user and showing him targeted advertisements, it is also important to control the private information flow and monitor greedy access to personal data by misbehaving apps and ads. They showed that current protection mechanisms that may block or alter the information with fake values are not sufficient since they can lead to a diminishment of the revenue stream for the developers of ad-supported applications. To this end they propose a market-aware privacy protection framework for achieving a balance between user privacy and ad revenue.

Seneviratne et al. [229] showed that even though paid apps are regulated by a different business model they also contain tracking libraries but tracking is happening to a lesser extent in paid apps. Their key findings denote that almost 60% of paid apps, contain trackers that leak sensitive information, compared to 85%-95% found for free apps.

Advertising libraries are not the only ones that have the potential to track users. In [165] the authors investigated what kind of data analytics libraries collect about users' in-app actions. They created Alde, an analytics libraries data explorer framework, that com-

bines static and dynamic analysis for uncovering privacy leaks from 8 analytics libraries in 300 apps. In the static analysis phase, Alde uses backward taint analysis on the app's smali code to find hardcoded information. During runtime, it hooks the libraries' tracking APIs and logs this information for further analysis. Their results show that analytics libraries not only leak information to app developers (i.e., users' email addresses, IP addresses and email subjects) but also leak personal information to different analytic companies. This is possible because analytics libs exists in many applications and these libraries are able to share the data they collect. In order to inform users about the privacy leakage from analytics libraries, the authors created an Xposed module (ALManager) that monitors the libs' tracking APIs and is able to block them or replace the information gained with empty values.

ReCon [219] is a cross-platform system build to reveal PII leaks over the network. It uses machine learning (Decision Trees) to identify flows that contain PII with 98.1% accuracy. Even though it requires ssl stripping for HTTPS flows and has to be deployed using VPN tunnels, the authors showed that compared to other similar tools (FlowDroid [64], Andrubis [163], AppAudit [278]) Recon is faster and identifies more PII. The PII leaks that Recon focuses on are Device Identifiers (ICCID, IMEI, IMSI, MAC address, Android ID, Android Advertiser ID, iOS IFA ID, Windows Phone Device ID), User Identifiers (name, gender, date of birth, e-mail address, mailing address, relationship status), Contact Information (phone numbers, address book information), Location and Credentials. Unfortunately one of the main limitations of Recon is PII's that have been obfuscated before entering the network.

Third-party libraries use different techniques to hide their suspicious activity from users as well as from researchers. Encryption and obfuscation are the most common and are used in order to make the source code of the library as well as the network generated harder to analyze. To this end, in [91] the authors proposed AGRIGENTO, a black-box differential analysis tool capable of identifying leaks even in the presence of obfuscation offering significant improvement over prior work. Their approach consists of two phases. First for any given app they execute it multiple times in an instrumented environment to collect raw network traces, and contextual information. This step allows to monitor common sources of non-determinism in network traffic such as random, timing, system and encrypted values. During the second phase they execute the app in the same instrumented environment with the only difference that they change one of the input sources of private information (e.g., IMEI) and compare the network flows from the previous step. Although, AGRIGENTO detect leaks that have been encrypted or obfuscated it cannot handle apps that use custom encryption or custom or native code for certificate checking.

TaintDroid [113] was amongst the first to perform dynamic taint analysis in the Dalvik Virtual machine for monitoring how applications access and manipulate users' personal data. Their study on 30 popular applications show that half of the studied applications

share location information with advertisement servers while some of these apps leak the device ID, the phone number and the SIM card serial number. Even though TaintDroid is extremely efficient in tracking taints it has to be paired with a dynamic testing approach that yields decent code coverage. PmDroid [122] used TaintDroid to track and block sensitive data obtained through certain permission protected calls from being sent to ad networks, but it obtains incomplete taint tracking coverage and relies on volatile domain information for identifying ad networks. VetDroid [286] extends the taint tracking logic of TaintDroid to monitor callbacks but suffers from the same coverage limitations. Since the adoption of the ahead-of-time ART compiler, new taint analysis techniques have emerged. TAINTART [252] presented an information flow tracking system integrated inside ART that can be used for detecting data leakage. ARTist [69] is a compiler-based app instrumentation framework that can be used for intra-app taint tracking, as well as dynamic permission enforcement by operating only at the application layer.

Wang et al. [265] employed text analysis and machine learning to infer how two specific permissions are used (`ACCESS_FINE_LOCATION` and `READ_CONTACT_LIST`) based on a manual labelling of 622 apps. They relied on the PScout mappings and reported an accuracy of 85% and 94% for the two permissions. To overcome the obstacle of obfuscated code, they recently incorporated a dynamic analysis aspect and conducted a study on 830 apps [266]. While their approach has similarities with Reaper it presents significant limitations. They rely on a modified version of TaintDroid in Android 4.3 and only perform stack inspection at sink points (e.g., the network). Since stack inspection at this layer does not provide much information about the purpose of the permission due to multithreading, they heavily modified Dalvik to also capture the stacktrace of the parent thread. Their system also induces a slowdown of up to 47% compared to stock Android, while relying on random fuzzy testing which is inherently limited. Reaper has negligible overhead and performs stack inspection at the access level; this allows to successfully monitor all PPCs for different Android versions, including those based on the ART compiler. Overall, while their study focuses on a different aspect of permissions, incorporating Reaper for their dynamic analysis would allow them to efficiently conduct a large scale study and achieve higher coverage without the drawbacks of their extensive OS modification.

6.2.2 Identifying Third-party Libraries

Older studies [130, 188] tried to identify third-party libraries by using the package structure or common identifiers. However, since this information can be easily modified, such techniques are not suitable for accurately detecting the presence of third-party libraries in apps. More recently, Li et al. [160] conducted a large scale analysis of 1.5 million apps from Google Play, in order to identify common Android libraries. Since common libraries exist in a large number of apps and are used without modifications, the authors clustered the

package names based on how frequently they occur inside the dataset. Even though their approach does not handle obfuscated code, they identified 1,353 third-party libraries and made their list publicly available. LibScout [70] bypassed the limitations of obfuscated code by using a variant of Merkle trees and performing profile-matching between known third-party libraries and the contents of the apk file being tested. This system is able to identify the version of the integrated third-party libraries and its package name. Since the results provided by LibScout are bound by the dataset they are trained with, it is possible for LibScout to miss some of the libraries that are integrated in the application.

In a similar project, LibD [161] statically analyzed the app's call graph and the relations between Java classes and methods in order to extract features which will be used in identifying potential third-party libraries. The extracted features are hashed and compared with features of other applications. Based on the number of appearances of these features in different apps, LibD is able to identify third-party libraries even when the app is obfuscated. In another recent study, Titze et al. [258] proposed Ordol, which applies plagiarism detection techniques for detecting a specific library and its version. Ordol measures the similarity of classes and methods between known libs and an app's code by using maximum weight bipartite matchings. As opposed to LibScout that relies heavily on the package structure, Ordol represents the objects of a library (such as the classes and methods) as vertices, and matches those vertices with the objects of the analyzed app. The similarity of the objects is calculated by the edge weights. Even though this approach is resilient against code inlining and unused code removal performed by obfuscators and optimizers, it still relies on a list of known third party libraries.

It is evident that significant research efforts have been applied towards identifying third-party libraries. Even though these systems have made significant progress and presented encouraging results, they still suffer from the inherent limitations of static analysis. Unless applications are analyzed dynamically, no system will be able to obtain a complete and accurate analysis of a third-party library's behavior. Missed behavior may range from innocuous actions to malicious functionality. A recent example of a malicious library masquerading as an advertising library is that of XavirAd [15]. This third-party library evades static analysis by leveraging dynamic code loading (DCL); specifically, it side loads malicious functionality by downloading a dex file from a remote server. After executing the downloaded code the library collects permission-protected PII and sends it over the network in an encrypted form.

6.2.3 Privacy Leakage between Apps and Mobile Browsers

Online services provide both a mobile-friendly website and a mobile application to their users and very often both choices are released for free. As mentioned earlier, developers usually gain revenue by integrating advertisements into their content and ad networks in

order to provide more personalized and effective advertisements they deploy pervasive user tracking. Since advertisement and tracking is employed in both websites and applications, raising this way significant privacy concerns, the research community studied the privacy leaks that occur when accessing the same service from the mobile application and its web counterpart.

In [159], Leung et al. conducted the first measurement study that analyzes 50 popular free online services in order to understand which is better with respect to user privacy. Using both Android and iOS devices they manually interacted with a given service using the mobile browser and the mobile app and captured network traces using Meddle, a VPN solution, and Mitmproxy. ReCon [219] as well as direct string matching on known PII was used to identify privacy leakage over the network flows. Even though both options leak personal information to advertising and analytics domains, the information leaked is different in each case. Websites tend to leak more demographic and location information while apps leak more device specific information. Their results indicate that in 40% of cases, websites leak more types of information than apps and there is no clear single answer as to which option (Web vs App) is better in terms of privacy. Therefore, the authors implemented an online service [88] aiming to recommend the users the best option for accessing a small sample of online services, based on the PII they care about the most.

Papadopoulos et al. [200] also analyzed what leaks occur while accessing the same service through the mobile app and a mobile browser. Compared to the previous study by Leung et al [159], the authors broaden the definition of privacy leakage and besides personal information such as gender and email addresses, they also included device-specific information that can be used as identifiers, such as the list of installed apps, known SSIDs, connected wifi, operating system's build information, carrier, etc. Their study is also on a significant larger dataset and their SSL-capable monitoring proxy is able to capture and analyze network traffic even if apps use certificate pinning. Surprisingly the authors showed that in case of device-specific privacy leaks, there is a clear winner with mobile browsers leaking significantly less information compared to mobile apps. In most cases their results indicate that mobile apps leak tons of information which mobile browsers did or could not leak and they urge users to consider using the mobile browser whenever they have the choice. Since this may not always be possible because websites may provide poor functionality to mobile devices; and because mobile browsers constitute ordinary apps and include third-party trackers of their own, they proposed an anti-tracking mechanism that fortifies mobile apps from trackers. Their solution, antiTrackDroid, is based on the Xposed framework and filters all outgoing requests and blocks the ones delivering tracking information. Their evaluation results show that antiTrackDroid is able to reduce the privacy leaks by 27.41%, when it imposes a negligible overhead of less than 1 millisecond per request.

6.2.4 HTML5 WebAPI and Mobile Sensors

The introduction of WebAPI has standardized many functions and features, providing greater support for developers, enriching websites and web apps and improving the user experience [214]. Snyder et al. [237] presented a study on the use of HTML5 functions in a small set (10K) of popular websites. The functionality provided by HTML5 allows users to experience multimedia content without the hassle and, more importantly, the vulnerabilities introduced by external plugins or proprietary software, such as the Adobe Flash plugin which was progressively abandoned and substituted with HTML5 media elements [169]. At the same time, smartphone browsing has become very popular in the past years, with devices facilitating browsing even in screens that are comparatively small. This transition was possible with the introduction of multi-touch gestures and the performance improvement of mobile devices and the mobile network [115].

Browser fingerprinting has gathered a lot of attention in recent years and the research community has extensively studied the techniques that make it possible [114]. Eckersley [111] introduced the Panopticlick project and explored browser fingerprinting in depth. With the growing usage of smartphones, traditional desktop fingerprinting techniques [261] (e.g., canvas, screen and graphics fingerprinting, etc.), are becoming less effective as some information is being standardized in many mobile browsers [142]. On the other hand, the development of new mobile-specific HTML5 WebAPIs offered new avenues for trackers to exploit other types of data that were not present in desktops. As previous work [55, 56, 80, 97–100, 106, 117, 134, 142, 143, 177, 187, 216, 220, 289, 291] has shown, the huge amount of input collected by smartphones sensors resulted in new opportunities for device fingerprinting. A notable case is by Olejnik et al. [196], that explores how the Battery Status API yields information about the maximum capacity of the battery and the discharge time, which can be used to effectively track users across the web. This attracted a lot of attention which resulted in Firefox discontinuing its support of the Battery API.

Modern smartphones contain a wide range of sensors that collect information about the current state of the device and the environment surrounding it. Websites and applications have access to these sensors with most of them (e.g., gyroscope, accelerometer, proximity etc.) not explicitly requiring the permission of the user. Das et al [96] presented a study on web scripts accessing mobile sensors. Their system is based on a modified version of OpenWPM and Firefox (OpenWPM-Mobile) and emulates mobile behavior by overriding specific values such as the user agent, the platform name, the appVersion, etc. The authors tested their environment using the fingerprintjs2 library and EFF's Panopticlick test suite and found that their instrumented browser's fingerprint is similar to the fingerprint of the mobile Firefox browser running on a Moto G5 Plus device (except for Canvas and WebGL fingerprints). Using OpenWPM-Mobile they crawled the Alexa top 100K from two different geographical locations (US and Europe) and extracted low and high level

features from the captured JavaScript files as well as info from the HTTP data. The low level features correspond to browser properties accessed and function calls made by the script, while the high level features capture the tracking related behavior of scripts such as browser fingerprinting techniques and whether the script is blocked by certain adblocker lists. To facilitate the analysis the authors clustered the JavaScript files using the DBSCAN algorithm and merged the clusters that are similar using Random Forests. After manual analysis of scripts from different clusters they found that 37% of the scripts collect sensor data to perform some form of tracking and analytics such as audience recognition, ad impression verification, and session replay. Moreover, their results indicate that scripts that access the motion sensor also engage in some form of browser fingerprinting (e.g., canvas, web rtc, audio context and battery fingerprinting). While Das et al. study also targets WebAPI calls for mobile sensors, our work [170] (Chapter 4) presents significant differences. In regards to the actual datasets, [170] is on a considerably larger set of domains. Moreover, the system presented in [96] detects only a subset of the mobile-specific WebAPI calls and their study focuses only on sensor-based fingerprinting, thus offering a limited examination of the risks that users face, while our study frames its findings within an attack taxonomy and provides a more comprehensive evaluation of the feasibility of a wide range of sensor-based attacks. Furthermore, our crawling infrastructure uses actual mobile devices and provides a unique end-to-end view of data requests and access from the application layer down to the operating-system Android internals. In [96] the crawlers rely on a modified version of OpenWPM running on desktop machines, which could be detected by evasive websites, e.g., through canvas and emoji elements [155], that subsequently alter their behavior. Nonetheless, both studies provide an important and complimentary view of the mobile web ecosystem, and the combination of their findings and public datasets will be useful resources for the research community.

Bai et al. [72] also studied the problem of privacy inference based on sensors (PIS) and proposed a static instrumentation framework that hooks at sensor specific APIs (including Java and native code) and either blocks access to sensor data or provide apps with random values. SensorGuardian modifies the original apk file to insert hooks at different points and using a policy manager application provides coarse-grain access control over the sensor data that each application can use. To handle case where apps change their behavior during runtime, such as dynamic code loading, the authors examined the build-in libraries that are dynamically loaded during the app's start time and replaced the function pointers of sensor-related APIs in the global offset table with their custom functions. These wrapper functions allow, block or provide random values depending on the current control policies that the user has selected in the policy manager app. As the authors acknowledge this approach does not handle new dex files or binaries downloaded through the network since static instrumentation techniques cannot predict the files to be downloaded and instrument the dynamically downloaded files at runtime. Moreover,

the static instrumentation requires repackaging of the original apk which adds extra effort for inexperienced users and can also result in apps not working since they can easily detect if their signature has changed. These limitations can be addressed by incorporating Reaper in SensorGuardian's design that uses hooks during run time execution. Furthermore Reaper's stacktrace analysis can provide fine-grain access control and can distinguish sensor requests needed by the app's core functionality from those requested by third-party libraries linked with the app.

6.2.5 WebViews and Advertising

Numerous studies have showed that misconfigured hybrid apps pose a significant risk to users' privacy, and Luo et al. [167] identified several attacks against WebViews. The most notorious example is the `@JavascriptInterface` that allows JavaScript code to access Java methods. Rizzo et al. [223] evaluated the impact of such possible code injection attacks using static information flow analysis, while BridgeScope [281] assesses JavaScript interfaces based on a custom flow analysis. Additionally, Mutchler et al. [184] performed a large-scale analysis of more than a million mobile apps and identified that 28% contains at least one WebView vulnerability.

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 [92, 109, 201, 218, 219]. Meng et al. [175] 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. [215] 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. [221] 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. [193] 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 [254]. 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 [150].

6.3 Protection Mechanisms

Android apps suffer from the privacy implications of the integrated third-party libraries. As such, it was only natural that ad blocking mechanisms would appear for mobile platforms as well. In this section we classified various notable studies based on the protection mechanism they offer. Specifically we present significant and popular studies that offer a protection mechanism through some form of ad blocking in the user's device or using a VPV approach, studies that separate and sandbox third-party libs from the host application, systems that provide a fine-grained access control mechanism and studies that focused on protecting users from sensor-based side-channel attacks.

6.3.1 Ad-blockers and Network Monitoring

AdAway [2] is an ad blocker available for Android. It comes in the form of an app which requires root access and is based on a blacklist implemented using the hosts file. Unfortunately access to this specific file does not require root permission, and third-party libraries could easily read the blacklisted domains and simply update their domain to one not presented in the host file. Another disadvantage of this solution is that the black list should be actively maintained since trackers which are not present in the black list cannot be blocked. This also applies for the popular extensions Adblock Plus [3], which is available for the mobile Firefox browser. Similar approach takes the antitrackDroid [200] module that monitors the creation of network sockets at the OS level and blocks network flows by checking if the destination's domain name exist in a blacklist of mobile trackers. A major advantage over prior work is that the antitrackDroid fortifies both apps and mobile browsers. Protection mechanisms that use a VPN solution for monitoring network traffic do not necessarily require a blacklist and since they do not operate at the device allow for cross-platform compatibility. For example, Recon [219] is capable of blocking or adding noise to PII leaking requests by analyzing network traffic and using machine learning. Since it operates on top of Meddle, it redirects a mobile device's internet traffic to a VPN proxy and can be used by iOS and Windows phones as well. AntMonitor [233] leverages Android's SDK VPNService to detect sensitive data leakage. However it detects and prevents leakage of sensitive information only over unencrypted traffic. PrivacyGuard [241] is a VPN-based solution that prevents data leakage by providing fake data for known PIs. Their approach is based on the VPNService class and does not require a trusted VPN server. Unfortunately, it sends fake data not only to third-party trackers, but to first-parties as well, thus risking the application's seamless functionality.

6.3.2 App and Third-party Library Compartmentalization

AdDroid [204] proposed an advertising API for separating the privileged functionality of the host app from the advertising libraries. The AdSplit [231] extension separates integrated ads and the host app into different process and user IDs. Since the core functionality and the advertising code run in separate processes, it eliminates the ability of applications to request extra permissions on behalf of their ad libraries. Aframe [285] achieves process isolation between activities from libraries and the host app and also includes display and input isolation. Wang et al. [264] presented a sandboxing approach where third-party libraries are in an isolated file space and developers can assign separate permissions. NativeGuard [251] separates native third-party libraries from the original application by splitting it into two apps. Liu et al. [164] proposed PEDAL, a system that can identify libraries even when the source code is obfuscated and instruments apps for solving the problems of privilege inheritance. CompARTist [141], is a compiler-based application compartmentalization system that enforces privilege separation and fault isolation of advertisement libraries on Android. Unlike previous approaches, it is able to partition Android applications at compile-time into isolated, privilege-separated compartments for the host app and the included third-party libraries without the need to modify the operating system.

6.3.3 Fine-grained Access Control

FLEXDROID [230] is an extension to Android's permission system that provides dynamic, fine-grained access control for third-party libraries, and allows developers to separate permissions needed by host apps from those required by the libraries. FLEXDROID identifies the principal of the currently running code using stack inspection and, depending on the identified principal, allows or denies the request by dynamically adjusting the app's permissions according to the pre-specified permissions in the app's manifest. However, this approach presents several drawbacks compared to our work [108] (Chapter 2). The heavy instrumentation of the OS and apps presents a significant obstacle to adoption. Moreover, they require developers to incorporate specific code in the manifest to protect users, but do not provide developers with incentives to do so. On the other hand, Reaper gives control to the users and traces the permissions requested by apps in real time and distinguishes those requested by the app's core functionality from those requested by third-party libraries linked with the app. By monitoring and identifying the origin of every permission protected call as well as non-permission protected functions that yield PII, provides more context about the functionality and the permission required. This information about the origin enables users to make better decisions about whether to grant a permission or not. XPrivacy [40] is designed to prevent PII-access but does not distinguish libraries or core functionality; incorporating the origin information produced by Reaper would allow for more fine-grained access control and significantly improve the usability

aspect of this tool. MockDroid [78] modified the Android OS so as to replace sensitive information with fake values. Fu et al. [120] proposed a permission policy manager that monitors each library's method invocation and tracks the execution thread tree. Unfortunately this approach requires to analyze each library's source code which is impossible due to the enormous amount of third-party libraries in the wild.

6.3.4 Sensor Blocking

SensorGuardian [72] modifies the original apk file to insert hooks at different points and using a policy manager application provides coarse-grain access control over the sensor data that each application can use. Unfortunately, the static instrumentation requires repackaging of the original apk which adds extra effort for inexperienced users and can also result in apps not working since they can easily detect if their signature has changed. On the contrary, 6thSense [234] does not require any OS or app modifications and utilizes machine learning-based detection mechanisms to detect malicious behavior associated with sensors. The system is trained to recognize the device's normal sensor behavior, which may include tasks like calling, Web browsing or driving and it continually checks the device's sensor activity against these learned behaviors.

6.4 Dynamic Analysis and Exploration

It is evident so far that the Android ecosystem contains many pitfalls that are being heavily abused by different entities. The academic community has presented static and dynamic approaches for solving different problems, with both techniques having advantages and drawbacks. A significant challenge when performing dynamic analysis on mobile apps is the traversal of the app's graph through the simulation of user interactions, without any a priori knowledge of the interactive content that will be displayed in the app. Previous work [54, 68, 85, 135, 168, 211, 212, 249, 255, 275, 288, 292] has explored the dynamic traversal of an application from different perspectives, such as achieving high traversing coverage or identifying malicious behavior. Specifically, [85, 168, 211, 275] showed that such an approach has better coverage than using random input events (e.g., adb monkey). Unfortunately apart from requiring static analysis of the apk [54, 68, 135, 211, 249, 255, 275, 288] they may require some form of app instrumentation [68, 135, 211] or OS code modification [168, 212, 288] or are pinned to a specific Android version [54, 85, 288, 292]. In this chapter we present notable examples of systems that perform UI exploration and have improved code coverage.

UIAutomator [31] is a useful tool available from the Android SDK that can dump the interactable objects of the display and provide additional information about them. However, UIAutomator presents two major disadvantages that render it unsuitable. First, if the app

uses the `WindowManager.LayoutParams.FLAG_SECURE` option, UI Automator has to respect this specific flag and cannot output information about the objects being displayed. This flag is a security feature that treats the contents of the window as "secure" and prevents taking screenshots or being viewed on non secure displays [36]. This flag is not uncommon and is used to secure apps (e.g., PayPal) from side channel attacks, and can be used by apps or third-party libraries that want to evade dynamic analysis. (which is reflected in the prevalence of obfuscation [70]). Second, the performance overhead introduced is significant, rendering UI Automator unsuitable for a large scale analysis [108]. Dynodroid [168], is able to generate both UI and system inputs by viewing an app as an event-driven program. Their system interacts with an app by observing, selecting and executing an event according to the app's state. In [68] the authors propose App Explorer (A3E), a system for dynamic depth first exploration. It uses static, taint-style, dataflow analysis on the app's bytecode for constructing a control flow graph. By using this control flow graph they are able to transit between different activities. AppsPlayground [212] is a dynamic analysis framework for analyzing apps for malicious activity. The authors perform minor changes to the OS to support event triggering and fuzzy testing. HierarchyViewer [16] is an Android SDK utility that allows developers to monitor the visual representation of the View layout. PUMA [135] is a programmable UI automation tool, that allows the exploration of an app. However, it requires app instrumentation so as to trigger app-specific events. Whenever a specified code point is reached, the control is transferred back to PUMA for executing commands written in PUMAScripts. IntelliDroid [275] relies on targeted input generation for dynamic malware detection, and combines the benefits of static and dynamic analysis. SmartDroid [288], enforces a hybrid, static and dynamic approach to triggering UI-based conditions that lead to sensitive activities. Their proposed approach performs a depth-first traversal in Android 2.3.3 and requires heavy modifications to the operating system. HARVESTER [211], is a system that uses program slicing with code generation and dynamic execution in order to extract sensitive values from malware samples. Their approach is able to bypass restrictions by obfuscated code and, since it directly executes code fragments, does not need methods for UI automation. Carter et al. [85] presented Curiousdroid, an automated system for exercising Android apps without the need of modifying the source code. Even though it is able to identify interactable objects, it has only been tested on deprecated APIs (4-16), and the publicly available tool is for the severely outdated API 10 (Android v2.3). UIHarvester [108] (Chapter 2) utilizes hooks in the Android rendering process for identifying interactive elements and their properties, for traversing the app's graph without a priori knowledge of the app's functionality or visual characteristics. Moreover, to further increase coverage, it automatically completes the account creation and login process by leveraging Facebook's Single Sign-On platform. UIHarvester introduces negligible overhead that is 30-38 times smaller than that of Android's UI Automator, and improves coverage by $\sim 26\%$ compared to the tool that achieved the

highest coverage in a comparative study [89].

Chapter 7

Conclusion

7.1 Summary

We showed that despite the efforts of the research community and the strong demand for better access control mechanisms, mobile users' privacy is still at risk. While the Android operating system has made significant improvements over the last years and its better than what it used to be, there are still unresolved issues that create opportunities for attacks with severe ramifications and place even security-cautious users at risk. In this dissertation we explore the problems that arise because of the absence of sufficient access control mechanisms in Android, the poorly-conceived OS design choices, and Android's inherent relation with novel features of the mobile web. Android is making significant efforts in achieving a balance between user privacy and user experience, but still has a long way to go. We hope that the work presented will contribute to the ongoing body of research pushing for better permission and access control management in Android.

At first, we identified that the proliferation and prevalence of third-party libraries renders them a significant privacy risk. To address this issue we developed Reaper, a novel dynamic analysis system that traces the origin of permission-protected calls and non-protected calls that access PII. Our subsequent study on over 5K of the most popular apps, revealed the extent of libraries accessing sensitive data and found that certain permission-protected calls were used exclusively by these libraries and not by the apps' core functionality. Reaper's functionality can enhance Android's fine-grained run time permission system and enable users to prevent third parties from accessing their personal data. Additionally, the functionality offered by Reaper can augment access control systems that are already in place.

In this dissertation we also explore attacks that were previously limited to mobile apps and can now migrate to the mobile. We presented a comprehensive evaluation of the threats that mobile users face when browsing the Web, due to capabilities offered by modern browsers. Specifically, we conducted the largest and most extensive study to date on the use of mobile-specific WebAPI calls in the wild. Our findings demonstrate that WebAPI capabilities are actively being used by websites for accessing mobile sensors. To provide

the appropriate context that highlights the true threat posed by this practice, we created a taxonomy of sensor-based attacks compiled from a wide range of attacks demonstrated in prior work. Our subsequent in-depth analysis correlated the sensor data currently being accessed by websites and the data requirements of prior attacks, leading to several alarming findings. Apart from the fact that the vast majority of the websites that leverage mobile-specific WebAPI calls can carry out privacy-invasive attacks, we also found that 5.4% of those websites included third-party scripts that accessed sensor data and were hosted on domains flagged as malware by security services. We believe that our findings support the need for more stringent policies for websites attempting to access sensor data, allowing users to explicitly declare preferences and set their own privacy policy.

Furthermore, as users spend the majority of their browsing time within mobile apps, mobile ads will often reach their audience through in-app ads. 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. 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.

7.2 Directions for Future Work

We have discussed in the previous chapters the limitations of our work and we outlined possible solutions that can be explored as part of future work. In the following we summarize the steps we plan to follow in order to extend the work presented in this dissertation.

In-depth exploration of Android ecosystem. The intricacies of the permission system render it an obscure component of the OS, and a significant body of work has focused on shedding light on its inner workings. A defining characteristic of Android is being open source, which has allowed a plethora of vendors to release Android smartphones. In practice, however, device manufacturers are prone to customizing the user interface as well

as modifying the underlying operating system based on proprietary software or hardware. We plan to conduct a novel in-depth exploration of the Android operating system for obtaining a more accurate and complete mapping of Android API calls to their corresponding permissions through the dynamic analysis of apps. Inspired by traditional taint analysis techniques we will design a mechanism to narrow down the Android API calls that need to be monitored, as hooking and monitoring every function of the Android OS is impractical from an engineering standpoint. The main idea behind the proposed mechanism lies in the way Android operates and the fact that Inter Process Communication (IPC) is conducted through the Binder. Whenever a function requests access to a permission-protected resource the ensuing chain of events, which may include one or more entities (threads or processes), will eventually reach Android Server's permission-checking mechanism. These entities communicate through the Binder Interface. By monitoring Binder we can find the caller threads and the functions that initiated that Binder transaction. Even though Binder is the entity responsible for inter-process communication in Android and used for different activities, it can be used as an entry point for efficiently narrowing down the functions that need to be hooked without the risk of losing any potential permission-protected functions.

Third-party library identification. It is evident that significant research efforts have been applied towards identifying third-party libs. Even though these systems have made significant progress and presented encouraging results, they still suffer from the inherent limitations of static analysis. To overcome the limitations of prior systems, we propose a novel approach for inferring commonly used third-party libraries based on the origin of permission-protected calls. Using Reaper we can obtain information regarding dynamically loaded library code, which will complement previous studies and lead to a comprehensive and constantly updated list of third-party libraries package names. However, apart from enriching the information about known libraries, our approach can also be applied for identifying new third-party libraries. Since Reaper returns the origin of permission-protected calls and non-protected calls that access PII, we can use the information contained in the stacktraces to identify package names that exist in multiple apps. If the apps are from different developers (i.e., signed with different keys), that is a strong indication that these package names correspond to libraries from third parties (as opposed to a developer reusing their own code across apps). The effectiveness of this approach relies on the third-party libraries actually accessing device characteristics or PII. These resources will be publicly released to facilitate the plethora of research projects exploring the Android ecosystem.

Dynamic exploration. Mobile UI exploration has long stumped developers and researchers alike. From a researcher's perspective automatic app interaction and exploration can be of great importance for dynamically analyzing apps. We will enhance the coverage of UIHarvester by applying machine learning on the interactable elements dis-

played on the screen based on their class (e.g., TextView, Button, Image, etc.), their properties (e.g., font, size, color, etc.) and structure (e.g., position and orientation on the screen), in order to identify which elements provide better code coverage. Additionally, as mobile websites and apps follow the design principles of human-computer interaction in order to make the user's experience as friendly as possible, we will explore whether the principles of a user-centered design can provide valuable knowledge on how to effectively analyze Android apps.

Bibliography

- [1] 10 reasons why mobile advertising is more effective than internet advertising. <https://www.marketingdive.com/ex/mobilemarketer/cms/opinion/columns/4755.html>.
- [2] AdAway. <http://free-software-for-android.github.io/AdAway/>.
- [3] Adblock Plus android firefox plugin. <https://addons.mozilla.org/en-US/android/addon/adblock-plus/>.
- [4] Alexa - top 50 banks and institutions. https://www.alexa.com/topsites/category/Business/Financial_Services/Banking_Services/Banks_and_Institutions.
- [5] Android distribution between platform versions. <https://developer.android.com/about/dashboards/index.html>.
- [6] Android library statistics. <https://www.appbrain.com/stats/libraries/>.
- [7] Android View Class. <https://developer.android.com/reference/android/view/View.html>.
- [8] Appsflyer - mobile app tracking & attribution. <https://www.appsflyer.com/>.
- [9] Appsflyer sdk integration - android. <https://support.appsflyer.com/hc/en-us/articles/207032126-AppsFlyer-SDK-Integration-Android/>.
- [10] Axplorer - demystifying the Android application framework. <http://axplorer.org/>.
- [11] Axplorer - demystifying the Android application framework. <http://axplorer.org/>.
- [12] The eu general data protection regulation. <https://eugdpr.org>.
- [13] Forbes - Google users: You're the product, not the customer. <https://www.forbes.com/sites/benkepes/2013/12/04/google-users-youre-the-product-not-the-customer/>.

- [14] Google Mobile Services, Google's most popular apps, all in one place. <https://www.android.com/gms/>.
- [15] The Google Play apps that say they don't collect your data, and then do. <https://nakedsecurity.sophos.com/2017/05/10/the-google-play-apps-that-say-they-dont-collect-your-data-and-then-do/>.
- [16] Hierarchy Viewer. <https://developer.android.com/studio/profile/hierarchy-viewer.html>.
- [17] How many people have phones in the world? <https://www.bankmycell.com/blog/how-many-phones-are-in-the-world>.
- [18] JailbreakRoot Detection Evasion Study on iOS and Android. <http://www.delaat.net/rp/2015-2016/p51/report.pdf>.
- [19] A Magic Mask to Alter System Systemlessly. <https://forum.xda-developers.com/apps/magisk/official-magisk-v7-universal-systemless-t3473445>.
- [20] Mdn web docs - magnetometer. <https://developer.mozilla.org/en-US/docs/Web/API/Magnetometer/Magnetometer>.
- [21] Mobile operating system market share worldwide. feb 2021 - feb 2022. <https://gs.statcounter.com/os-market-share/mobile/worldwide>.
- [22] Mobileappscrutinator:a simple yet efficient dynamic analysis approach for detecting privacy leaks across mobile oss. <https://arxiv.org/pdf/1605.08357v2.pdf>.
- [23] Mozilla support - does firefox share my location with websites? <https://support.mozilla.org/en-US/kb/does-firefox-share-my-location-websites>.
- [24] Over half of 3rd party Android in-app ad libraries have privacy issues and possible security holes. <http://www.androidauthority.com/3rd-party-android-app-privacy-issues-67669/>.
- [25] Pc world - researchers: Mobile users at risk from lack of HTTPS use by mobile ad libraries. <http://www.pcworld.com/article/2093460/mobile-users-at-risk-from-lack-of-https-use-by-mobile-ad-libraries-security-researchers-say.html>.
- [26] Permission protection level. https://android.googlesource.com/platform/frameworks/base.git/+android-cts-5.1_r19/core/res/AndroidManifest.xml.
- [27] Raccoon - APK downloader. <http://www.onyxbits.de/raccoon>.

- [28] A repository of Android common libraries and advertisement libraries. <https://github.com/serval-snt-uni-lu/CommonLibraries>.
- [29] Request prompts for dangerous permissions. <https://developer.android.com/guide/topics/permissions/overview#dangerous-permission-prompt>.
- [30] Threatpost - Unnamed Android Mobile Ad Library Poses Large-Scale Risk. <https://threatpost.com/unnamed-android-mobile-ad-library-poses-large-scale-risk/102543/>.
- [31] Ui automator - ui testing framework suitable for cross-app functional ui testing across system and installed apps. <https://google.github.io/android-testing-support-library/docs/uiautomator/>.
- [32] Us banks on the internet. http://www.thecommunitybanker.com/bank_links/.
- [33] Virustotal: Analyze suspicious files and urls to detect types of malware. <https://www.virustotal.com>.
- [34] Virustotal: goggle.com. <https://tinyurl.com/VTgoggle-com>.
- [35] Virustotal: youtube.com. <https://tinyurl.com/VTyoutube-com>.
- [36] Window layout - flag_secure. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE.
- [37] Xposed Framework API. <http://api.xposed.info/reference/packages.html>.
- [38] Xposed Hook Overhead. <https://github.com/townbull/XposedExp/blob/master/overhead.md>.
- [39] Personally identifiable information (pii), 2014. <https://searchfinancialsecurity.techtarget.com/definition/personally-identifiable-information>.
- [40] The ultimate, yet easy to use, privacy manager for android, 2016. <https://github.com/M66B/XPrivacy>.
- [41] Iab fy 2018 podcast ad revenue study, 2019. https://www.iab.com/wp-content/uploads/2019/06/Full-Year-2018-IAB-Podcast-Ad-Rev-Study_6.03.19_vFinal.pdf.
- [42] Number of available applications in the google play store from december 2009 to june 2019, 2019. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>.

- [43] Top 7 Mobile Security Threats, 2019. <https://usa.kaspersky.com/resource-center/threats/top-seven-mobile-security-threats-smart-phones-tablets-and-mobile-internet-devices-what-the-future-has-in-store>.
- [44] What's Shaping the Digital Ad Market, 2019. <https://www.emarketer.com/content/global-digital-ad-spending-2019>.
- [45] Year-over-year change of advertising expenditure in selected countries from 2016 to 2018, 2019-07-22. <https://www.statista.com/statistics/276805/global-advertising-market-forecast/>.
- [46] Jmango - mobile apps vs. mobile websites: User preferences, 2020. <https://jmango360.com/wiki-pages-trends/mobile-app-vs-mobile-website-statistics/>.
- [47] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 1151–1164, New York, NY, USA, 2018. ACM.
- [48] Jagdish Prasad Achara, Gergely Acs, and Claude Castelluccia. On the unicity of smartphone applications. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society, WPES '15*.
- [49] AdMob. How much revenue can you earn from AdMob. <https://admob.google.com/home/resources/how-much-revenue-can-you-earn-from-admob/>, 2021. Accessed: 2021-02-25.
- [50] AdSense. Ad implementation policies - Ad placement policies. <https://support.google.com/adsense/answer/1346295>, 2021. Accessed: 2021-02-23.
- [51] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. 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, 2015.
- [52] Furkan Alaca and Paul C van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 289–301. ACM, 2016.
- [53] David M Allen. Mean square error of prediction as a criterion for selecting variables. *Technometrics*, 13(3):469–475, 1971.

- [54] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *ASE '12*, 2012.
- [55] Irene Amerini, Rudy Becarelli, Roberto Caldelli, Alessio Melani, and Moreno Nicolai. Smartphone fingerprinting combining features of on-board sensors. *IEEE Transactions on Information Forensics and Security*, 12(10):2457–2466, 2017.
- [56] Irene Amerini, Paolo Bestagini, Luca Bondi, Roberto Caldelli, Matteo Casini, and Stefano Tubaro. Robust smartphone fingerprint by mixing device sensors features for mobile strong authentication. *Electronic Imaging*, 2016(8):1–8, 2016.
- [57] S Abhishek Anand and Nitesh Saxena. Speechless: Analyzing the threat to speech privacy from smartphone motion sensors. In *2018 IEEE Symposium on Security and Privacy (SP). Vol. 00*, pages 116–133, 2018.
- [58] Andrei Popescu. Geolocation API. <https://www.w3.org/TR/geolocation-API/>. Accessed: 2018-07-13.
- [59] Android Developers. Class Overview: BroadcastReceiver. <http://developer.android.com/reference/android/content/BroadcastReceiver.html>.
- [60] Anssi Kostiainen. Ambient light sensor API. <https://www.w3.org/TR/ambient-light/>. Accessed: 2018-07-13.
- [61] Anssi Kostiainen. Vibration API. <https://www.w3.org/TR/vibration/>. Accessed: 2018-07-13.
- [62] Anssi Kostiainen, Alexander Shalamov. Accelerometer. <https://www.w3.org/TR/accelerometer/>. Accessed: 2018-07-13.
- [63] Anssi Kostiainen, Rijubrata Bhaumik. Proximity sensor API. <https://www.w3.org/TR/proximity/>. Accessed: 2018-07-13.
- [64] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*.
- [65] Elias Athanasopoulos, Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. Naclroid: Native code isolation for android applications. In *European Symposium on Research in Computer Security, ESORICS '16*.

- [66] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*.
- [67] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2012.
- [68] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA '13*, 2013.
- [69] M. Backes, S. Bugiel, O. Schranz, P. v. Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroSP)*, 2016.
- [70] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*.
- [71] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Ocateau, and Sebastian Weisgerber. On demystifying the android application framework: Revisiting android permission specification analysis. In *25th USENIX Security Symposium USENIX Security 16*.
- [72] Xiaolong Bai, Jie Yin, and Yu-Ping Wang. Sensor guardian: prevent privacy inference on android sensors. *EURASIP Journal on Information Security*, 2017(1):10, 2017.
- [73] Mahesh Balakrishnan, Iqbal Mohamed, and Venugopalan Ramasubramanian. Where's that phone?: Geolocating ip addresses on 3g networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, IMC '09*.
- [74] Anna M. Bardone-Cone and Kamila M. Cass. Investigating the impact of pro-anorexia websites: a pilot study. *European Eating Disorders Review*, 14(4):256–262, 2006.
- [75] Paul Barford, Igor Canadi, Darja Krushevskaja, Qiang Ma, and S. Muthukrishnan. Adscape: Harvesting and analyzing online display ads. In *Proceedings of the 23rd International Conference on World Wide Web, WWW '14*, pages 597–608, New York, NY, USA, 2014. ACM.
- [76] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 481–496, Austin, TX, 2016. USENIX Association.

- [77] Ben Alman. Monkey-patch (hook) functions for debugging and stuff. <https://github.com/cowboy/javascript-hooker>. Accessed: 2018-04-23.
- [78] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mock-droid: trading privacy for application functionality on smartphones. In *HotMobile '11*, 2011.
- [79] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. Hard drive side-channel attacks using smartphone magnetic field sensors. In *International Conference on Financial Cryptography and Data Security*, pages 489–496. Springer, 2015.
- [80] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. Mobile device identification via sensor fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014.
- [81] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. In *MoST '13*.
- [82] Armir Bujari, Bogdan Licar, and Claudio E Palazzi. Movement pattern recognition through smartphone's accelerometer. In *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pages 502–506. IEEE, 2012.
- [83] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. *HotSec*, 11:9–9, 2011.
- [84] Liang Cai and Hao Chen. On the practicality of motion based keystroke inference attack. In *International Conference on Trust and Trustworthy Computing*, pages 273–290. Springer, 2012.
- [85] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security FC '16*.
- [86] Supriyo Chakraborty, Wentao Ouyang, and Mani Srivastava. Lightspy: Optical eavesdropping on displays using light sensors on mobile devices. In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 2980–2989. IEEE, 2017.
- [87] Gong Chen, Wei Meng, and John Copeland. Revisiting mobile advertising threats with madlife. In *The World Wide Web Conference*, pages 207–217, 2019.
- [88] David Choffnes. App vs web. <https://recon.meddle.mobi/appvsweb/>, 2016.
- [89] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *ASE '15*, 2015.

- [90] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.
- [91] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *NDSS '17*, 2017.
- [92] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *NDSS*, 2017.
- [93] Cortesi, Aldo and Hils, Mayimilian and Kriechbaumer, Thomas. mitmproxy. <https://mitmproxy.org>. v. 3.0.3.
- [94] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 123–134, 2014.
- [95] Daniel C. Burnett, Adam Bergkvist, Cullen Jennings, Anant Narayanan, Bernard Aboba. Media capture API. <https://www.w3.org/TR/mediacapture-streams/>. Accessed: 2018-07-13.
- [96] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of ACM CCS, October 2018*, 2018.
- [97] Anupam Das, Nikita Borisov, and Matthew Caesar. 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*, pages 441–452. ACM, 2014.
- [98] Anupam Das, Nikita Borisov, and Matthew Caesar. Tracking mobile web users through motion sensors: Attacks and defenses. In *NDSS*, 2016.
- [99] Anupam Das, Nikita Borisov, and Edward Chou. Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures. *Proceedings on Privacy Enhancing Technologies*, 2018(1):88–108, 2018.
- [100] Erhan Davarci, Betul Soysal, Imran Erguler, Sabri Orhun Aydin, Onur Dincer, and Emin Anarim. Age group detection using smartphone motion sensors. In *Signal Processing Conference (EUSIPCO), 2017 25th European*, pages 2201–2205. IEEE, 2017.

- [101] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. Free for all! assessing user data exposure to advertising libraries on android. In *23rd Annual Network and Distributed System Security Symposium, NDSS '16*.
- [102] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *WOOT*, 2014.
- [103] Android Developers. Interstitial (legacy API). <https://developers.google.com/admob/android/interstitial>, 2021. Accessed: 2021-02-25.
- [104] Android Developers. Multi-Window Support. <https://developer.android.com/guide/topics/ui/multi-window>, 2021. Accessed: 2021-04-26.
- [105] Android Developers. Permissions updates in Android 11. <https://developer.android.com/about/versions/11/privacy/permissions>, 2021. Accessed: 2021-04-26.
- [106] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. Accelprint: Imperfections of accelerometers make smartphones trackable. In *NDSS*, 2014.
- [107] Michalis Diamantaris, Francesco Marcantoni, Sotiris Ioannidis, and Jason Polakis. 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), July 2020.
- [108] Michalis Diamantaris, Elias P. Papadopoulos, Evangelos P. Markatos, Sotiris Ioannidis, and Jason Polakis. Reaper: Real-time app analysis for augmenting the android permission system. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY '19*, pages 37–48. ACM, 2019.
- [109] Michalis Diamantaris, Elias P. Papadopoulos, Evangelos P. Markatos, Sotiris Ioannidis, and Jason Polakis. 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, 2019.
- [110] DoubleVerify. DOUBLEVERIFY PRIVACY NOTICES - SOLUTIONS PRIVACY NOTICE. <https://doubleverify.com/privacy-notice/>, 2021.
- [111] Peter Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2010.
- [112] Nicole Eling, Siegfried Rasthofer, Max Kolhagen, Eric Bodden, and Peter Buxmann. Investigating users' reaction to fine-grained data requests: A market experiment. In *49th Hawaii International Conference on System Sciences, HICSS 2016*.

- [113] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*.
- [114] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1388–1401. ACM, 2016.
- [115] Fazal-e-Amin. Characterization of web browser usage on smartphones. *Computers in human behavior*, 51:896–902, October 2015.
- [116] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11*.
- [117] Tobias Fiebig, Jan Krissler, and Ronny Hänsch. Security impact of high resolution smartphone cameras. In *WOOT*, 2014.
- [118] Maximiliano Firtman. Mobile HTML5 Compatibility on Mobile Devices. <http://mobilehtml5.org/>. Accessed: 2018-04-22.
- [119] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057, 2017.
- [120] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang, and Xin Wang. Perman: Fine-grained permission management for android applications. In *ISSRE' 17*, 2017.
- [121] Matteo Gadaleta and Michele Rossi. Idnet: Smartphone-based gait recognition with convolutional neural networks. *Pattern Recognition*, 74:25–37, 2018.
- [122] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. Pmdroid: Permission supervision for android advertising. In *Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems, SRDS '15*.
- [123] Daniel Genkin, Mihir Pattani, Roei Schuster, and Eran Tromer. Synesthesia: Detecting screen content via remote acoustic side channels. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.
- [124] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *International cryptology conference*, pages 444–461. Springer, 2014.

- [125] ghostwords. Browser fingerprinting protection for everybody. <https://github.com/ghostwords/chameleon>. Accessed: 2018-06-09.
- [126] Global Stats. Mobile and tablet internet usage exceeds desktop for first time worldwide. <http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide>. Accessed: 2018-04-19.
- [127] Google. WebView - A View that displays web pages. <https://developer.android.com/reference/android/webkit/WebView>, 2020. Accessed: 2020-10-28.
- [128] Google. Provide advance notice to the Google Play App Review team. https://support.google.com/googleplay/android-developer/contact/adv_note, 2021. Accessed: 2021-04-26.
- [129] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, WISEC '12*.
- [130] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.
- [131] Glenn Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Macmillan, 2014.
- [132] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*, pages 108–122. Springer, 2015.
- [133] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 100–110, Piscataway, NJ, USA, 2015. IEEE Press.
- [134] Jun Han, Emmanuel Owusu, Le T Nguyen, Adrian Perrig, and Joy Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9. IEEE, 2012.
- [135] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys '14*, 2014.

- [136] Samuli Hemminki, Petteri Nurmi, and Sasu Tarkoma. Accelerometer-based transportation mode detection on smartphones. In *Proceedings of the 11th ACM conference on embedded networked sensor systems*, page 13. ACM, 2013.
- [137] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [138] Duncan Hodges and Oliver Buckley. Reconstructing what you said: Text inference using smartphone motion. *IEEE Transactions on Mobile Computing*, 2018.
- [139] Shaohan Hu, Lu Su, Shen Li, Shiguang Wang, Chenji Pan, Siyu Gu, Md Tanvir Al Amin, Hengchang Liu, Suman Nath, Romit Roy Choudhury, and Tarek F. Abdelzaher. Experiences with enav: A low-power vehicular navigation system. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 433–444, New York, NY, USA, 2015. ACM.
- [140] Jingyu Hua, Zhenyu Shen, and Sheng Zhong. 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):286–297, 2017.
- [141] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 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*, CCS '17, pages 1037–1049, New York, NY, USA, 2017. ACM.
- [142] Thomas Hupperich, Davide Maiorca, Marc Kühner, Thorsten Holz, and Giorgio Giacinto. 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*, pages 191–200. ACM, 2015.
- [143] Felix Juefei-Xu, Chandrasekhar Bhagavatula, Aaron Jaech, Unni Prasad, and Marios Savvides. 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*, pages 8–15. IEEE, 2012.
- [144] Matthew Kaplan. 52 In-App Advertising Statistics You Should Know. <https://www.inmobi.com/blog/2019/05/21/52-in-app-advertising-statistics-you-should-know>, 2021. Accessed: 2021-02-25.
- [145] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *USEC 2012*.

- [146] Kristen Kennedy, Eric Gustafson, and Hao Chen. Quantifying the effects of removing permissions from android applications. In *MoST '12*.
- [147] Kicelo and Dominik Schuermann. Adaway default blocklist. <https://adaway.org/hosts.txt>, 2016.
- [148] Hyungsub Kim, Sangho Lee, and Jong Kim. Exploring and mitigating privacy threats of html5 geolocation api. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 306–315. ACM, 2014.
- [149] Hyungsub Kim, Sangho Lee, and Jong Kim. Exploring and mitigating privacy threats of html5 geolocation api. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14*, pages 306–315, New York, NY, USA, 2014. ACM.
- [150] Joongyum Kim, Jung-hwan Park, and Sooel Son. The abuser inside apps: Finding the culprit committing mobile ad fraud. In *NDSS*, 2011.
- [151] Joongyum Kim, Jung-hwan Park, and Sooel Son. The abuser inside apps: Finding the culprit committing mobile ad fraud. In *28th Network & Distributed System Security Symposium (NDSS'21)*, pages 1–16, 2020.
- [152] Jennifer R Kwapisz, Gary M Weiss, and Samuel A Moore. Activity recognition using cell phone accelerometers. *ACM SigKDD Explorations Newsletter*, 12(2):74–82, 2011.
- [153] Nicholas D Lane, Petko Georgiev, and Lorena Qendro. Deeppear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015.
- [154] Pierre Laperdrix. *Browser Fingerprinting: Exploring Device Diversity to Augment Authentication and Build Client-Side Countermeasures*. PhD thesis, Rennes, INSA, 2017.
- [155] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 878–894. IEEE, 2016.
- [156] Adam Lella. U.S. Smartphone Penetration Surpassed 80 Percent in 2016. <https://www.comscore.com/Insights/Blog/US-Smartphone-Penetration-Surpassed-80-Percent-in-2016>. Accessed: 2018-04-18.
- [157] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. Don't kill my ads!: Balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems Applications, Hot-Mobile '12*, 2012.

- [158] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *USENIX Security Symposium*, 2016.
- [159] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. Should you use the app for that?: Comparing the privacy implications of app- and web-based online services. In *Proceedings of the 2016 Internet Measurement Conference, IMC '16*, 2016.
- [160] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*, 2016.
- [161] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 335–346. IEEE, 2017.
- [162] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing, UbiComp '12*.
- [163] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS '14*.
- [164] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15*.
- [165] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *IEEE Transactions on Mobile Computing*, pages 1–1, 2019.
- [166] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. Alde: privacy risk analysis of analytics libraries in the android ecosystem. In *12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2016.
- [167] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352, 2011.

- [168] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *ESEC/FSE '13*, 2013.
- [169] Yogesh Maheshwari and Y Raghu Reddy. A study on migrating flash files to html5/javascript. In *Proceedings of the 10th Innovations in Software Engineering Conference*, pages 112–116. ACM, 2017.
- [170] Francesco Marcantoni, Michalis Diamantaris, Sotiris Ioannidis, and Jason Polakis. 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*. ACM, 2019.
- [171] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.
- [172] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 551–562. ACM, 2011.
- [173] McAfee. Customer URL Ticketing System, Check Single URL. <https://trustedsource.org/>. Accessed: 2019-04-22.
- [174] Maryam Mehrnezhad, Ehsan Toreini, Siamak F Shahandashti, and Feng Hao. Stealing pins via mobile sensors: actual risk versus user perception. *International Journal of Information Security*, 17(3):291–313, 2018.
- [175] Wei Meng, Ren Ding, Simon P. Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *23rd Annual Network and Distributed System Security Symposium, NDSS '16*.
- [176] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 319–333. IEEE, 2017.
- [177] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, pages 1053–1067, 2014.
- [178] Elinor Mills. Device identification in online banking is privacy threat, expert says. <https://www.cnet.com/news/device-identification-in-online-banking-is-privacy-threat-expert-says/>.

- [179] Emiliano Miluzzo, Alexander Varshavsky, Suhril Balakrishnan, and Romit Roy Choudhury. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336. ACM, 2012.
- [180] Prashanth Mohan, Suman Nath, and Oriana Riva. Prefetching mobile ads: Can advertising systems afford it? In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 267–280, New York, NY, USA, 2013. ACM.
- [181] Tyler Moore and Benjamin Edelman. Measuring the perpetrators and funders of typosquatting. In *International Conference on Financial Cryptography and Data Security*, pages 175–191. Springer, 2010.
- [182] Mounir Lamouri, Marcos Cáceres. Screen orientation API. <https://www.w3.org/TR/screen-orientation/>. Accessed: 2018-07-13.
- [183] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of W2SP 2012*, May 2012.
- [184] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, 2015.
- [185] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.
- [186] Sashank Narain, Amirali Sanatinia, and Guevara Noubir. Single-stroke language-agnostic keylogging using stereo-microphones and domain specific machine learning. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 201–212. ACM, 2014.
- [187] Sashank Narain, Triet D Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 397–413. IEEE, 2016.
- [188] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.
- [189] Suman Nath. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15*.

- [190] Sarfraz Nawaz and Cecilia Mascolo. Mining users' significant driving routes with low-power sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, pages 236–250, New York, NY, USA, 2014. ACM.
- [191] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the "s" in https. In *Proceedings of the 10th ACM CoNEXT*, 2014.
- [192] Khuong An Nguyen, Raja Naeem Akram, Konstantinos Markantonakis, Zhiyuan Luo, and Chris Watkins. Location tracking using smartphone accelerometer and magnetometer traces. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 96:1–96:9, New York, NY, USA, 2019. ACM.
- [193] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share first, ask later (or never?)-studying violations of gdpr's explicit consent in android apps. In *USENIX Security Symposium*, 2021.
- [194] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and privacy (SP), 2013 IEEE symposium on*, pages 541–555. IEEE, 2013.
- [195] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. On the workings and current practices of web-based device fingerprinting. *IEEE Security & Privacy*, 12(3):28–36, 2014.
- [196] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The leaking battery. In *Data Privacy Management, and Security Assurance*, pages 254–263. Springer, 2015.
- [197] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. Battery status not included: Assessing privacy in web standards. In *3rd International Workshop on Privacy Engineering (IWPE'17)*. San Jose, United States, 2017.
- [198] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACcessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012.
- [199] Pierluigi Paganini. Android apps use the motion sensor to evade detection and deliver Anubis malware. <https://securityaffairs.co/wordpress/80037/malware/android-apps-motion-sensor.html>, 2019.

- [200] Elias P. Papadopoulos, Michalis Diamantaris, Panagiotis Papadopoulos, Thanasis Petsas, Sotiris Ioannidis, and Evangelos P. Markatos. The long-standing privacy debate: Mobile websites vs mobile apps. In *Proceedings of the 26th International Conference on World Wide Web, WWW '17*, 2017.
- [201] Elias P Papadopoulos, Michalis Diamantaris, Panagiotis Papadopoulos, Thanasis Petsas, Sotiris Ioannidis, and Evangelos P Markatos. 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, 2017.
- [202] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. Exclusive: How the (synced) cookie monster breached my encrypted vpn session. EuroSec'18, 2018.
- [203] BI INDIA PARTNER. 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>, 2020. Accessed: 2021-04-26.
- [204] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12*.
- [205] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12*, New York, NY, USA, 2012. Association for Computing Machinery.
- [206] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security, EuroSec '14*.
- [207] Dan Ping, Xin Sun, and Bing Mao. 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*, pages 24:1–24:12, New York, NY, USA, 2015. ACM.

- [208] Sebastian Poehlau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS '14*.
- [209] Andrea Possemato and Yanick Fratantonio. Towards {HTTPS} everywhere on android: We are not there yet. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 343–360, 2020.
- [210] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. ispy: Automatic reconstruction of typed input from compromising reflections. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11*, pages 527–536, New York, NY, USA, 2011. ACM.
- [211] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security, NDSS '16*.
- [212] Vaibhav Rastogi, Yan Chen, and William Enck. Appsp playground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13*.
- [213] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *Proc. 23rd Annual Network & Distributed System Security Symposium, NDSS '16*.
- [214] Ashis Kumar Ratha, Shibani Sahu, and Priya Meher. *Html5 in web development: A new approach*. 2018.
- [215] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 603–620, 2019.
- [216] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks (TOSN)*, 6(2):13, 2010.
- [217] Tomas Reimers/Github. Axolotl Machine Learning Framework. <https://github.com/tomasreimers/axolotl>, 2017. Accessed: 2020-10-28.
- [218] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. A longitudinal study of pii leaks across android app versions. In *Network and Distributed System Security Symposium*, 2018.

- [219] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16*, 2016.
- [220] Yanzhi Ren, Yingying Chen, Mooi Choo Chuah, and Jie Yang. 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*, pages 149–157. IEEE, 2013.
- [221] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, Serge Egelman, et al. "won't somebody think of the children?" examining coppa compliance at scale. In *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*, 2018.
- [222] Rich Tibbett, Tim Volodine, Steve Block, Andrei Popescu. Device orientation event. <https://www.w3.org/TR/orientation-event/>. Accessed: 2018-07-13.
- [223] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. Babelview: Evaluating the impact of code injection attacks in mobile webviews. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 25–46. Springer, 2018.
- [224] rovo89. Xposed framework. <https://repo.xposed.info>. Accessed: 2018-06-14.
- [225] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. A long way to the top: Significance, structure, and stability of internet top lists. In *IMC*, 2018.
- [226] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.
- [227] Android SDK. SYSTEM_ALERT_WINDOW permission. https://developer.android.com/reference/android/Manifest.permission#SYSTEM_ALERT_WINDOW, 2021. Accessed: 2021-02-10.
- [228] Android SDK. Window Layout - FLAG_SECURE. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE, 2021. Accessed: 2021-02-10.
- [229] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. A measurement study of tracking in paid mobile applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec '15*, 2015.

- [230] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEX-DROID: enforcing in-app privilege separation in android. In *23rd Annual Network and Distributed System Security Symposium, NDSS '16*.
- [231] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security '12*.
- [232] Chao Shen, Shichao Pei, Zhenyu Yang, and Xiaohong Guan. Input extraction via motion-sensor behavior analysis on smartphones. *Computers & Security*, 53:143–155, 2015.
- [233] Anastasia Shuba, Anh Le, Minas Gjoka, Janus Varmarken, Simon Langhoff, and Athina Markopoulou. 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*, pages 25–27. ACM, 2015.
- [234] Amit Kumar Sikder, Hidayet Aksu, and A Selcuk Uluagac. 6thsense: A context-aware sensor-based attack detector for smart devices. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 397–414, 2017.
- [235] Laurent Simon and Ross Anderson. Pin skimmer: Inferring pins through the camera and microphone. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 67–78. ACM, 2013.
- [236] Prabhsimran Singh. 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>, 2020.
- [237] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, pages 97–110. ACM, 2016.
- [238] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 179–194. ACM, 2017.
- [239] Konstantinos Solomos, Panagiotis Ilia, Sotiris Ioannidis, and Nicolas Kourtellis. Talos: An automated framework for cross-device tracking detection. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2019.

- [240] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *23rd Annual Network and Distributed System Security Symposium, NDSS '16*.
- [241] Yihang Song and Urs Hengartner. Privacyguard: A vpn-based platform to detect information leakage on android devices. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15*.
- [242] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing SAC '13*.
- [243] Raphael Spreitzer. 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*, pages 51–62. ACM, 2014.
- [244] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [245] Terry Stancheva. 50+ App Revenue Statistics - Mobile Is Changing the Game. <https://techjury.net/blog/app-revenue-statistics/#gref>, 2021.
- [246] Greg Sterling. Mobile devices now driving 56 percent of traffic to top sites.
- [247] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *MoST '12*.
- [248] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15*, pages 127–140, New York, NY, USA, 2015. ACM.
- [249] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *ESEC/FSE '17*, 2017.
- [250] Jason Summerfield. Mobile website vs. mobile app: Which is best for your organization? <https://www.hswsolutions.com/services/mobile-web-development/mobile-website-vs-apps/>.

- [251] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks, WiSec '14*.
- [252] Mingshen Sun, Tao Wei, and John Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [253] Keen Sung, JianYi Huang, Mark D Corner, and Brian N Levine. Re-identification of mobile devices using real-time bidding advertising networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.
- [254] Mohammad Tahaei and Kami Vaniea. "developers are responsible": What ad networks tell developers about privacy. 2021.
- [255] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS '15*, 2015.
- [256] NEXD team. The Nexd Perspective: Gyroscope ads engage your audience. <https://www.nexd.com/blog/using-gyroscope-ads-to-better-engage-your-audience/>, 2020. Accessed: 2020-10-28.
- [257] Yuan Tian, Ying Chuan Liu, Amar Bhosale, Lin Shung Huang, Patrick Tague, and Collin Jackson. All your screens are belong to us: Attacks exploiting the html5 screen sharing api. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 34–48. IEEE, 2014.
- [258] Dennis Titze, Michael Lux, and Julian Schuette. Ordol: Obfuscation-resilient detection of libraries in android applications. In *Trustcom/BigDataSE/ICISS, 2017 IEEE*, pages 618–625. IEEE, 2017.
- [259] Debra Trampe, Diederik A. Stapel, and Frans W. Siero. The self-activation effect of advertisements: Ads can affect whether and how consumers think about the self. *Journal of Consumer Research*, 37(6):1030–1045, 10 2010.
- [260] Sipat Triukose, Sebastien Ardon, Anirban Mahanti, and Aaditeshwar Seth. Geolocating ip addresses in cellular data networks. In *Proceedings of the 13th International Conference on Passive and Active Measurement, PAM'12*.
- [261] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. A classification of web browser fingerprinting techniques. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pages 1–5. IEEE, 2015.

- [262] Pelayo Vallina, Alvaro Feal, Julien Gamba, Narseo Vallina-Rodriguez, and Antonio Fernandez Anta. Tales from the porn: A comprehensive privacy analysis of the web porn ecosystem. In *Proceedings of the 2019 Internet Measurement Conference*, 2019.
- [263] Rick Waldron. Generic Sensor API. <https://www.w3.org/TR/generic-sensor/>, 2019. Accessed: 2021-02-10.
- [264] Fabo Wang, Yuqing Zhang, Kai Wang, Peng Liu, and Wenjie Wang. Stay in your cage! a sound sandbox for third-party libraries on android. In *ESORICS 2016*.
- [265] Haoyu Wang, Jason Hong, and Yao Guo. Using text mining to infer the purpose of permission use in mobile apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '15*.
- [266] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. Understanding the purpose of permission use in mobile apps. *ACM Transactions on Information Systems, TOIS '17*.
- [267] Na Wang, Pamela Wisniewski, Heng Xu, and Jens Grossklags. Designing the default privacy settings for facebook applications. In *CSCW '14'*.
- [268] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. Strider typo-patrol: Discovery and analysis of systematic typo-squatting. *SRUTI*, 6:31–36, 2006.
- [269] Yong Wang, Daniel Burgener, Marcel Flores, Aleksandar Kuzmanovic, and Cheng Huang. Towards street-level client-independent ip geolocation. In *NSDI*, volume 11, pages 27–27, 2011.
- [270] Wardell Bagby. Sensor Disabler - This Xposed module allows you to modify and disable various sensors on your device. <https://github.com/wardellbagby/Sensor-Disabler>, 2021. Accessed: 2021-08-10.
- [271] Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. Routedetector: Sensor-based positioning system that exploits spatio-temporal regularity of human mobility. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.
- [272] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. 2018.
- [273] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium, USENIX Security '15*.

- [274] Robert Williams. Facebook's ad revenue rises 25% to record \$20.7B. <https://www.mobilemarketer.com/news/facebooks-ad-revenue-rises-25-to-record-207b/571362/>, 2020. Accessed: 2020-10-28.
- [275] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security NDSS '16*.
- [276] Yuanyi Wu, Dongyu Meng, and Hao Chen. Evaluating private modes in desktop and mobile browsers and their resistance to fingerprinting. In *Communications and Network Security (CNS), 2017 IEEE Conference on*, pages 1–9. IEEE, 2017.
- [277] Yuxi Wu, Panya Gupta, Miranda Wei, Yasemin Acar, Sascha Fahl, and Blase Ur. Your secrets are safe: How browsers' explanations impact misconceptions about private browsing mode. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 217–226. International World Wide Web Conferences Steering Committee, 2018.
- [278] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914, May 2015.
- [279] Chenren Xu, Sugang Li, Gang Liu, Yanyong Zhang, Emiliano Miluzzo, Yih-Farn Chen, Jun Li, and Bernhard Finner. Crowd++: Unsupervised speaker count with smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '13*, pages 43–52, New York, NY, USA, 2013. ACM.
- [280] Zhi Xu, Kun Bai, and Sencun Zhu. 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*, pages 113–124. ACM, 2012.
- [281] Guangliang Yang, Abner Mendoza, Jialong Zhang, and Guofei Gu. Precisely and scalably vetting javascript bridge in android hybrid apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 143–166. Springer, 2017.
- [282] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. 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*, pages 351–360, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

- [283] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proceedings of the 2012 USENIX ATC*.
- [284] Jiexin Zhang, Alastair Beresford, and Ian Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.
- [285] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC '13*.
- [286] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, 2013.
- [287] Yury Zhauniarovich and Olga Gadyatskaya. Small changes, big changes: an updated view on the android permission system. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2016.
- [288] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '12*.
- [289] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. Acoustic fingerprinting revisited: Generate stable device id stealthily with inaudible sound. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 429–440. ACM, 2014.
- [290] Tong Zhu, Qiang Ma, Shanfeng Zhang, and Yunhao Liu. Context-free attacks using keyboard acoustic emanations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 453–464, New York, NY, USA, 2014. ACM.
- [291] John Zulueta, Andrea Piscitello, Mladen Rasic, Rebecca Easter, Pallavi Babu, Scott A Langenecker, Melvin McInnis, Olusola Ajilore, Peter C Nelson, Kelly Ryan, et al. Predicting mood disturbance severity with mobile phone keystroke metadata: A biaf-fect digital phenotyping study. *Journal of medical Internet research*, 20(7), 2018.
- [292] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *CCS'17*, 2017.