“And all the pieces matter...”

Hybrid Testing Methods for Android App’s Privacy Analysis

by

Álvaro Feal

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Telematic Engineering

Universidad Carlos III de Madrid

Advisor: Narseo Vallina-Rodriguez

October 2022

"And all the pieces matter..."
Hybrid Testing Methods for Android App's Privacy Analysis

Prepared by:
Álvaro Feal, IMDEA Networks Institute, Universidad Carlos III de Madrid
contact: alvaro.feal@imdea.org

Under the advice of:
Narseo Vallina-Rodríguez, IMDEA Networks Institute / ICSI

This work has been supported by:

institute IMDEA networks

Unless otherwise indicated, the content of is thesis is distributed under a
Creative Commons Attribution - Non Commercial - Non Derivatices (CC BY-NC-ND).

cc BY NC ND

To the people that made this journey possible.

Acknowledgements

The works presented in this thesis were partially funded within the framework of the following projects and grants:

- European Union's Horizon 2020 Innovation Action program (Grant Agreement No. 786741, SMOOTH Project and Grant Agreement No. 101021377, TRUST AWARE Project).
- Spanish Government ODIO NºPID2019-111429RB-C21/PID2019-111429RBC22.
- The Spanish Data Protection Agency (AEPD)
- AppCensus Inc.

Published content

- Don't Accept Candy from Strangers: An Analysis of Third-Party Mobile SDKs
  Álvaro Feal, Julien Gamba, Juan Tapiador, Pirmal Wijesekera, Joel Reardon, Serge Egelman, Narseo Vallina-Rodriguez.
  In Data Protection and Privacy: Data Protection and Artificial Intelligence 2021, Bloomsbury Publishing.

  - My role: I was the leading actor in this publication, responsible for most of the writing as well as running the different experiments included in this work.
  - This paper is partially included in this thesis in chapters 2 and 5.
  - The material from this source included in this thesis is not singled out with typographic means and references.

- Angel or Devil? A Privacy Study of Mobile Parental Control Apps
  Álvaro Feal, Paolo Calciati, Narseo Vallina-Rodriguez, Carmela Troncoso, Alessandra Gorla.
  In Proceedings of Privacy Enhancing Technologies (PoPETS) 2020, July 14–18, virtual event
  This paper received the "Premio de Investigación en Protección de Datos Personales Emilio Aced" by the Spanish Data Protection Agency (AEPD) in 2020.

  - My role: As the leading actor, I was responsible for making sure all the different experiments were integrated correctly in the paper, as well as running most experiments (except those related to static analysis which are included in Dr. Paolo Calciati UPM's doctoral thesis as a contribution).
  - This paper is partially included in this thesis in chapter 3.
  - The material from this source included in this thesis is not singled out with typographic means and references.

- 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system
  Joel Reardon, **Álvaro Feal**, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Serge Egelman.
  In 28th USENIX Security Symposium (USENIX Security 19), Aug. 14–16, Santa Clara, CA, USA
  This paper received the "Distinguished Paper Award" in USENIX Security 19, the "Premio de Investigación en Protección de Datos Personales Emilio Aced" by the Spanish Data Protection Agency (AEPD) in 2021 and the "CNIL - Inria Data Protection Award" in 2021.

  - My role: My contribution in this work relates mostly to the dynamic analysis experiments, analyzing the raw results in look for potential privacy issues. I was also involved in the writing of the paper, specially so in the background and related work sections.
  - This paper is partly included in this thesis as chapter 4.
  - The material from this source included in this thesis is not singled out with typographic means and references.

Other research merits

The following list includes other research papers I have co-authored during the course of my PhD, but that are not included in this thesis.

- Mixed Signals: Analyzing Software Attribution Challenges in the Android Ecosystem
  Kaspar Hageman, **Álvaro Feal**, Julien Gamba, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, Narseo Vallina-Rodriguez.
  Submitted in IEEE Transactions on on Software Engineering. April 2022

- Not Your Average App: A Large-scale Privacy Analysis of Android Browsers
  Amogh Pradeep, **Álvaro Feal**, Julien Gamba, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, David Choffnes
  Accepted for publication in Proceedings of Privacy Enhancing Technologies Symposium (PoPETS). July 2023

- Blocklist Babel: On the Transparency and Dynamics of Open Source Blocklisting
  Álvaro Feal, Pelayo Vallina, Julien Gamba, Sergio Pastrana, Antonio Nappa, Oliver Hohlfeld, Narseo Vallina-Rodriguez and Juan Tapiador.
  In IEEE Transactions on Network and Service Management. April 2021

- Trouble Over-The-Air: An Analysis of FOTA Apps in the Android Ecosystem
  Eduardo Blázquez, Sergio Pastrana, **Álvaro Feal**, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez and Juan Tapiador.
  In IEEE Symposium on Security and Privacy 2021, May 23-27, virtual event

- Mis-shapes, Mistakes, Misfits: An Analysis of Domain Classification Services
  Pelayo Vallina-Rodriguez, Victor Le Pochat, **Álvaro Feal**, Marius Paraschiv, Julien Gamba, Tim Burke, Oliver Hohlfeld, Juan Tapiador and Narseo Vallina-Rodriguez.
  In ACM IMC 2020, Oct. 27 - 29, virtual event

- Understanding Incentivized Mobile App Installs on Google Play Store
  Shehroze Farooqi, **Álvaro Feal**, Tobias Lauinger, Damon McCoy, Zubair Shafiq, Narseo Vallina-Rodriguez.
  In ACM IMC 2020, Oct. 27 - 29, virtual event

- The Price is (Not) Right: Comparing Privacy in Free and Paid Apps
  Catherine Han, Irwin Reyes, **Álvaro Feal**, Joel Reardon, Primal Wijesekera, Narseo Vallina-Rodriguez, Amit Elazari Bar On, Kenneth Bamberger, Serge Egelman.
  In Proceedings of Privacy Enhancing Technologies (PoPETS) 2020, July 14–18, virtual event

- On The Ridiculousness of Notice and Consent: Contradictions in App Privacy Policies

Ehimare Okoyomon, Nikita Samarin, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, Irwin Reyes, **Álvaro Feal**, Serge Egelman.
In Workshop on Technology and Consumer Protection (ConPro) 2019, May 23, San Francisco, CA, USA

- Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps
  Catherine Han, Irwin Reyes, Amit Elazari Bar On, Joel Reardon, **Álvaro Feal**, Serge Egelman, Narseo Vallina-Rodriguez.
  In Workshop on Technology and Consumer Protection (ConPro) 2019, May 23, San Francisco, CA, USA

- Tales from the Porn: A Comprehensive Privacy Analysis of the Web Porn Ecosystem
  Pelayo Vallina-Rodriguez, **Álvaro Feal**, Julien Gamba, Narseo Vallina-Rodriguez and Antonio Fernández.
  In ACM IMC 2019, Oct. 21 - 23, Amsterdam, The Netherlands

Abstract

Smartphones have become inherent to the every day life of billions of people worldwide, and they are used to perform activities such as gaming, interacting with our peers or working. While extremely useful, smartphone apps also have drawbacks, as they can affect the security and privacy of users. Android devices hold a lot of personal data from users, including their social circles (e.g., contacts), usage patterns (e.g., app usage and visited websites) and their physical location. Like in most software products, Android apps often include third-party code (Software Development Kits or SDKs) to include functionality in the app without the need to develop it in-house. Android apps and third-party components embedded in them are often interested in accessing such data, as the online ecosystem is dominated by data-driven business models and revenue streams like advertising.

The research community has developed many methods and techniques for analyzing the privacy and security risks of mobile apps, mostly relying on two techniques: static code analysis and dynamic runtime analysis. Static analysis analyzes the code and other resources of an app to detect potential app behaviors. While this makes static analysis easier to scale, it has other drawbacks such as missing app behaviors when developers obfuscate the app's code to avoid scrutiny. Furthermore, since static analysis only shows potential app behavior, this needs to be confirmed as it can also report false positives due to dead or legacy code. Dynamic analysis analyzes the apps at runtime to provide actual evidence of their behavior. However, these techniques are harder to scale as they need to be run on an instrumented device to collect runtime data. Similarly, there is a need to stimulate the app, simulating real inputs to examine as many code-paths as possible. While there are some automatic techniques to generate synthetic inputs, they have been shown to be insufficient.

In this thesis, we explore the benefits of combining static and dynamic analysis techniques to complement each other and reduce their limitations. While most previous work has often relied on using these techniques in isolation, we combine their strengths in different and novel ways that allow us to further study different privacy issues on the Android ecosystem. Namely, we demonstrate the potential of combining these complementary methods to study three inter-related issues:

- A regulatory analysis of parental control apps. We use a novel methodology that relies on easy-to-scale static analysis techniques to pin-point potential privacy issues and violations of current legislation by Android apps and their embedded SDKs. We rely on the results from our static analysis to inform the way in which we manually exercise the apps, maximizing our ability to obtain real evidence of these misbehaviors. We study 46 publicly available apps and find instances of data collection and sharing without consent and insecure network transmissions containing personal data. We also see that these apps fail to properly disclose these practices in their privacy policy.
- A security analysis of the unauthorized access to permission-protected data without user consent. We use a novel technique that combines the strengths of static and dynamic analysis, by first comparing the data sent by applications at runtime with the permissions granted to each app in order to find instances of potential unauthorized access to permission protected data. Once we have discovered the apps that are accessing personal data without permission, we statically analyze their code in order to discover covert- and side-channels used by apps and

SDKs to circumvent the permission system. This methodology allows us to discover apps using the MAC address as a surrogate for location data, two SDKs using the external storage as a covert-channel to share unique identifiers and an app using picture metadata to gain unauthorized access to location data.

- A novel SDK detection methodology that relies on obtaining signals observed both in the app's code and static resources and during its runtime behavior. Then, we rely on a tree structure together with a confidence based system to accurately detect SDK presence without the need of any a priory knowledge and with the ability to discern whether a given SDK is part of legacy or dead code. We prove that this novel methodology can discover third-party SDKs with more accuracy than state-of-the-art tools both on a set of purpose-built ground-truth apps and on a dataset of 5k publicly available apps.

With these three case studies, we are able to highlight the benefits of combining static and dynamic analysis techniques for the study of the privacy and security guarantees and risks of Android apps and third-party SDKs. The use of these techniques in isolation would not have allowed us to deeply investigate these privacy issues, as we would lack the ability to provide real evidence of potential breaches of legislation, to pin-point the specific way in which apps are leveraging cover and side channels to break Android's permission system or we would be unable to adapt to an ever-changing ecosystem of Android third-party companies.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Privacy Risks in Android

Android is the leading mobile operative System (OS) with over 3 Billion active devices as of May 2021 [268]. Users access functionality in their phones through the use of applications (apps) that offer a myriad of different functionalities and purposes. However, Android apps can cause privacy and security damage to end users as they host a wide range of personal and sensitive data, including behavioral data about their usage of the phone (i.e., app usage, visited webpages), their whereabouts (i.e., nearby devices, location of the phone) and their identity (i.e., unique identifiers, e-mail addresses).

Different actors in the Android ecosystem might be interested on accessing such data for their own purposes. Some of these actors might be malicious, such as malware developers that might distribute their products through Android apps. In order to avoid abusive behavior by these actors, the Google Play Store—the number one public repository for Android apps—implements its own protections [20] that attempt to automatically detect malware and other apps that might be dangerous for the security and privacy of their users. However, not all app markets deploy these security methods, and even those that do lack the ability to detect every instance malicious or dangerous behaviors by Android apps. However, most actors in the Android ecosystem are not malicious, but rather part of an industry that relies on data-driven business models, such as advertising, to generate revenue through user's personal data [55]. These companies are present in Android apps via third-party libraries or SDKs. In fact, Software Development Kits (SDKs or third-party libraries) are often embedded in app's code [187, 229, 100] to include different features such as UI elements, network protocols or tracking capabilities. Third-party SDKs have desirable features, such as enabling code re-use for a given functionality which makes it easier to spot bugs and errors since they are used by large number of developers [148]. However, the use of SDKs can also lead to third-party companies with an interest on accessing personal data having control of certain parts of the app's behavior which can become problematic for the privacy of users.

From a data protection and regulatory perspective, the way in which online services can access personal data from users is regulated in Europe by the General Data Protection Regulation (GDPR [77]) which became effective in May 2018. This legislation defines several rights that are common to all European users, including the right to access their data, modify or remove such data, and oppose to the collection of data altogether. The most common legal basis for data collection is that users consent to the collection of their personal data in order for apps to access (and share) it. Therefore, if a given app (or an embedded library) is collecting a given piece of data, this information should be clearly disclosed on its privacy policy. However, in the four years in which GDPR has been effective, there have been many examples of companies failing to adhere to these provisions and behaving in ways detrimental to users privacy [212, 193, 210]. In California, the California Consumer Protection Act (CCPA [261]) has similar provisions to protect the privacy of California residents. Also in the US, the Children Online Privacy Protection Act (COPPA [101]) applies specifically to all children below the age of 13. This law includes special provisions, such as the need for verifiable parental consent to collect personal data from children. Special provisions for vulnerable populations are also present in the GDPR, affecting children under 13-16 years of age (the threshold depends on the member state).

The Android operative system protects access to user's personal data via its permission system. This system also works based on consent, as when a given app wants to access a piece of protected data, then it must show a warning to the user that must be approved [27] (at install time for "normal" permissions and during runtime for "dangerous" permissions). However, the Android permission model is not perfect. While this system is based on consent, there is a clear lack of transparency as the user does not know what is the purpose for which a given permission. This is worsened by the fact that SDKs inherit all of the permissions granted to the "host" app and so they can piggyback on the permissions granted to the app to access protected data without user knowledge. This hinders the ability of users to provide informed consent, as they lack information about whether a given piece of data is collected for the primary purpose of the service (e.g., the location on a navigation app) or for secondary purposes (e.g., the location by an advertisement SDK). Finally, previous work has also shown that it is possible to take advantage of faults on its design and implementation to gain access to protected resources without user consent [107, 294].

Many studies have analyzed the security and privacy impact of apps and third-party SDKs on users, finding issues in apps regardless of their origin [116, 286, 60] and functionality [76, 238, 215, 236, 283]. In most cases, the issues arise from the collection and sharing of user's personal data in potentially dangerous ways [251, 178, 236, 229, 238, 99, 283]. In fact, several studies have shown different ways in which Android apps did not comply with different privacy legislation [238, 96, 208], and that is common to find contradictions between the behavior of an app and its privacy policy [212, 166]. Similarly, prior work has also show that some SDKs can be classified as malware due to their malicious behavior [181, 291]. There is also a vast literature in terms of SDK detection tools [146, 252, 263, 74, 91, 187, 48]. However, most research efforts have limitations as they often rely either on the static analysis of the app's code and resources or the dynamic analysis of the app's runtime behavior. When used in isolation, both of these techniques have their own shortcomings, preventing for a complete analysis of the privacy and security risks of Android apps at scale:

- **Static analysis** relies on extracting features from the code and other elements of an app, without the need of executing it. Therefore, static analysis techniques are often easier to scale, but they are prone to miss app behaviors and since app's code is often obfuscated [292, 33]. Developers rely on obfuscation techniques to make it difficult to study the code, for instance, by changing class names or adding dummy code. Furthermore, static analysis techniques are unable to analyze code that is dynamically loaded during runtime, potentially missing some of the app's behaviors. Many of these tools also have the inability to distinguish dead or legacy code (i.e., code that is conspicuously present but will never execute [146, 252, 263, 74]), leading to potential over-reporting of behaviors that are never executed in reality. Therefore, all observations made through static analysis provide a higher bound on potential app behavior.
- **Dynamic analysis** methods allow to obtain real evidence of the app's runtime behavior, it also has its drawbacks when used in isolation. Dynamic analysis suffers from scalability issues due to the need of running it on an actual device. One solution to the issue could be the use of

emulators, but most software nowadays include anti-testing techniques to modify their behavior if they discover that they are being executed on a test environment [285, 31]. On the other hand, dynamic analysis shows only a lower bound of the app's behavior, as it is difficult to cover all of the possible behaviors of an app during testing. Exercising the app (a process called *fuzzing*) is very complex, specially in modern software that has many different code-paths some of which might be behind login screens [75, 196, 288].

In this thesis, we explore the benefits of building methodologies that us static and dynamic analysis in combination, in ways in which they complement their limitations and allow for more accurate, sound and scalable analysis of privacy and security issues in Android apps.

1.2 A Hybrid Approach to Android Privacy Analysis

We argue that **static and dynamic analysis techniques should be used in combination**, complementing each other's drawbacks and limitations, and allowing for sound, scalable and reliable privacy and security analysis of the Android ecosystem. Namely, static analysis can provide scalability, providing a higher-bound of potential privacy issues while dynamic analysis can then more accurately explore these potential issues and show actual evidence of their presence on a given app. In this thesis we set out to test the following hypothesis:

Hypothesis 1 (H1): *Static and dynamic analysis can be used in conjunction to complement each other, allowing for more scalable, complete and sound privacy analysis techniques for Android apps.*

We test this hypothesis by developing new hybrid methodologies to study three inter-related issues: i) a regulatory compliance analysis of the access to children's sensitive data by parental control apps and their embedded third-party SDKs, ii) a security analysis of the ability of apps and SDKs to circumvent Android's permission system to gain unauthorized access to personal data and protected resources, and iii) a new methodology to detect and characterize SDK presence in apps at scale without any a-priori knowledge.

1.2.1 Case One: Regulatory Compliance Analysis

The dependency of society on mobile services to perform daily activities has drastically increased in the last decade [262]. Children are no exception to this trend. Just in the UK, 47% of children between 8 and 11 years of age have their own tablet, and 35% own a smartphone [211]. Unfortunately, the Internet is home for a vast amount of content potentially harmful for children, such as uncensored sexual imagery [58], violent content [298], and strong language [158], which is easily accessible to minors. Furthermore, children may (unawarely) expose sensitive data online that could eventually fall in the hands of predators [8], or that could trigger conflicts with their peers [80]. Aiming at safeguarding children's digital life, some parents turn to *parental control applications* to monitor children's activities and to control what they can do on their smartphones [199, 254]. The typical parental control app allows parents to filter, monitor, or restrict communications, content, system features, and app's execution [296]. Other apps provide parents with fine-grained reports about children's usage of the phone, their social interactions, and their physical location.

To provide these features, parental control apps rely on the collection and handling of children's behavioral (e.g., location, browsing activities, or phone calls) and personal data (e.g., unique identifiers or contacts), in many cases, using techniques and methods similar to those of spyware [73]. Yet, as with many regular Android applications, parental control software may also integrate data-hungry third-party advertising SDKs to monetize their software, and analytics SDKs to monitor the behavior of their users, create bug reports, and build user profiles. As a consequence of Android's permission model, these SDKs enjoy the same set of permissions that are granted by the user to the host app. Apps also might inadvertently expose data to in-path network observers if they use insecure protocols to transmit sensitive data. These practices involve great privacy risks for minors, e.g., if data is used to profile children's behavior or development, or if the data becomes compromised [202].

To help parents choose among parental control solutions, security centers at the European level [89, 157] have analyzed a number of parental control solutions. Their analysis considers four dimensions: functionality, effectiveness, usability, and security—defined as their effectiveness at deterring children from bypassing the system. However, these reports *do not provide any privacy risk analysis*, nor consider the lack of transparency or other potential violations of relevant privacy laws with specific provisions to safeguard minors' privacy, e.g., the European General Data Protection Regulation (GDPR) [77] and the U.S. Children Online Privacy and Protection Act (COPPA) [101].

In this case study, we use a combination of static and dynamic analysis to present the first comprehensive privacy-oriented analysis of parental control apps for Android from a technical and regulatory point of view. We use static analysis of the app's code to better understand the potential privacy issues of these apps by analyzing their permission and third-party SDK usage. Then, we extract potentially dangerous behavior using flow analysis and use it to inform the inputs that we use to exercise the apps in order to analyze their runtime behavior. By analyzing the traffic generated during runtime, we are able to obtain actual evidence about the data collection and sharing practices of these apps. By putting these results in context with their privacy policies, we can better understand whether the apps obtain appropriate parental consent.

1.2.2 Case Two: Security Analysis

Android implements a permission-based system to regulate access to these sensitive resources by third-party applications. In this model, app developers must explicitly request *permission* in their Android Manifest file [25] to access sensitive resources. This model is supposed to give users control in deciding which apps can access which resources and information; in practice it does not address the issue completely [107, 294].

The Android operating system sandboxes user-space apps to prevent them from interacting arbitrarily with other running apps. Android implements isolation by assigning each app a separate *user ID* and further mandatory access controls are implemented using SELinux. Each running process of an app can be either code from the app itself or from SDK libraries embedded within the app; these SDKs can come from Android (e.g., official Android support libraries) or from third-party providers. As previously discussed, any third-party service bundled in an Android app inherits access to all permission-protected resources that the user grants to the app. In other words, if an app can access the user's location, then all third-party services embedded in that app can as well.

In practice, security mechanisms can often be circumvented; *side channels* and *covert channels* are two common techniques to circumvent a security mechanism [1]. These channels occur when there is an alternate means to access a protected resource that is not audited by the security mechanism, thus leaving the resource unprotected. A *side channel* exposes a path to a resource that is outside the security mechanism; this can be because of a flaw in the design of the security mechanism or a flaw in the implementation of the design. A classic example of a side channel is that power usage of hardware when performing cryptographic operations can leak the particulars of a secret key [172]. As an example in the physical world, the frequency of pizza deliveries to government buildings may leak information about political crises [245]. A *covert channel* is a more deliberate and intentional effort between two cooperating entities so that one with access to some data provides it to the other entity without access to the data in violation of the security mechanism [175]. As an example, someone could execute an algorithm that alternates between high and low CPU load to pass a binary message to another party observing the CPU load.

The research community has previously explored the potential for covert channels in Android using local sockets and shared storage [191], as well as other unorthodox means, such as vibrations and accelerometer data to send and receive data between two coordinated apps [7]. Examples of side channels include using device sensors to infer the gender of the user [195] or uniquely identify the user [257]. More recently, researchers demonstrated a new permission-less device fingerprinting technique that allows tracking Android and iOS devices across the Internet by using factory-set sensor calibration details [302]. However, there has been little research in detecting and measuring at scale the prevalence of covert and side channels in apps that are available in the Google Play Store. Only isolated instances of malicious apps or libraries inferring users' locations from WiFi access points were reported, a side channel that was abused in practice and resulted in about a million dollar fine by regulators [103].

In fact, most of the existing literature is focused on understanding personal data collection using the system-supported access control mechanisms (i.e., Android permissions). With increased regulatory attention to data privacy and issues surrounding user consent, we believe it is imperative to understand the effectiveness (and limitations) of the permission system and whether it is being circumvented as a preliminary step towards implementing effective defenses.

To this end, we extend the state of the art by developing new methods to detect actual circumvention of the Android permission system, at scale in real apps by using a combination of dynamic and static analysis. In short, we ran apps to see when permission-protected data was transmitted by the device, and scanned the apps to see which ones *should not* have been able to access the transmitted data due to a lack of granted permissions. We grouped our findings by *where* on the Internet *what* data type was sent, as this allows us to attribute the observations to the actual app developer or embedded third-party libraries. We then use static analysis of the responsible component's code to determine exactly how the data was accessed. Finally, we also statically analyzed our entire dataset to measure the prevalence of the channel. We focus on a subset of the *dangerous* permissions that prevent apps from accessing location data and identifiers. Instead of imagining new channels, our work focuses on tracing evidence that suggests that side- and covert-channel abuse is occurring in practice.

[1] For a more complete definition of these attacks, we refer the reader to Chapter 4

1.2.3 Case Three: SDK Detection Methodology

Mobile app developers rely on third-party Software Development Kits (SDKs) offered by specialized advertising and tracking companies [188, 230]. In general, the use of third-party SDKs by mobile app developers can be perceived as a good software engineering practice. In fact, SDKs provide desirable properties such as code reuse and extensive testing. However, while some SDKs can provide valuable functionality to app developers such as cryptographic functions or graphic engines, others are known to be detrimental to user's privacy [230] by collecting sensitive user data without user awareness.

However, the way in which the Android permission model, opens the door to SDKs to exploit it and generate potential privacy and transparency issues for the user. As previously explained in this thesis, SDKs run with the same user ID and privileges as the host app. As a result, embedded SDKs can access all the permissions granted by the user to the host app. This can lead to cases in which and SDK leverages a permission granted to the app for secondary purposes, potentially without user consent (i.e., permission escalation). In other cases, SDKs might require app developers to request more permissions that would not be needed to offer the primary service of the app, resulting in apps being over-privileged [183, 85].

Therefore, the ability to accurately attribute a given behavior to either the app itself or an embedded SDK plays a mayor role in the privacy analysis of the Android apps. This distinction is critical for many applications such as (i) discerning whether personal data is collected for primary or secondary purposes, (ii) assessing the regulatory compliance of applications, and the completeness of their privacy policies, and (iii) being able to quickly respond to a vulnerability caused by an app's component that can harm user's security. In fact, modern privacy legislation like the GDPR [77] operate under the principle of purpose limitation, which dictates that data processing and consent are bound by the purpose for which it was originally collected.

The research community has produced several tools that allow scholars, regulators and security practitioners to identify SDKs embedded on mobile apps. Most of the available tools like Exodus[91], LibRadar [187] and LibScout[48] rely on static analysis of the app's code. Specifically, they rely on code fingerprints extracted from existing SDKs to detect their presence. However, approaches based on static fingerprints suffer from serious coverage limitations. These methods need to extract and compile fingerprints for new SDKs and releases, many of which are not publicly available. This is problematic in an ever-changing ecosystem, where new SDKs are constantly released and smaller players are often acquired and merged into SDKs from bigger companies due to commercial agreements [32, 162]. Moreover, static analysis can fail to detect SDKs due to dynamically loaded or obfuscated code, which hinders their ability to extract static fingerprints on obfuscated code [292, 33]. In fact, previous work has reported the inability of many SDK detection tools to accurately detect SDKs in the presence of obfuscators and packers [303]. Code-bases solutions might also report false positives, as they often times lack the ability to discard SDKs which are present in the app due to dead or legacy code and that is never executed at run-time [146, 252, 263, 74]).

To overcome these issues, some researchers turned to dynamic analysis methods to identify the presence of SDKs (and their data collection practices) by analyzing the runtime behavior of the app [231, 256, 235, 215, 168, 99]. While dynamic analysis solves some of the aforementioned static analysis issues, it suffers from its own limitations. On the one hand, dynamic analysis is harder to scale because of the need to actually run the app on a test device (e.g., a sandbox). Most of these solutions are useful for the privacy analysis of Android apps as they rely on the analysis of app's traffic to understand data collection and sharing practices. However, from an SDK detection point of view, this is not enough as the fact that a given app contacts a domain related to and SDK does not always mean that this SDK is indeed present in the app. Furthermore, dynamic analysis is prone to miss app behaviors and SDK presence as the ability to fully test an app is severely limited by the presence of pay walls and login screens, and the difficulty to automatically drive Android apps testing [75, 288]. Furthermore, some apps have anti-testing methods in place, that aim to change the app's behavior if they observe that they might be under scrutiny [31].

In this thesis, we propose a new methodology for SDK detection that combines both static and dynamic analysis, to complement each other's limitation. This thesis presents LibSeeker, a novel hybrid approach for SDK detection and analysis that does not rely on static profiles pre-generated from SDKs prevalent in the market at a given time. Instead, LibSeeker relies on automatically identifying third-party components and behaviors from a set of signals obtained both by analyzing static components of the app and its runtime behavior. This new paradigm allows avoiding the arms race between detection tools and the ever-changing ecosystem of third parties and advertisement and tracking libraries.

1.3 Contributions

In this thesis, we rely on a combination of static and dynamic analysis methods in a uniquer and novel way to present several contributions to the field of Android privacy analysis:

Parental control apps. We conduct the first in-depth study of the Android parental control app's ecosystem from a privacy and regulatory point of view. To that end, we employ a novel technique that combines the static analysis of app's code to detect potential privacy mis-behaviors in apps and uses it to drive the dynamic analysis of apps in order to find actual evidence of such privacy issues. We exhaustively study 46 apps from 43 developers which have a combined 20M installs in the Google Play Store and find that: these apps are on average more permissions-hungry than the top 150 apps in the Google Play Store, and tend to request more dangerous permissions with new releases; 11% of the apps transmit personal data in the clear; 34% of the apps gather and send personal information without appropriate consent; and 72% of the apps share data with third parties (including online advertising and analytics services) without mentioning their presence in their privacy policies. In summary, parental control applications lack transparency and lack compliance with regulatory requirements. This holds true even for those applications recommended by European and other national security centers [99]. We responsibly disclosed our findings to the Spanish Data Protection Agency (AEPD) as well as to national cybersecurity centers (i.e., IS4K from INCIBE).

Covert and side channels. We rely on our testing infrastructure to run thousands of apps in an instrumented environment, monitoring apps' runtime behavior and network traffic looking for evidence of side- and covert-channels being used in real-world Android apps. Our novel technique combines static and dynamic analysis, using dynamic app testing to look for sensitive data being sent over the network for which the sending app did not have permissions to access. We then rely on static code analysis of the apps and third-party SDKs responsible for this behavior to determine how the unauthorized access occurred. We also use software fingerprinting methods to measure the static prevalence of the technique that we discover among other apps in our corpus. Using this testing environment and method, we uncovered a number of side and covert channels in active use by hundreds of popular apps and third-party SDKs to obtain unauthorized access to both unique identifiers as well as geolocation data [233]. We tested our pipeline on more than 88,000 apps and discovered 5 apps exploiting the MAC addresses of the connected WiFi base stations from the ARP cache as a surrogate for location data; two Chinese SDKs—Baidu and Salmonads—independently make use of the SD card as a covert channel in 13 apps; and finally, one app that used picture metadata as a side channel to access precise location. We responsibly disclosed our findings to Google and the U.S. Federal Trade Commission (FTC), and received a bug bounty for our efforts.

SDK Detection. We present LibSeeker, a SDK detection tool based on a hybrid methodology that can accurately detect third-party code in apps without over-reporting due to unused SDKs and without the need of a-priori knowledge about an SDK to be able to detect it. To that end, we rely on a pool of signals extracted using a combination of static code analysis and dynamic runtime behavior analysis, which we then combine to compute a confidence score that allows us to accurately discard SDKs that are not used by the app. We compare LibSeeker to three state-of-the-art tools, namely LibRadar, LibScout and Exodus using a set of ground-truth apps as well as a dataset of publicly available apps. We show that LibSeeker improves the state-of-the art by: 1) detecting third-party code that other solutions miss because of the lack of library profiles; 2) improving the robustness against advanced obfuscation thanks to the use of both static and dynamic analysis signals; and 3) reducing the over-reporting of SDK presence by analyzing whether an SDK is part of dead or unused code. We also highlight other ways in which our methodology can be useful for privacy research. Namely, we present a study of the prevalence of SDKs in publicly available apps, in which we show clear differences between all detection tools. We also include a study on the use of permissions by third-party code, showing that SDKs often times leverage the set of permissions granted to the app to access protected data for secondary purposes.

Chapter 2 Background

Android is an open-source system based on the Linux kernel and perhaps its most glaring feature is its openness. Any developer can publish their app on the public Google Play Store, provided that: 1) it pays the 25$ fee for a developer account [153]; and 2) the app passes the automatic checks done by Google Play Protect. This is Android's threat protection service, which claims to use machine-learning mechanisms to detect apps with harmful behaviors [20]. While previous work found malware to be present on Google Play [291, 181], Play Protect has evolved over the years to detect more malicious apps, privacy abuses and violations of Android's publication policies. While Google Play is the most commonly used market for Android apps, they can come from other alternative sources such as third-party stores (e.g., Amazon, Baidu, F-Droid or APKMirror [203, 286]) and even pre-loaded on the phone at some point of the supply chain [116]. For instance, Google Play does not operate on the Chinese market, meaning that residents of this country must download apps from a different app store. Therefore, different stores and origins serve their own unique purposes, and they each might (or may not) have different security mechanisms in place.

Android phones portray a lot of information regarding the user, including unique identifiers [18], their physical location [68] and their usage behaviors and preferences [59] (i.e., used apps or visited webpages). This information is highly valuable to different actors, including malware developers that might target Android users with faulty programs and malicious attacks and state actors that might have an interest on accessing large pools of data from mobile phone users. Furthermore, most companies that take part on the Android ecosystem have data-driven business models in which personal data from users is highly coveted as it allows them to generate revenue (e.g., through advertisement and tracking [230]). For instance, the Ad-tech ecosystem is highly profitable (both in mobile and on the web) and it is forecasted to drive 1 Trillion$ in 2030 [5]. In most cases, Android developers rely on third-party libraries or SDKs to provide advertisement and tracking capabilities. One example of companies interested on collecting personal data from users, can be targeted advertisements. Companies collect user data in order to build complete profiles about their preferences, which allows them to show highly targeted ads that are more likely to lead to interactions from the user [170]. This interest for user's personal data has led to proliferation of privacy misbehavior and aggressive data collection practices in Android apps regardless of their origin [286, 115, 60, 213], price [251, 149, 176] purpose [229, 236, 253, 215, 263] or target audience [182, 280, 238, 99, 9].

In this chapter, we present current privacy regulation and discuss how it can protect Android user's privacy (§ 2.1), we dive into Android's permission model and its shortcomings (§ 2.2), we present the ecosystem of third-party libraries in Android and how they can affect user's privacy (§ 2.3) and finally we present the advantages and limitations of current analysis methods (§ refsec:analysis-methods).

2.1 Regulatory Frameworks

In the past years, there has been a compelling effort across different parts of the world to come up with regulations to govern how personal data can be collected, used and shared. While the applicable law depends on the residence and nature of the data subject, most of them share a common background which is the notion of notice and consent. Namely, the user has the right to be informed about what type of data will be collected and what for, and then she can decide to grant access or refuse it.

The EU General Data Protection Regulation (GDPR [77]) regulates the way in which personal data from European citizens can be accessed, processed, and shared. All European users have the right to be informed about data collection practices by organizations of all kinds and types, including digital services like websites and mobile apps and—unless grounds such as legitimate interest exist—no data collection should be allowed before the user has granted explicit consent. Generally, users turn to the app's privacy policy to make an informed decision about whether to consent to data collection. However, previous work has found that many of these policies are missing, incomplete, or too difficult for the average user to grasp [212, 164, 216]. Other than the right to refuse to data collection, the GDPR provides other rights to users, including (but not limited to) the right to access, erase or modify their data [156]. Finally, Article 32 [120] is also relevant for this thesis, as it states that "the processor shall implement appropriate technical and organizational measures to ensure a level of security appropriate to the risk". We argue that for data sharing between apps and third-party services like analytics or advertising providers through their SDKs, this means that network connections should have a minimum level of security (i.e., the use of encryption).

Prior work has looked at different aspects of how regulation is adopted by online services, as well as to potential violations. One important topic that has been widely analyzed is the correctness and fairness of consent banners for data collection practices and cookies [282, 193, 210]. Consent banners are the way in which websites communicate their data collection practices and their use of cookies (i.e., a web mechanism to track users behavior on a given website). These studies conclude that consent banners are often designed and presented in a way that is confusing to the user, in an attempt to get users to consent to their data collection practices. Kollnig et al. study the changes in the tracking ecosystem of Android apps after GDPR came into effect, showing that there are little differences in the number and types of trackers before and after GDPR [174]. They analyze the *.dex files of over 1M apps in search for URLs and compare the number of tracking domains found before and after GDPR, concluding that the number of contacted hosts (and number of companies behind them) remained stable after new legislation came into place. This results are interesting, as previous work has shown that GDPR did have an effect on the tracking ecosystem on the web, with the number of tracking cookies significantly reducing after GDPR [81].

2.1.1 Vulnerable Populations

The GDPR has special provisions for protecting the privacy of children under the age of 13-16 (the actual age varies across different state members). In the US, the COPPA [101] regulates data collection practices for minors under the age of 13. Both rules require the app to gather verifiable parental consent before collecting any personal or behavioral data from children. This is a specific type of consent in which the parent must prove their identity, in a way that can be verified by the party receiving consent. For instance, parents might need to provide extra information such as a credit card or a national ID card. There are different ways in which SDKs handle these special provisions. Some libraries directly state in their Terms of Service (ToS) that they are not suitable to be used by apps targeting a children audience [238], while others integrate switches to adapt their behavior when the developer states that the application is directed at children. There are several resources available for developers to choose libraries that respect legislation specific to children data. One remarkable example is Google's list of self-certified suitable for children libraries [128]. Unfortunately, it has been proven that self-certification does not guarantee that SDKs are indeed complying with current legislation without external auditing and enforcement [238]. Likewise, Apple provides recommendations for developers of applications that collect, store and process children data. Apple recommends that these applications avoid including third-party analytics and advertisement SDKS. If this were not possible, the developer must ensure that that embedded SDKs comply with any applicable privacy laws [42]. In fact, Google Play has a special category, called the Designed for Families (DFF) program, which includes apps that target children and thus should be compliant with COPPA and GDPR provisions for children data usage [140].

Others have studied privacy implications for children in domains such as social media [189] and smart toys [280, 297]. Reyes et al. [238] is the closest work to the regulatory compliance presented in this thesis (§ 3), as they use dynamic analysis to make an assessment of mobile apps' compliance with the COPPA regulation. They analyzed over 5k apps and show that the majority of them are in potential violation of these rule, including 19% embedding SDKs that prohibit their use in apps targeted at children. In the context of parental solutions specifically, there have been studies on their effectiveness from a non-technical perspective: Mathiesen argued that such policies are a violation of children's right to privacy [192] while Wisniewski et al. presented a qualitative feature analysis of 75 parental control apps. They identified 42 unique features offered by these apps and concluded that privacy invasive monitoring features are more common than those focusing on teen self regulation [296]. Furthermore, Eastin et al. showed that parenting style has an effect on the type of restrictions that parents set on their children's phone usage [87]. Chatterjee et al. also studied the parental control ecosystem. However, they focus on assessing how such applications may be used for other purposes (e.g., spyware and partner violence) [73]. The authors identify instances in which users posted public comments on app stores about using a specific parental control app to spy on their partner.

2.2 Android Permission Model

As we have previously discussed, Android protects unauthorized access to sensitive data and resources through its permission system. This system is based on the security principle of *least privilege*. That is, an entity should only have the minimum capabilities it needs to perform its task. This standard design principle for security implies that if an app acts maliciously, the damage will be limited. Developers must declare the permissions that their apps need beforehand, and the user is given an opportunity to review them and decide whether to install the app. The Android platform, however, does not judge whether the set of requested permissions are all strictly necessary for the app to function. Developers are free to request more permissions than they actually need and users are expected to judge if they are reasonable. Furthermore, the permission system has evolved with time, adding new policies for some permissions [29] and including new permissions while removing others.

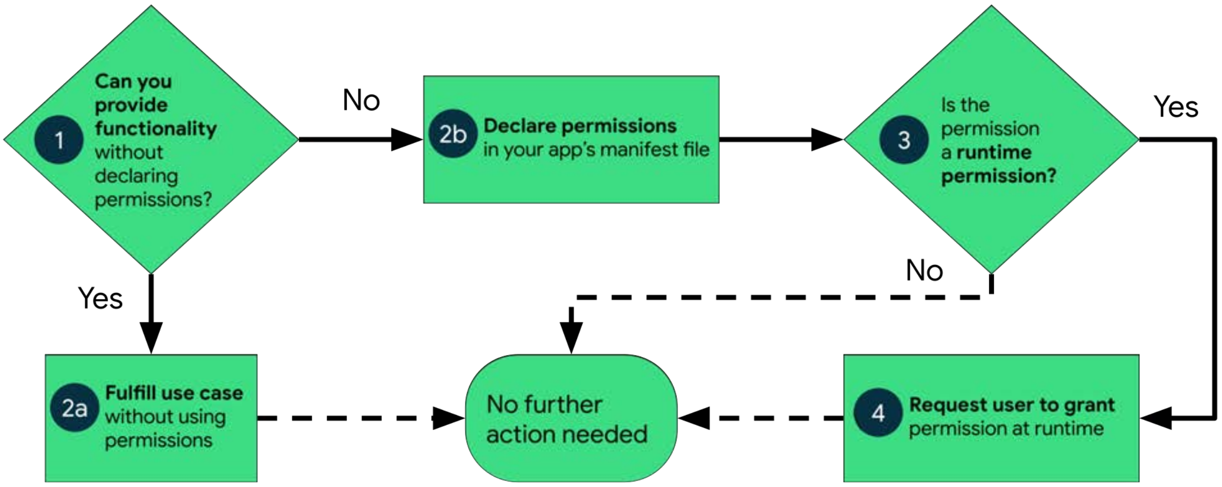

1
Can you provide functionality without declaring permissions?
No
2b
Declare permissions in your app's manifest file
3
Is the permission a runtime permission?
Yes
No
Yes
2a
Fulfill use case without using permissions
No further action needed
4
Request user to grant permission at runtime

Figure 1: High-level workflow for using permission in Android

This can lead to apps keeping permissions that are no longer needed because of legacy code [106].

The Android permission model has two important objectives: obtaining user consent before an app is able to access any of its requested permission-protected resources, and then ensuring that the app cannot access resources for which the user has not granted consent. The permission system has evolved with time and, since Android 6, this verification happens in runtime (for those permissions considered *dangerous* for Android). Whenever an app needs access to a permission protected resource, the app must check if the permission has been granted. If affirmative, then the app can continue its normal execution and gets access to the protected resource. Otherwise, the app can show a pop-up requesting access to the permission and the user has the ability to grant access or reject it. Figure 1 (obtained from Android's official documentation) shows the high-level workflow of using permissions in Android.

As we have briefly introduced, permissions have an associated *protection level* that relates to its implied potential risk. This, however, affects the procedure that the operating system follows to determine whether or not to grant a given permission to a requesting app:

- Permissions with the *normal* protection level are considered not to pose much risk to the user's privacy or the device's security, and are automatically granted at installation time.
- Permissions with a *signature* protection level will also be granted by the system at installation time, but only if the app requesting the permission is signed with the same certificate as the app defining it.
- *Dangerous* permissions protect resources that are considered sensitive (e.g., the device's location) and therefore require explicit user approval. Permissions with this protection level are granted at runtime since Android 6.

There are a number of permissions that fall outside of the Android-defined ones, known as *custom* permissions. Developers can define (or expose) their own permissions to enable controlled access to their own components and features, allowing them to expand Android's the permission system. This facilitates *"regulated"* programmatic inter-app communication and data sharing, despite each app running with a different UID. These permissions can be declared with any of the previously described protection levels.

Therefore, apps should not be able to access data permission-protected data if they do not hold such permission or if the permission has been denied [276, 179]. In practice, this is not always the case. Furthermore, there is a long line of work uncovering issues on how the permission model interacts with the user: users are inadequately informed about why apps need permissions at installation time, users misunderstand exactly what the purpose of different permissions are, and users lack context and transparency into how apps will ultimately use their granted permissions [107, 270, 294, 179]. While all of these are critical issues that need attention, the focus of the security analysis presented in this thesis (§ 4) is to understand how apps are circumventing system checks to verify that apps have been granted various permissions.

2.3 Third-party SDKs

Most software products (including Android apps), rely on code from third parties to include functionality without the need to develop it in-house. This has clear advantages for app developers, as it helps them improve their products at a reduced cost (both monetary and time-wise) is often perceived as a reflection on software engineering best practices [48, 62, 198, 239]. However, SDK presence can also have drawbacks for the privacy of users as many of the companies that offer these third-party libraries (or SDKs) have data-driven business models that rely on the collection of personal data (e.g., advertisement and tracking of users).

Previous work has shown the presence of third-party libraries in all kind of applications [230] regardless of their audience [238, 280], origin [115, 286] or price [149]. The use of SDKs in the mobile software supply chain is so extended that even apps that come preinstalled on the phone are packaged with third-party advertising and analytics libraries [115]. In terms of advertisement and tracking, Android applications contact on average six domains related to these data-driven purposes, typically offered by different companies. [230].

App developers might include third-party code due to its functionality, without realizing that this SDK might be harmful for users' privacy. The fact that boundaries between many SDK categories are unclear, and that SDK providers tend to offer more than one product to app developers can further confuse developers. For instance, analytics and advertising services have become extremely entangled since most Ad-Tech companies integrate both functionalities in the same SDK, potentially using the data gathered by the analytics service for user profiling or advertising [239].

With new privacy regulation, it is extremely important that developers have a clear understanding about the behavior of a given SDK has before including it on their app. In most cases, app publishers are responsible for informing users on the privacy policy about the presence of third-party libraries, the type of personal data that they collect and how they will treat that data. This approach has limitations, as SDKs often times operate as a black box and the lack of understanding about their behavior by app developers can lead to incomplete policies. This is dangerous for the privacy of users, but also for developers which could be liable for any privacy malpractice inflicted by third-party SDKs present in their products. The SDK itself is only responsible if the host application has nothing to do with the data collection process of the SDK. One example of this situation would be a third-party SDK that collects data in an application and uses it for different purposes than those originally intended, thus deciding the objectives and means of processing [237].

2.3.1 Types of SDKs

There are a broad range of third-party SDK providers that specialize in offering one or multiple features, services, or technologies to application developers. The type of services they offer range from SDKs offering UI support, to SDKs that collect user data in order to generate revenue. To illuminate this ecosystem, we present a purpose made taxonomy that extends that of previous work (namely Libradar [188]) thanks to a manual effort that spans years of Android privacy research [230, 99, 60]. Namely, we rely on publicly available information from SDKs found in different Android apps in order to to present a comprehensive classification of mobile SDKs by their offered functionality:

- **Development Support:** These are libraries which help developers adding support features to their code, such as widgets, UI features or JavaScript Object Notation (JSON) and XML formatters. Examples of this category include the Android Support Library and GSON (Google's JSON implementation). These libraries are expected to be found in many applications and, assuming that they have not been tampered with to include malicious code, they should be harmless to users' privacy since they do not collect personal data. Therefore, a-priori they do not need to be included in documents such as the privacy policy. Nevertheless, some development SDKs might engage in personal data collection, like Google's Firebase [127] and Unity3D [274], a library that supports the development of games but also includes analytics and advertisement capabilities. In this case, their ability to collect sensitive data will vary from one application to another, depending on how application developers integrate these services in their mobile products. It is possible to identify multiple subcategories of development support libraries, depending on their intended purpose:
  - *Networking and protocol support:* These libraries offer support for implementing network protocols such as HTTP/HTTPS or Google's QUIC.
  - *Database support:* These SDKs provide developers with code to manage and store data, implementing well known database solutions like SQL.
  - *Cryptography support:* These libraries help developers implementing cryptographic solutions for data storage or secure communications.
  - *Cloud integration and support:* The SDKs in this group allow for the integration of cloud services capabilities into applications, for instance Amazon Web Services [13] or Google's Firebase [127].
  - *Browser support:* These SDKs provide functionalities to open web content, such as Android's WebView which allows applications to render webpages.
  - *Cross-platform development:* While application code in Android is developed using one of the two languages supported by the platform (Kotlin and Java), there are several SDKs that allow to include code in other languages for cross-platform development. One example is Facebook Hermes [93], which allows to include React code in Android and iOS apps, or Apache Cordova [34], which allows using web development techniques to build mobile apps.
- **Push Notifications/Consumer Engagement:** Push notifications are small server-to-client messages used to reach mobile audiences anywhere and anytime. This technology is at the core of companies offering "customer engagement" services to create a direct communication channel between an external stakeholder (consumer) and an organization (often a company,

developer, advertiser, or brand). Many of the companies offering these services also offer analytics and advertisement. This is the case of Google, which offers its own crossplatform service – Firebase Cloud Messaging (FCM) [110] – JPush [169] or airPush [6].

- **Online Payments:** Several SDK providers like AliPay [10] and Google Pay [134] allow developers to include online payment services. Many mobile applications, especially mobile games, no longer implement advertising-based monetisation models. Solutions like Fortumo [113] allow developers to explore alternative sources of revenues by requesting users to pay a small fee for unlocking premium features or purchasing virtual goods.
- **Maps and Location Services:** SDKs like the Google Maps SDK [139], Here.com [152] or Baidu Maps[51] allow application developers to add maps, geo-location, and navigation capabilities to their products. The set of features and services offered by maps and location providers is very broad. While some offer pure mapping services, others like Google Maps provide data-driven added value like location-based business searches, geo-coding and even Augmented Reality (AR) services [138].
- **Authentication:** These are services that allow application developers to protect parts of the application's functionality from unauthorized access using an online identity or two-factor authentication mechanisms. Examples of these SDKs are OAuth and Google's Firebase (Google Login).
- **Social Networks:** These SDKs allow developers to include functionality from social networks, such as login capabilities and the ability to share content with a list of friends. One remarkable example is the Facebook Graph SDK, which also provides analytics and advertisement services. Applications integrating these libraries might be able to harvest personal data from the social network profile of the user.
- **Analytics:** Many companies provide analytics tools to understand how users interact with their app, find, and solve bugs and crashes, optimize user engagement, and generate revenue with highly detailed data about customers. Therefore, analytics SDKs could be broken down into several subcategories, with some SDKs providing more than one functionality to the app, including bug reporting (e.g., Crashlytics [111]), A/B testing (e.g., Firebase A/B testing [109]), and user engagement or CRM (e.g., StartApp [260]).
- **Advertisement:** Advertisement SDKs are used by app developers to show ads to users, generating revenue for the developer. Because of targeted advertisement, many of these libraries also collect personal data in order to generate user profiles to better understand the type of content that a given user is interested on. Examples of these libraries are Google's AdMob [133], Unity3D [273] or Twitter's MoPub [200].

As we have seen, many SDKs offer multiple capabilities to app developers in a single library. This impedes attributing a single label in most cases. Table 2 provides examples for each of these types, showing how the same SDK can be labeled differently depending on its behavior.

Google's Firebase SDK is a remarkable example, as it unifies analytics services, bug reporting, two-factor-authentication, services for integrating apps with Google cloud, and more [112]. While we acknowledge that this taxonomy might not be complete, we believe that it offers a representative overview of the most common solutions that can be found in today's mobile applications.

Type	Examples
Development support	Android Support, GSON, Unity3D
Network support	OKHTTP, Facebook Fizz, jmDNS
Database support	ORMLite, Android Wire, Firebase
Crypto support	Jasypt, Bouncy Castle
Browser support	HTML TextView, Chromium
Cloud integration and support	Google Firebase (Google Cloud)
Cross Platform Development	Apache Cordova, Facebook Hermes
Push notifications/Consumer engagement	Firebase Cloud Messaging, JPush, airPush
Online payments	AliPay, Fortumo
Social Networks	Facebook, Twitter, VK
Authentication	Google Firebase (2FA)
Maps/Location services	Google Maps, MapsForge, Baidu Maps
Analytics	Firebase, Baidu, Flurry
Advertisement	Unity, Google Ads, Amazon Mobile Ads

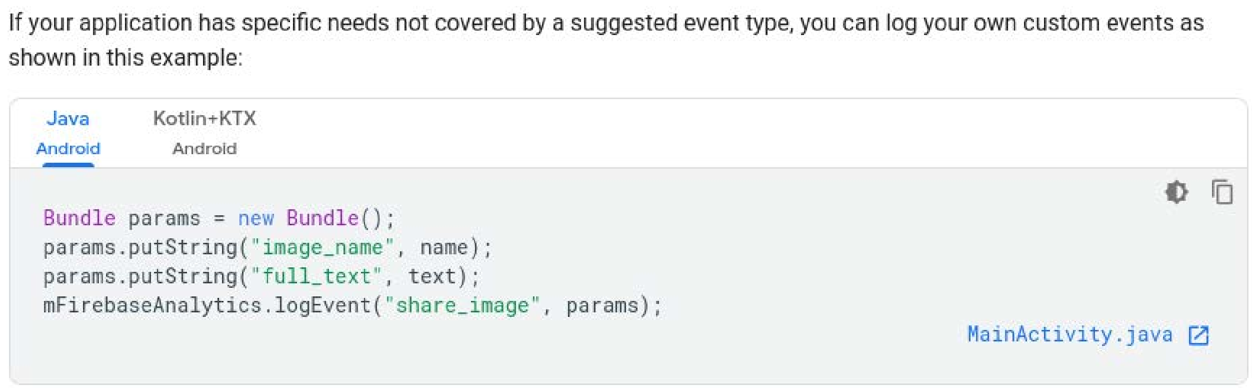
Figure 2: Examples of SDKs for each category

2.3.2 Understanding SDK's Data Collection Practices

As we have just discussed SDKs do not only have the ability to access sensitive data and resources for secondary purposes, but they are incentivized by the business model of online advertisement. Specifically those SDKs that need access to such data in order to function correctly.

Social Networks. These SDKs represent a potential threat to privacy as they give social networks the ability to monitor users' activities outside of their own mobile apps. This means that social networks can potentially leak user data to the app developer or other third parties present in the app. For example, Facebook, through its own permission model, grants access to data such as the list of friends of the user or the pages that the user has liked on the platform [94]. Likewise, Twitter4J [272] allows developers to interact directly with the user's profile on the platform and Spotify allows gaining access to user data like gender, email account, or age. Cases such as the Cambridge Analytica scandal [266], in which a political consulting firm got access to data from millions of Facebook users through a third-party app, highlight how dangerous social media data can become if it falls in the wrong hands.

Analytics. Analytics SDKs serve different purposes and, as a result, their privacy risks can vary greatly depending on how app developers integrate them into their solutions. Some analytics libraries are used for user engagement; thus, they collect behavioral data that could be linked to a given user profile. Another example are A/B testing libraries, which rely on showing two different versions of an app component to different users and measuring which of the versions receives more positive interactions, which could reveal cognitive disabilities of the user [83]. All of these uses of analytics tools are legitimate and both apps and users might benefit from them, but the collection of such behavioral data linked to sensitive data (e.g., particularly unique identifiers) should be informed to and consented by the user. Some analytics SDKs allow developers to collect events defined by the application developer, known as "custom events". For instance, the developer of a medical app might want to monitor in their analytics dashboard the number of users showing certain symptoms in a



If your application has specific needs not covered by a suggested event type, you can log your own custom events as shown in this example:

```
Bundle params = new Bundle();
params.putString("image_name", name);
params.putString("full_text", text);
mFirebaseAnalytics.logEvent("share_image", params);
```

Figure 3: Example of a custom event declaration with Firebase

geographical area. One library that allows for this kind of behavior is Firebase, in which developers can register any event that they want to track even if it's not part of the events reported by the SDK by default (see Figure 3 for a screenshot of this feature). This level of detail gives SDKs the ability to track users' every move and constitute a danger for their privacy, especially when analytics services collect information that can identify users uniquely.

Advertisement. Advertisement libraries collect personal data in order to show highly targeted advertisements to users and maximize revenue [223]. This brings severe privacy implications to users, with a high number of mobile SDKs collecting user data to create profiles and with the appearance of companies like data brokers [267], which specialize in selling these types of profiles. Furthermore, because the advertisement model is highly distributed and dynamic, multiple ad publishers bid for the ability to show an ad to a given user depending on the personal characteristics of such a user [299]. This might result in user data being broadcasted to multiple organizations without the user knowing or consenting to as pointed out by ICO. As in the case of mobile analytics, targeted advertisement is not necessarily against the user's privacy if the user has consented to such data collection and if there are appropriate mechanisms in place so that the user can exercise the associated data rights.

2.3.3 Privilege Escalation

One of the main issues of third-party SDKs on Android is inherited from Android's permission system (§ 2.2). This app-centric permission model presents fundamental limitations to properly inform users about the access to sensitive data by embedded SDKs, since they run with the same privileges and permissions as the host app, as shown in Figure 4. Many mobile users might trust the app developer when they give the app access to a piece of sensitive data such as location. However, they might not necessarily trust opaque third-party SDKs embedded in the product, particularly when the user is not even aware of their presence or is not familiar with the company, its business, and the way they will process their data.

Unfortunately, mobile operating systems and platforms fail to inform users about the SDKs that might be embedded in an app and whether they access sensitive data. Android apps do not include include a declaration of the purpose of a given permission when is requested to the user (other than a default message from the OS). On the other hand, the only way in which users can understand the third parties operating on an Android app is through the privacy policy, provided that the developer

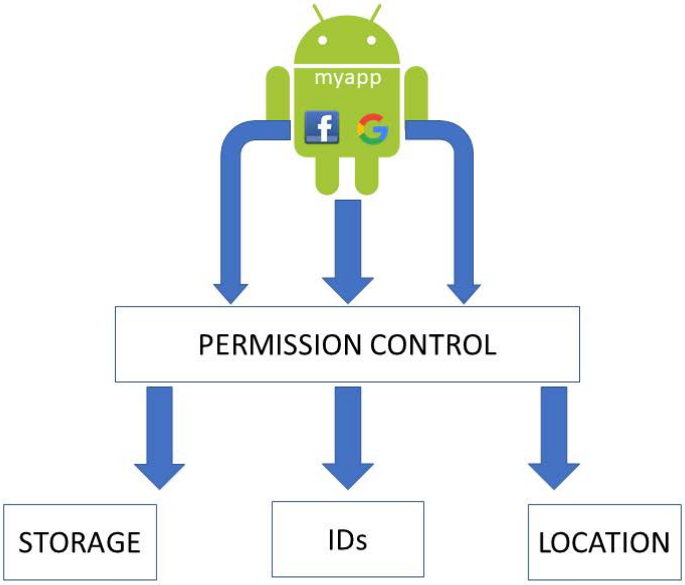


Figure 4: Permission escalation in Android: SDKs can leverage the same permissions as the host application to access protected resources. In this example, the app has access to unique identifiers, location information and the external storage. Both embedded SDKs (Google and Facebook) could access those resources without requesting the appropriate permission to do so

has disclosed this list and that the policy is complete (which is not often the case [212]). According to Google, the inclusion of privacy policies in mobile apps is not mandatory except when the application collects sensitive data or is aimed at children [142]. However, Google does not detail what review process it is followed to actively look for apps violating such a policy.

2.4 Analysis Methods

There is a vast literature on the analysis of Android apps for the analysis of their impact on the privacy and security of users. The research community has primarily relied on two methods. In short, static analysis studies *software as data* by reading it; dynamic analysis studies *software as code* by running it. Both approaches have the goal of understanding the software's ultimate behavior, but they offer insights with different certainty and granularity; static analysis reports instances of hypothetical behavior; dynamic analysis gives reports of observed behavior.

2.4.1 Static Analysis

These techniques do not rely on running the software on a system but rather on analyzing the code itself, which generally makes these techniques easier to scale. On the other side, static analysis does not produce actual observations of privacy violations as it can only suggest that a violation may happen if a given part of the code gets executed at runtime. This means that static analysis provides an upper bound on hypothetical behaviors (i.e., yielding false positives). The biggest advantage of static analysis is that it is easy to perform automatically and at scale. Developers, however, have options to evade detection by static analysis because a program's runtime behavior can differ enormously from its superficial appearance. For example, they can use code obfuscation [76, 98, 190] or alter the flow of the program to hide the way that the software operates in reality [76, 98, 190]. Native code in unmanaged languages allow pointer arithmetic that can skip over parts of functions that guarantee pre-conditions. Java's reflection feature allows the execution of dynamically created instructions and dynamically loaded code that similarly evades static analysis. Recent studies have shown that around 30% of apps render code dynamically [181], so static analysis may be insufficient in those cases.

There is a vast arsenal of static analysis tools that have been used for the analysis of privacy and security issues on Android. FlowDroid [45] is an Android-specific flow analysis tools that allows to highlight potential information leakage in Android apps. Namely, FlowDroid models taint analysis, following data originated at a given source (e.g., location or contacts manager) until it reaches a given sink (e.g., a network interface or the filesystem). Other examples of flow analysis tools are DroidSafe, a object sensitive and flow insensitive data-flow analysis [145]; and Amandroid, which is a flow analysis tool that implements a context-aware and inter-component data flow analysis [289]. However, these methods often miss flows because of their inability to model big apps, analyze reflective calls in the code, and other general flaws in their implementation [304].

In terms of permission analysis, Stowaway is a tool that relied on a permissions to Android API mapping to be able to detect over-privilege by apps [106]. The authors showed that about one third of Android apps requested permissions that were in reality never used. Sellwood et al. relied on static analysis of the Android permission model to showcase a potential issue called dormant permissions. These are permissions that are undefined at install time, and are later defined on an update without the user noticing [250]. Similarly, Wu et al. relied on static code analysis to discover a privilege escalation in Android permissions called confused deputy in over 1k apps. This attack is enabled when several parties expose a resource with the same name, and can be exploited to access sensitive resources without the appropriate permissions. Calciati et al. also relied on static analysis techniques to show that apps were taking advantage of permission groups to be able to silently request new permissions of the same group via app updates [71]. They showed that 17% of the apps that they analyze silently request at least one new dangerous permissions through an app update.

Another highly studied line of work are tools for detecting SDK presence, with myriad tools based on static analysis. Examples of SDK detection tools that rely on static analysis are LibRadar, LibScout and LibPecker [188, 48, 305]. The problem with these approaches is that it is extremely difficult to identify SDKs in applications using code obfuscation techniques. To overcome this issue, many of these tools rely on fingerprinting the SDKs to be able to detect them in obfuscated APKs.Nevertheless, SDKs are in constant evolution [72, 236] and, therefore, these fingerprints must be updated and maintained, or else the tool will become obsolete. Another detection tool that uses static analysis is Exodus [91], which relies on matching code packages names and URLs found with the list of packages names and domains related to a given SDK provider (i.e., the package `com.crashlytics` and the domain `crashlytics.com` would be matched to the Crashlytics SDK). However Exodus does not have any specific anti-obfuscation mechanisms, making it vulnerable to basic obfuscation techniques such as class renaming.

2.4.2 Dynamic Analysis

Dynamic analysis studies an executable by auditing its behavior during runtime. To that end, dynamic analysis techniques run the executable in a controlled environment, such as an instrumented mobile OS [88, 293], to gain observations of an app's behavior. There are several methods that can be used in dynamic analysis, one example is taint analysis [88, 124] which can be inefficient and prone to control flow attacks [244, 248]. One of the most prominent examples of a taint analysis tool is FlowDroid [88]. Much like the examples of flow analysis in § 2.4.1, TaintDroid follows data that originated in potentially sensitive sources through the course of the app execution, and reports those values that reach a sink. The main drawbacks of this solution are its overhead (32%) and the fact that TaintDroid needs to be updated to work with every new version of Android, making it very hard to keep up with changes on the ecosystem.

Nowadays it is common for apps to employ anti-testing and anti-emulation techniques to detect if they are being run in virtual or privileged environments to avoid scrutiny [47, 244]. This has led to new auditing techniques that involve app execution on real phones, such as by forwarding traffic through a VPN in order to inspect network communications [231, 256, 227]. One example of such a tool is AntMonitor [256], which was tested for several months in the phone's of 9 users showing small overhead and the ability to detect personal data leakage. One of the main issues with these techniques is the difficulty to to actually run apps with user input. In the case of Android applications, there are several tools for simulating user's interaction such as Apium [39], Culebra Tester [79], or the Android exerciser Monkey [23]. More concretely, the Monkey is a UI fuzzer that generates synthetic user input, ensuring that some interaction occurs with the app being automatically tested. The Monkey has no context for its actions with the UI, however, so some important code paths may not be executed due to the random nature of its interactions with the app. As a result, this gives a lower bound for possible app behaviors, but unlike static analysis, it does not yield false positives.

One way to solve this issue is via crowd-sourcing, hence collecting data from real user's devices. One example of this strategy is the Lumen monitor (formerly Haystack [231]) which relied on installing an user certificate on the phone in order to intercept traffic through Android's VPN interface and be able to inspect encrypted flows. The authors crowd-sourced this solution in order to study several aspects of the Android ecosystem, including the third-party tracking ecosystem [230], or the use of TLS across Android apps [232]. However, changes in Android and the growth in popularity of techniques robust to man-in-the-middle attacks [95, 123, 232] have rendered most of these solutions obsolete.

A remarkable example of dynamic analysis still being used for research is the Appcensus platform. This commercial solution relies on running apps in a highly instrumented version of Android and monitors access to personal data, permission usage and network traffic in order to understand what kind of personal data an app collects and who is responsible for such data collection [149, 238, 233]. However, this platform suffers from many of the same issues already discussed, mainly the difficulty to scale up the testing and the complexity to generate test inputs automatically that cover all possible code paths (specially for apps behind paywalls or login walls).

2.4.3 Hybrid Analysis

In this thesis, we argue that static and dynamic analysis methods complement each other and are best used in combination. In fact, some types of analysis benefit from a hybrid approach, in which combining both methods can increase the coverage, scalability, or visibility of the analyses. This is the case for malicious or deceptive apps that actively try to defeat one individual method (e.g., by using obfuscation or techniques to detect vitalized environments or TLS interception [292, 33, 285, 31]). However, to the best of our knowledge, most research efforts have relied solely on one or the other. A notable example on the literature is the work by Pan et al., which relies on a combination of static and dynamic analysis to identify Android apps sharing audio and video data through the network [215]. To that end, the authors first employ static analysis methods to find apps that have access to media-related permissions in order to analyze their runtime behavior in search for evidence of the sharing audio or video data. In this thesis, we present an analysis of parental control apps [99] as well as a security analysis of the use of covert and side channels to circumvent the Android permission system [233] as motivating examples on the effectiveness of combining both analysis methods.

Chapter 3 Regulatory Compliance Analysis: Mobile Parental Control Apps

In this chapter, we set out to test our initial hypothesis ($\mathcal{H}$ 1) by applying a novel methodology that combines static and dynamic analysis methods to better understand whether parental control apps are in potential violation of special provisions for minors in relevant legislation (i.e., GDPR and COPPA). We use static analysis as a proxy to detect apps with potentially dangerous behavior for the privacy of users, by analyzing their permission and SDK usage. Then, informed by a flow analysis of potential data sharing, we dynamically test all apps in the dataset to obtain actual evidence of potential privacy and regulatory misbehaviors. By combining static and dynamic analysis in this novel way, we are able to produce a full regulatory compliance analysis of a previously unexplored ecosystem of highly invasive Android apps. We discovered that parental control apps often collect personal data from minors, oftentimes without parental consent, and share it with first and third party domains through the network, sometimes without appropriate security measures.

In this chapter, we first introduce the way in which parental control apps operate (§ 3.1) and our novel methodology to study the privacy risks of these apps and how they comply with specific regulations (§ 3.2). Relying in this methodology, we first present an analysis of the permissions requested by the apps in our dataset (§ 3.3) and the third-party SDK that are embedded in them (§ 3.4). Then, we show actual evidence of privacy misbehavior via dynamic analysis (§ 3.5) and put our findings in context with the privacy policies presented by the apps (§ 3.6). We finish this chapter with a number of recommendations to reduce the risks of relying on these solutions (§ 3.7).

3.1 Parental Control Apps

Android parental control apps offer diverse features to control children's digital activities. In line with previous work [296, 185], we classify apps that enable parents to monitor children's behavior, including location [144], as *monitoring tools*; and we classify apps that enable parents to filter content and to define usage rules to limit the children's actions as *restriction apps*. Some apps offer multiple functionalities simultaneously, and we label them as *monitoring* since it is the most invasive of the two categories.

The way in which parental control apps enforce these functionalities is varied. Common alternatives are replacing Android's Launcher with a custom overlay in which only whitelisted apps are available; installing a web browser where developers have full control to monitor and restrict web traffic; or leveraging Android's VPN permission [30] to gain full visibility and control over network traffic. The most common criteria to customize app's behavior are age and block lists / allow lists. The former restricts the type of content available to the children or defines the level of monitoring depending on the children's age. In general, older children are granted access to a larger set of webpages and applications than youngsters. In the latter, parents can either block applications or webpages that they deem inappropriate for their children, or add appropriate content that is allowed.

Most parental control apps have two different components or modes: *i*) the parent app or mode, and *ii*) the children app or mode. The parent app can run either on the children's or on the parent's device enabling access to the children's app data, and to the dashboard for setting monitoring or blocking rules. When the parent mode runs on the children's device, parents' monitoring can be done locally on this device. However, when the parent and children app run on different devices, the children app often uploads information to a central server in order to enable remote access to the children's information to parents. Many apps in our study provide a web-based control panel in which parents can access all information, and change both control and blocking rules. This approach *requires* uploading data to the cloud, and as a results, parents must trust service providers to not disseminate nor process children's data for secondary purposes other than the ones declared in the privacy policy.

To assist developers of children's apps, Google has made public best development practices [18], as well as a list of self-certified third-party advertising and tracking libraries that respect children's privacy [128]. While parental control apps are not necessarily listed in the Designed for Families (DFF) program [126], a Google Play program for publishing applications targeting (and suitable for) children, given their nature we expect them to follow regulation's special provisions and follow best practices for collecting data about minors.

3.1.1 Regulatory Framework

As discussed in 2.1, there are several regulations relevant to children privacy. Minors may be less concerned regarding the risks and consequences of the dissemination of their personal data to the Internet [77, 184]. This has motivated regulators to develop strict privacy laws to protect children privacy such as the Children Online Privacy Protection Act (COPPA) in the US. COPPA dictates that personal data of children under the age of 13 can only be gathered by online services, games, and mobile applications after obtaining explicit and verifiable parental consent [101]. In the EU, the General Data Protection Regulation directive (GDPR) enforces privacy transparency and requires data collectors to obtain consent from European subjects prior to gathering personal data from them, clearly stating the purpose for which it is collected [77]. As in the case of the USA COPPA rule, the GDPR contains additional articles in relation to the privacy of minors —namely Art. 8 [154] and Recital 38 [155] of GDPR—which require organizations to obtain verifiable consent from the parent or legal guardian of a minor under the age of 16 before collecting and processing their personal data. Both regulations explicitly prohibit the collection and processing of minor's data for the purpose of marketing or user profiling, and force developers to implement security best practices (e.g., use of encryption) and to disclose the presence and data collection practices of embedded third-party such as advertising and tracking services [77, 101].

3.2 Data Collection and Analysis Methodology

Google Play does not provide any specific category for parental control apps. We identify them by searching for the string *"Parental control app"* on Google Play's search engine. As this process can produce false positives, we manually eliminate any app that does not implement parental control functionalities. We repeated this process at different points during the project to add newly published apps to our study. In total, we found 61 parental control apps. Interestingly, the majority of these apps are classified as *Tools* (42% of apps) by developers, despite the presence of the more specific *Parenting* category (15% of apps).

Android app developers often release new versions to include new features to their software or to patch vulnerabilities [72]. Such changes may have an impact on users' privacy [236]. We crawl APKPure [35] —a public dataset which contains historical versions of Android applications—to obtain 429 old versions for the list of 61 apps so that we can study their evolution over time. We discard versions prior to 2016, as we deem them too old for our study. This also results in discarding 15 apps that have not been updated since 2016. Anecdotally, one of the developers only made the parent version available through Google Play, while its counterpart was only available through direct download at the developer's website. We decided to download the children version to avoid bias in our dynamic analysis. Our final dataset contains 419 versions from 46 parental control apps. Table 3.1 provides a classification of each app attending to whether it is a monitoring or restriction app, if it is currently available on Google Play and if it is benchmarked by the SIP and IS4K alternatives.

For each app we harvest metadata publicly available on the Google Play store: app descriptions, number of downloads, user ratings, privacy policies, and developer details. We use this metadata in our analysis to contextualize our results, showing potential differences across developer type and app popularity, and for analyzing the completeness of privacy policies. [1]

Apps' popularity. The number of installs per app—a metric often used to infer a given app's popularity [287, 306]—varies widely. The dataset contains 22% of apps with over 1M downloads, whereas 37% of apps have less than 100k downloads. When considered together, the 46 apps have been installed by more than 22M users (lower bound). The most downloaded apps typically have a better user rating (on average 3.5 out of 5 for apps over 1M downloads) than those with a lower number of installs (on average 3.2 out of 5 for apps below 100k installs).

App developers. We extract the apps' signing certificates to identify the set of companies (or individuals) behind parental control apps. We find that most developers release one single parental control app, except for 3 developers that have released two apps. 21% of the developers also publish unrelated software in Google Play besides the identified parental control apps. The most remarkable cases are *Yoguesh Dama* [143], an Indian company which has published 38 apps (for volume control, screen locking, wallpaper, and more), and *Kid Control Dev* [141] a Russian company which also builds apps such as flashlights.

One may assume that the use of parental control apps is mostly predicated on trust. Therefore, we investigate whether the identity of the app developer plays a role in the parents' choice. Our initial hypothesis is that parents might prefer software developed by well-known security companies like *McAfee* or *Norton*. However, we find that these developers only account for 9% of total apps, and that they are not the most installed ones – only one of them reaches 1M installs. The most popular parental control apps are those developed by companies specializing in children-related software (e.g., *Kiddoware* [171] and *Screen Time Labs* [249]). Most parental control apps seem to monetize their software through in-app purchases (from 1 EUR to 350 EUR) to unlock features and monthly licenses, yet they typically offer a free-trial period.

[1] The set of 46 apps may not contain every available parental control app, but the apps are diverse from different standpoints and can be considered as a representative the dataset.

Table 3.1: Summary of our corpus of apps attending to different features. The column “benchmarked” indicates whether the app has been analyzed by [89, 157]. The values “M” and “R” stand for Monitoring and Restriction, respectively

Name	Type	Listed as of 2019/05	Benchmarked
com.mmguardian.childapp	M	✓	
com.infoweise.parentalcontrol.secureteen.child	M	✓	
com.qustodio.qustodioapp	M	✓	✓
com.kaspersky.safekids	M	✓	
com.kiddoware.kidsafebrowser	R	✓	
com.securekids.launcher_reloaded	M	✓	✓
com.eset.parental	M	✓	✓
com.symantec.familysafety	M	✓	
con.netspark.mobile	M	✓	
com.nationaledtech.Boomerang	M	✓	
com.mobilefence.family	M	✓	
com.infoweise.parentalcontrol.shieldmyteen	M		
com.teenlimit.android.child	M	✓	
com.bhanu.childlocklauncher	R	✓	
com.safekiddo.kid	M	✓	
com.kidoz	R	✓	
com.cresoltech.trackidz	M		
net.safelagoon.lagoon2	M	✓	
com.screentime	M	✓	✓
com.kiddoware.kidsplace	R	✓	
com.mcafee.security.safefamily	M	✓	
io.familytime.parentalcontrol	M	✓	✓
com.vionika.parentalBoardAgent	M		
de.protectyourkid	M	✓	
com.parentsware.ourpact.child	R	✓	
ru.kidcontrol.gpstracker	M	✓	
com.kidslox.app	R	✓	
com.zerodesktop.appdetox.dinnertimeplus	R	✓	
com.pdlp.android.app	R	✓	✓
com.redgreentree.app	R		
com.sentry.kid	M	✓	
com.whisperarts.kidsshell	R	✓	
funamo.funamo	M	✓	
com.bitdefender.parental2013	M		
com.gizmoquip.gametime	R		
com.kakatu.launcher	M		
com.calmean.parental.control	M	✓	
mobicip.com.safeBrowserff	M	✓	✓
com.saferkid.parentapp	M	✓	
com.fsp.android.c	M	✓	
com.locategy.family	M	✓	
com.parentycontrol	R		
com.leapteen.parent	M		
com.kidgy.android	M		
com.mspy.lite	M	✓	
com.appobit.familyorbit	M		

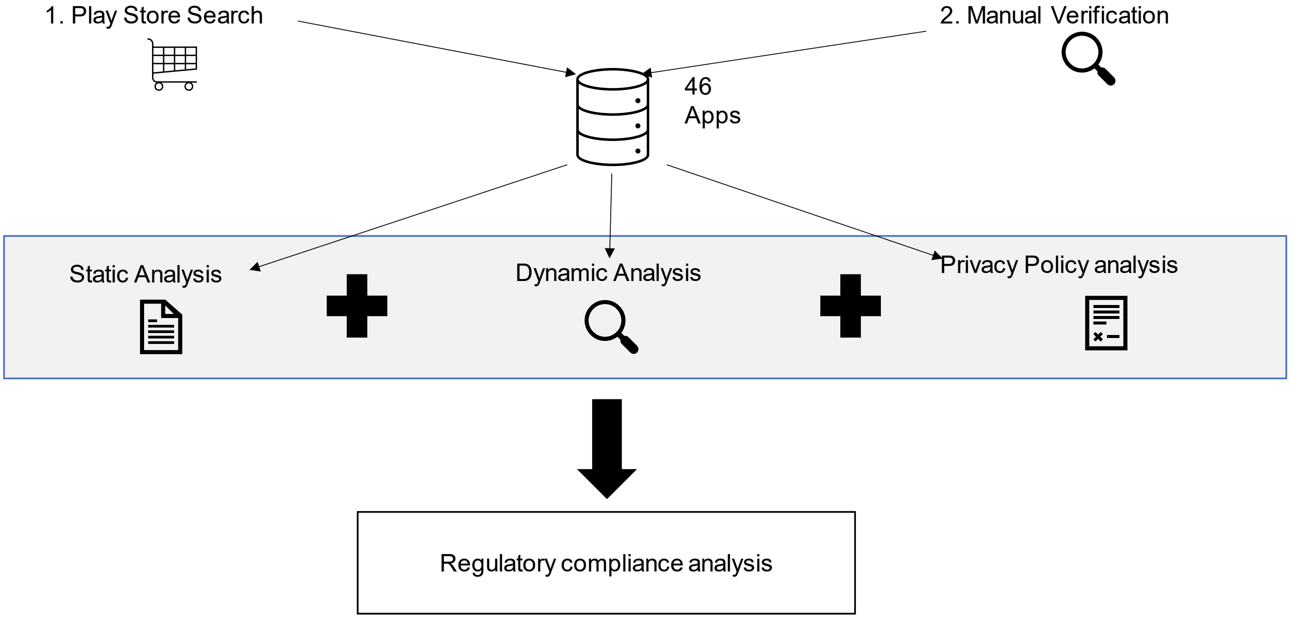


Figure 5: Analysis pipeline to assess the privacy risks and regulatory compliance of parental control apps.

Delisted apps. We note that 10 applications were removed from Google's app store since our initial app collection (Feb. 2018). The reason why these apps have been delisted is unclear: their removal could be triggered by the developers (e.g., companies no longer operating), or by Google Play's store sanitization process (Play Protect [20]). We still analyze these apps, and report when they cannot be tested because they no longer work.

3.2.1 Analysis Pipeline

To carry out our analysis we take advantage of static and dynamic analysis techniques previously developed by the research community—in some cases extending them to meet our research objectives—as shown in Figure 5. As we discussed in § 2.4, static and dynamic analysis methods present limitations when used in isolation. Therefore, we use them in combination to overcome this limitations and present a more accurate and complete analysis of the privacy risks of parental control apps and the SDKs that they embed. We discuss the limitations of our research in § 3.7.

Static Analysis. For each app and version, we first parse its Android Manifest file [25] to understand its high level behavior without analyzing the binary code. Concretely, we search for: *i*) apps with unusual permission requests, and *ii*) apps that request a different number of permissions across versions. We complement this analysis with static taint analysis to investigate potential personal data dissemination by apps and embedded third-party SDKs. Finally, we look at the binary code to identify third-party SDKs embedded in the app using LibRadar [188]. This last step is critical to attribute observed behaviors to the relevant stakeholder. We present the results of our static analysis in § 3.3 and § 3.4. As discussed throughout this thesis, static analysis has no visibility into server-side logic, and cannot analyze apps with obfuscated source-code. Further, it may report false positives so to improve our ability to understand the privacy risks posed by parental control apps and their embedded SDKs, we complement it with dynamic analysis, as discussed next.

Dynamic Analysis. The fact that apps declare a given permission or embed a given third-party SDK does not mean that the permission will be used or that the library will be executed. We use dynamic analysis to collect actual evidence of personal data dissemination (§ 3.5.2), assess the security practices of these apps when regarding network communication (§ 3.5.4), and identify consent forms rendered to the user at runtime (§ 3.5.3). We use the Lumen Privacy Monitor app [231], an advanced traffic analysis tool for Android that leverages Android's VPN permission to capture and analyze network traffic—including encrypted flows—locally on the device and in user-space. The use of Lumen only provides a lower bound to all network communications in an app since it can only capture flows triggered by user-, system- or environmental stimuli and codepaths at runtime. Achieving full coverage of parental control apps' actions is complicated and time consuming: we must manually set up a parent account and then exhaustively test the app by mimicking phone usage by a child. The process of testing the applications needs to be manual as the Android exerciser Monkey [23] is not able to either fill login forms or test app features in a non-random way [75]. We provide details on the testing methodology in § 3.5.2.

Hybrid Analysis. We have just described the different limitations of static and dynamic analysis when used on isolation. However, in this chapter, we use them in combination to overcome such limitations. We first understand the potential privacy issues of parental control apps using static analysis techniques. Then, we rely on those findings to inform the way in which we exercise the apps for dynamically testing their runtime behavior, allowing us to get evidence of potential violations of GDPR and COPPA when analyzing their runtime behavior.

Privacy Policy Analysis. We conduct a manual analysis of the latest version of the apps' privacy policies (§ 3.6) fetched from their Google Play profile. We inspect them to identify: *i*) whether the apps provide understandable and clear terms of use and privacy policies to end users; *ii*) whether they are compliant with the provisions of privacy regulations; and *iii*) whether their disclosed policies match their behavior in terms of access to and dissemination of sensitive data and the presence of third-party SDK.

3.2.2 Ethical Considerations.

Previous work has shown the use of parental control applications may have ethical implications [73, 192], therefore **we do not collect any real data from children or any other user**. We perform our data collection on fake accounts owned and operated by ourselves. Furthermore all interaction with the apps is done by the authors, without any children or end user participation. We put our focus on understanding the security and privacy practices of these apps to determine to what extent they could be a privacy threat to children. We also stress that some of the privacy issues found in mobile applications might be the result of bad coding habits and lack of awareness—specially when integrating privacy-abusive third-party SDKs—, rather than intentional developer misbehavior. We communicated our findings to the Spanish National Data Protection Agency (AEPD) and other government agencies promoting Internet safety, namely INCIBE's IS4K [157].

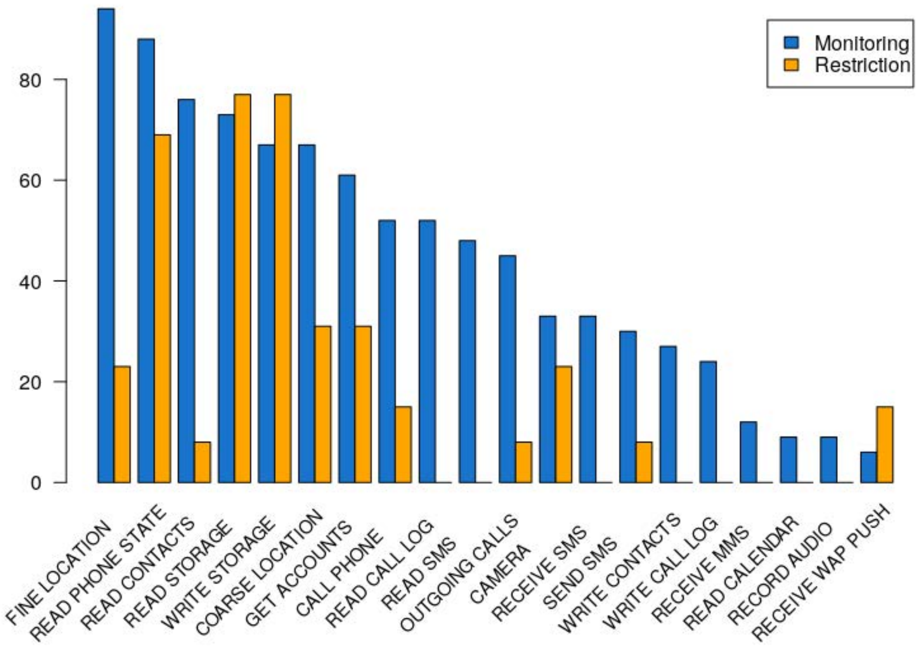


Figure 6: Comparison of percentage of apps asking for dangerous permissions between Monitoring and Restriction apps.

3.3 Permission Analysis

As we deeply explored in § 2.2, Android implements a permission model to control app's access to personal data (e.g., contacts and email) and sensitive system resources (e.g., sensors like GPS to access location) [17]. The analysis of the permissions requested by an app offers a high level idea of what protected system resources and data the app has access to. Besides Android's official permission list defined in the Android Open Source Project—grouped in different risk levels, the most sensitive ones labeled as "dangerous"—any app can define its own "custom permissions" [271, 115] to expose its resources and capabilities to other apps. We study the use of Android permissions and their evolution across releases to study what data parental control apps access, how it changes across app versions [236, 72], e.g., to adapt to stricter privacy requirements at the platform level [28], whether this data is potentially sent over the Internet, whether these permissions are used by third-party SDKs, the app itself, or by both; and whether there are noticeable differences between monitoring and restriction apps in terms of permissions request.

3.3.1 Prevalence and Usage of Permissions

Figure 7 shows the permissions requested by each one of the apps in our dataset, as well as the category of the permission. The top part of the plot shows restriction apps and the bottom part lists monitoring apps. Column-wise, we differentiate permissions according to their privacy risk level [19]:

1. Dangerous permissions: (first block from the left, in blue)—i.e., those that could potentially affect the user's privacy or the device's normal operation, and therefore should be explicitly granted by the user.
2. Signature permissions: (second block, in orange)—i.e., those that the system grants only if the requesting application is signed with the same certificate as the application that declared the

permission. We also render system permissions in the same category, even those that are deprecated in recent releases of Android, but still work on older releases.

3. Custom permissions (last block, in yellow)—i.e., those that apps define to protect their own resources.

To improve the visibility of this figure, we do not display normal permissions—i.e., those which regulate access to data or resources that pose little risk to the user's privacy, and are automatically granted by the system. However, they are part of our study.

We use color gradients to show apps' permissions request changes over time in Figure 7. Darkest colors show that all releases of the app request the permission (i.e., no changes). Figure 7 shows that, as expected due to their invasive nature, in general, parental control apps need lots of permissions to offer their service. Considering the median value, parental control apps request 27 permissions, 9 of them being labeled as dangerous. For comparison, the top 150 apps from the Google Play Store in May 2019 request 18 permissions, 5 of them labeled as dangerous. In fact, we find that the most extreme parental control app in our dataset (*Boomerang*) requests 91 permissions, of which 16 are dangerous; much more that typically considered privacy invasive non-parental control apps.[2]

We find that over 80% of parental control apps, regardless of their category, request access to location, contacts, and storage permissions. Popular non-dangerous permissions requested by parental control apps are related to Android's package manager (possibly to monitor installed apps), and also to control system settings. Despite the fact that we focus primarily on dangerous permissions, it is important to highlight that non-dangerous ones can be harmful too. One example is the `BIND_ACCESSIBILITY_SERVICE` permission, which is requested by 11 apps of the monitoring category, and 1 of the restriction apps. This permission can be used to monitor and control the UI feedback loop and take over the device [114]. During the installation process many of these apps explain that such permission is necessary to better control the actions of the child. Likewise, despite not being explicitly flagged as "dangerous", some of the deprecated system permissions are also worth noting, since they protect low level functionalities such as the ability to retrieve the list of running processes on the device. Furthermore, the permission model can be exploited to access personal data without user consent via side-channels like using WiFi information to get city level location or the MAC address information as a unique identifier [233].[3] While the type of service provided by these apps might justify the use of such dangerous permissions, the presence of third-party SDKs that can piggyback off these privileges to gather children's data is a privacy threat that we will analyze in detail in § 3.4. [4]

At the other side of the spectrum, we notice that two parental control apps do not request any dangerous permission. One of them (*Parental Control App by Redgreentree*) replaces the Android default launcher to let the child use only apps approved by the parent. The other app (*SaferKid*) is a parent app (which explains the lack of dangerous permissions) present in our dataset because the developers only made the companion app and not the children version of the app available in Google Play.

[2] For reference, *Facebook* requests 59 permissions, 16 of which are dangerous.

[3] In fact, Android 10 introduced special permissions to avoid the usage of the MAC address as a location surrogate and the access to non resettable identifiers

[4] Because of Android's permission model, third-party SDKs inherit the same set of permissions as the host application.

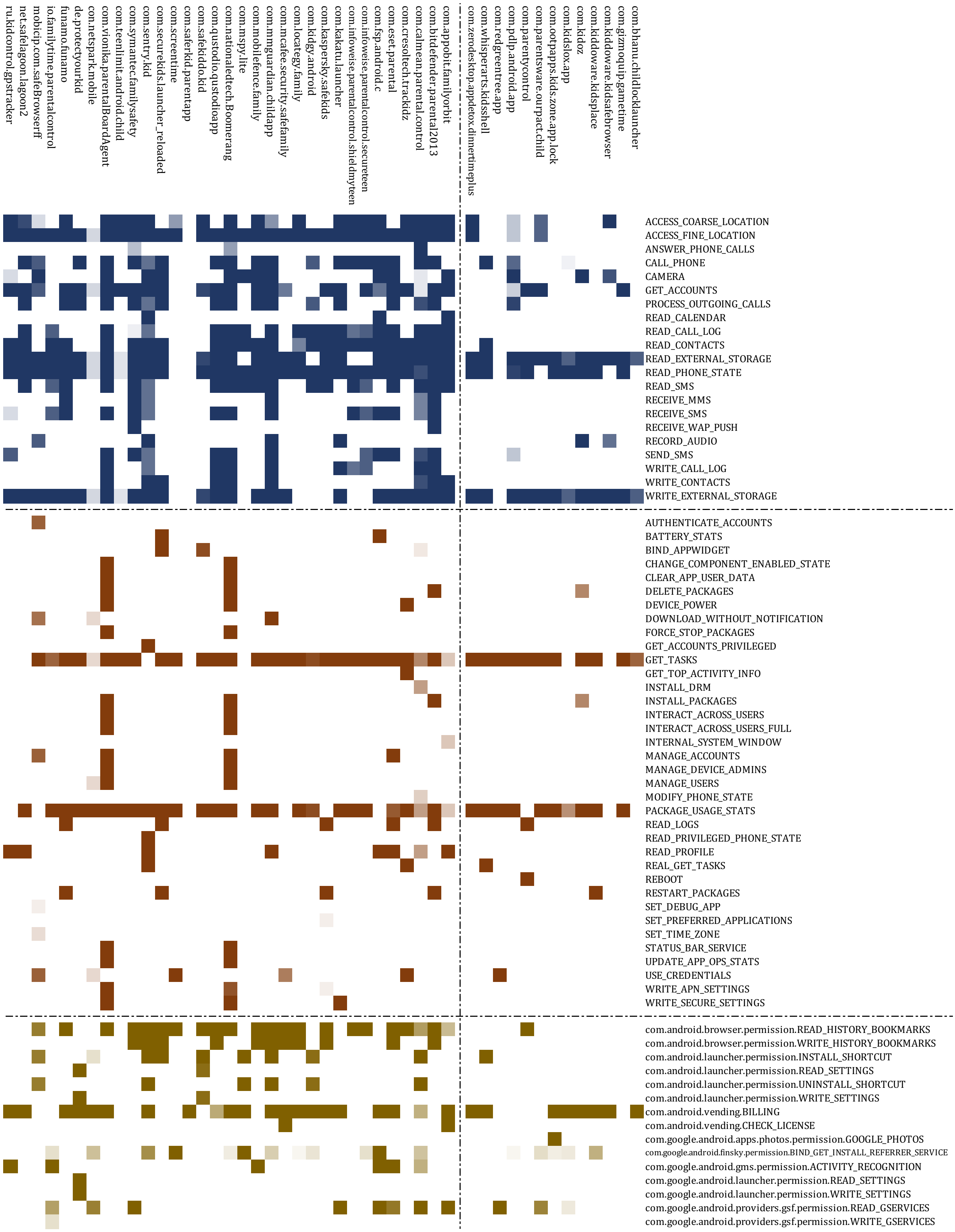


Figure 7: Permission heatmap. We list one app per row, and we indicate the permissions it requests. We show restriction apps at the top, and monitoring at the bottom. We differentiate dangerous permissions (blue), from signature (orange), and custom (yellow). Gradient shows the percentage of releases requesting the permission, with darker meaning a higher number of releases.

Table 3.2: Unusual permission requests. We report whether the apps have been unlisted from Google Play (GPlay column).

App	GPlay	Permissions
com.pdlp.android.app		PROCESS_OUTGOING_CALLSSEND_SMS
com.whisperarts.kidsshell		READ_CONTACTS
com.appobit.familyorbit	✓	READ_CALENDAR
com.mmguardian.childapp		RECORD_AUDIO
com.bitdefender.parental2013	✓	RECEIVE_WAP_PUSH
com.sentry.kid		READ_CALENDAR RECORD_AUDIO
com.symantec.familysafety		RECEIVE_WAP_PUSH
com.fsp.android.c		READ_CALENDAR
com.kakatu.launcher	✓	RECORD_AUDIO

Anomalous permission requests. While most of the apps requests dangerous permissions, we find that some are rarely used by most parental control apps. We analyze in depth the permissions that are requested by only 10%, or less, of the apps in our dataset,—considering monitoring and restriction apps separately—to identify whether these permissions are justified. We also search for any information on their privacy policy that may justify their use. Table 3.2 provides a summary of the anomalous permissions for which we found no justification in the app's description or privacy policy. Two restriction apps request permissions to send SMS, process outgoing calls and read contacts. Yet, we could only find a justification for one of them in the app's description. Regarding the monitoring category, we identify seven apps requesting anomalous permissions (i.e., receive WAP push, record audio and read calendar). Three of them are no longer indexed on the Google Play Store (*Kakatu*, *Family Orbit* and *Bitdefender*). The remaining four do not justify why they require access to these permissions in their Google Play description or privacy policy.

Permission requests across app releases. Previous studies show that Android apps tend to increase the number of requested permissions across app releases [70, 290, 265]. While 24% of the apps in our dataset increased the number of permissions requested over time, we find that another 24% of the apps decrease the number of permissions requested over time, *opposite* to the general trend. Part of this decrease is explained by the fact that in early 2019 Google changed its permission policy and disallowed apps to access SMS and calls permissions unless they were the default messaging app [130].

Restriction vs. Monitoring apps. Monitoring apps' goal is to gather behavioral and usage data from children's phone and reporting it to their parents, either via a dedicated web portal or in a parent-specific companion app. Therefore, as Figure 7 reveals, monitoring apps tend to request more dangerous permissions than restriction apps. Figure 6 goes deeper into the specific cases. Our empirical observations are aligned with our expectations: compared to restriction apps, monitoring apps commonly request permissions necessary to gather children data such as geolocation, read call log or read SMS. The difference between monitoring and restriction apps is notable even for custom permissions (see Figure 7). Monitoring apps access the browser bookmark history more than restriction apps; however, restriction apps tend to rely more heavily on custom-specific browser permissions to control the browsing activity on the children's device.

Possible information dissemination. The high request rate of dangerous permissions does not imply that parental control apps actually disseminate sensitive information over the network, either to their own servers or to third parties. We apply static taint analysis to estimate to what extent these apps access and disseminate the sensitive information they access and with whom. Using the context and flow sensitive analysis implemented in Flowdroid [45], we observe that the apps in our dataset include 1,034 method invocations to access sensitive data when aggregated, never less that 78, and on average 324. We also observe that 67% of apps also have at least 1 sink—a part of the OS where data can leave the system, such as the network interface—in reachable code, suggesting potential information leaks. Flowdroid reports at least one valid information flow in 14 of these apps, with the extreme case of *Kids Zone*, for which it highlights *72 potential leaks*. While it seems that most of the reported information leaks involve logging user actions (e.g., opening a new activity, logging a failed/successful authentication, etc.), we also observe more critical leaks involving sensitive data like unique identifiers. Specifically, we find that two apps (*SecureTeen* and *ShieldMyTeen*) share the MAC address and SSID through the Internet (they can be used as a side-channel to uniquely identify the device and geolocate the user [233]). The aforementioned apps and *Dinner Time Plus* also disseminate the device IMEI, a unique identifier; and four apps (*MMGuardian*, *Shield My Teen*, *GPS Tracker* and *Mobilefence*) disseminate GPS-level geolocation. We will further investigate in § 3.4.2 the origin and destination for these information leaks.

Custom Permissions. Android gives developers the opportunity to define custom permissions in order to expose parts of the apps' functionalities to other apps [271]. In the case of custom permissions the user never gets to accept or reject them as they are automatically granted without showing a warning or consent form to the user, as opposed to Android official permissions. Figure 8 shows the number and type of custom permissions for each app in our dataset.

We find 28 custom permissions that—judging by their name—have been declared by parental control app developers. We find examples of custom permissions from companion apps, e.g., the *Spin browser* [63], that substitutes the default user browser, or the companion parent app (e.g., *com.safekiddo.parent*). These permissions are used from the children app to access or control functionalities of the companion apps. We also find apps using custom permissions declared by other developers, such as the app *Parents Around* using custom permissions from *Protect your kid*. This suggests the existence of agreements between app developers to leverage functionality implemented by other apps when both are installed on the same device. We also find several apps using custom permissions related to phone manufacturers, possibly enabled by pre-installed applications [115]. These vendor-declared permissions allow app developers to gain access to other system features not available through the official Android Open Source Project. In some cases, as in the case of the Samsung KNOX API, app developers must become Samsung partners to gain access to their proprietary APIs [243]. Although the parental control app *Parents Around* requires access to custom permissions belonging to five manufacturers, in general apps in our dataset tend to declare permissions for either one vendor or none of them. The most frequent custom permissions are related to Samsung and they are used by various releases in 21 apps of our dataset. The most common vendor permissions are declared by Huawei (10 apps), HTC (8 apps), and Sony and Oppo (4 apps).

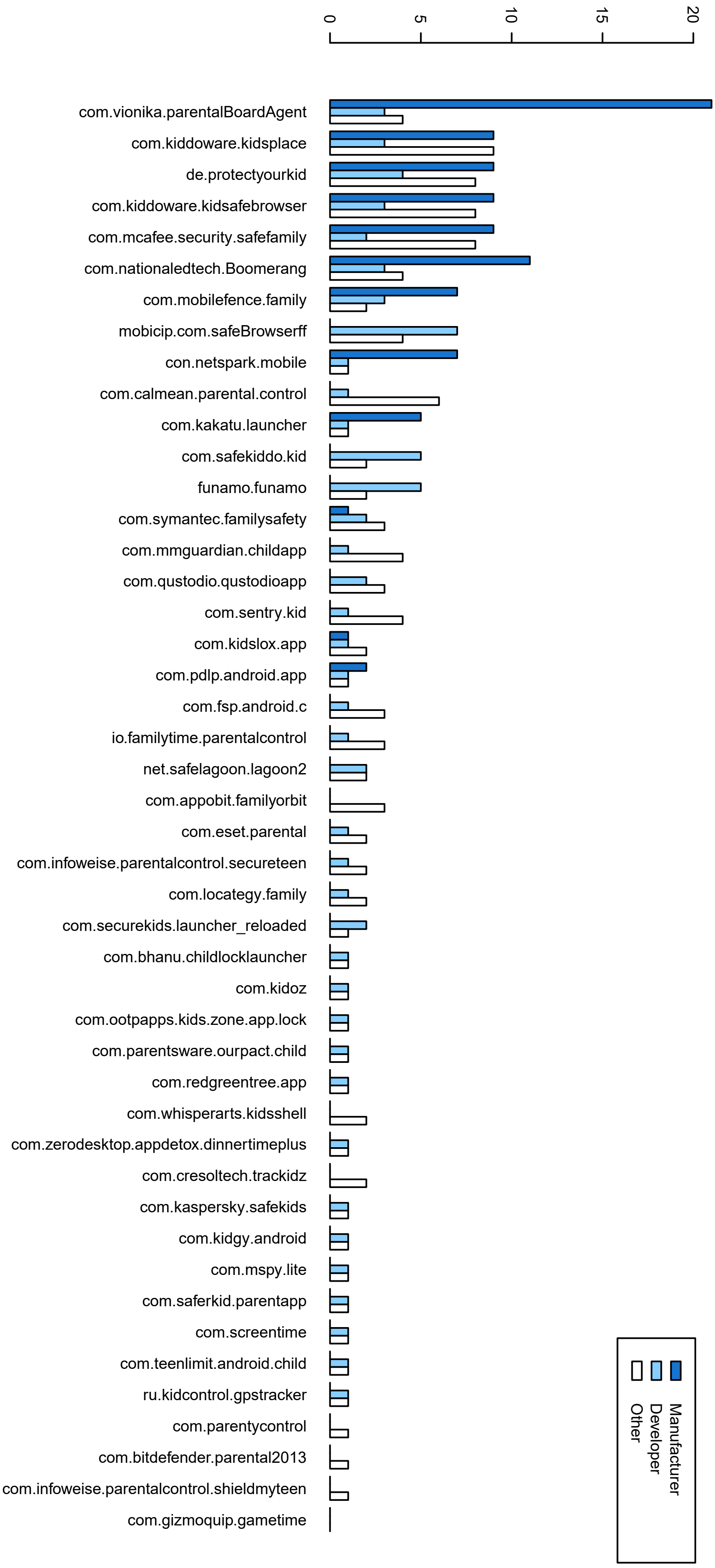


Figure 8: Number and type of custom permissions found in the apps in our dataset

Table 3.3: Classification, number and description of third-party SDKs found in parental control apps.

Category	Lib #	Description
Social Network	1 (60 pkgs)	Social networks
Advertisement	7 (82 pkgs)	Advertisement
Development	56 (582 pkgs)	Development support tools
Functionality	29 (172 pkgs)	App features (e.g., animations)
Support	43 (443 pkgs)	Support libraries (e.g., DB drivers)
Analytics	21 (220 pkgs)	Analytics services
Unrecognized	44	Unidentified libraries

3.4 Third-party SDK Analysis

As we discussed extensively in § 2, due to Android's permission model, any SDK embedded in an app inherits the privileges and permissions of the host app. This gives many organizations, including third-party providers offering analytics or advertisement services, access to the same data as the parental control app.

Many third-party SDK providers often prohibit their usage in children-oriented software, such as *Branch* [65] and *Appboy* (now rebranded as *Braze*) [67] due to the strict regulatory frameworks implemented in the USA and the EU to protect minors. Additionally, Google released in May 2019 a list of third-party SDKs that self-verify being in compliance with the provisions set by the GDPR and COPPA [128].

We inspect the parental control apps—across app versions— in our dataset to find third-party SDKs and classify them by the functionality provided using LibRadar [188]. We manually analyze its output to sanitize the results and improve LibRadar's coverage, also mapping them to the actual organization providing the service. [5] After this process, we could successfully identify 157 unique libraries (44 could not be classified due to the use of code-obfuscation techniques). Then, we use publicly available information to classify each one of the libraries by its purpose, including their own product descriptions and developer forums.

Table 3.3 shows for each resulting category the number of SDKs found, the total number of package names matching each one of these libraries, and a short description of the category. Most third-party SDKs embedded in parental control apps are tools for development support—e.g., general purpose development libraries, JSON parsers or UI support libraries—followed by SDKs offering analytics services and advertisements—e.g., Flurry and AppsFlyer. The latter category of SDKs are more concerning from a privacy standpoint given their data-driven behavior and business models. Therefore, we focus our analysis on the libraries providing social network integration, online advertisement, and analytics services.

Figure 9 shows that Google Ads, Google Firebase and Google Analytics are present in over 50% of the apps, followed by Facebook SDK at 43%. Some of these libraries belong to the same parent company. For instance, Crashlytics, Firebase, Google Analytics, and Google Ads all belong to Alphabet [11]. Therefore, when we group the SDKs by their parent company, we can observe that Alphabet is present in 93% of the apps in the dataset.

[5] For example, LibRadar can report multiple package names for the same library (e.g., `com/paypal/.../onetouch` and `com/paypal/.../data`) so we cluster those belonging to the same provider: PayPal.

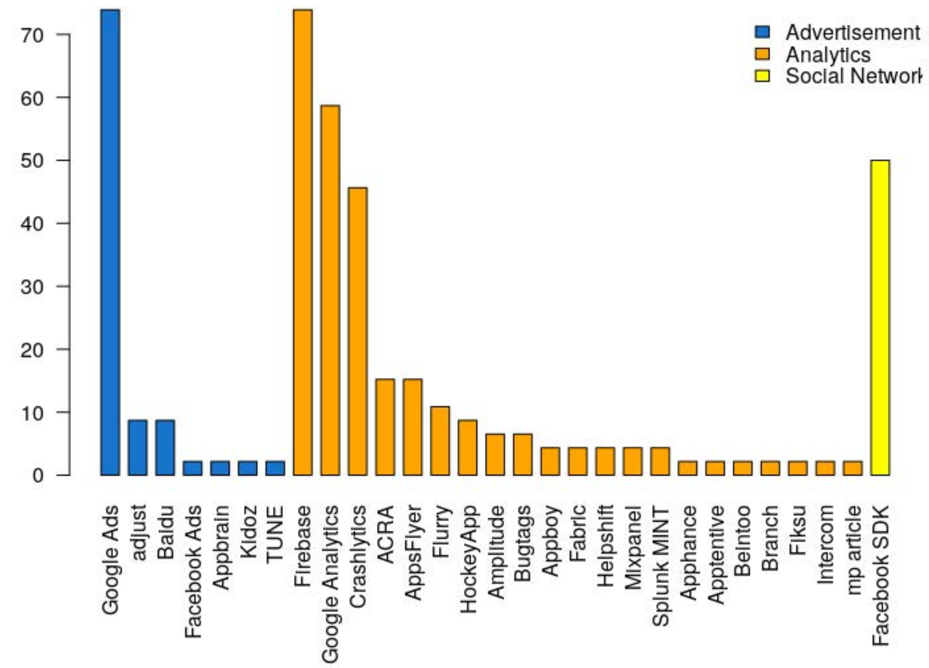


Figure 9: Percentage of apps using Advertisement, Analytics and Social Network libraries.

3.4.1 Apps Violating SDKs' ToS

Finally, we study whether the third-party SDKs embedded in each parental control apps allow their use in children-oriented software in their Terms of Service. Reyes et al. showed that several apps designed for children and published in Google Play's Designed for Family program (DFF) used third-party SDKs whose Terms of Service (ToS) prohibit their integration in children apps [238]. Following this insight, we use a static analysis pipeline based on Soot and Flowdroid [281, 45] to check whether parental control apps follow a similar behavior. We note that some SDK providers changed their ToS between our study and the previous work by Reyes et al. We also study whether these libraries are included in Google's list of self-certified SDKs suitable for children apps (as of May 2019) [128].

While both AppBoy and Branch still indicate that they should not be included in applications directly targeting children, our static analysis pipeline indicates that they are present in children-oriented software as parental control apps. We verify that two apps in our dataset (*GPS tracker* and *Qustodio*) integrate these libraries in the latest version available in our dataset (November 2018). Out of the seven advertising and analytics libraries that we find in our dataset, only two (AdMob and Unity) are present in Google's list. This is the relevant wording, asextracted from their Terms of Service:

Branch.io [65]. *You will not use our Services to: {...} (ix) create lists or segments of children under the age of 13 (and in certain jurisdictions under the age of 16), advertise mobile Apps that are directed to children under 13 (and in certain jurisdictions under 16), and/or knowingly market products or services to children under 13 (and in certain jurisdictions under the age of 16), without employing appropriate settings within the Branch SDKs to limit data collection for children under 13 (and in certain jurisdictions under 16), in order to comply with any applicable laws protecting children (including, but not limited to, GDPR and the U.S. Children's Online Privacy Protection Act ("COPPA");*

Table 3.4: Attribution of permission usage to code belonging to application or third-party SDKs for each dangerous permission. The last column reports uses that we cannot attribute.

Permission	% app only	% lib only	% app & lib	% unclear
ACCESS_COARSE_LOCATION	36	12	52	0
ACCESS_FINE_LOCATION	33	9	55	3
CAMERA	14	7	7	72
GET_ACCOUNTS	25	12	4	59
PROCESS_OUTGOING_CALLS	12	0	0	88
READ_CALENDAR	33	0	0	67
READ_CONTACTS	8	0	0	92
READ_PHONE_STATE	39	26	18	17
READ_SMS	19	0	6	75
RECEIVE_MMS	0	25	0	75
RECORD_AUDIO	20	20	0	60
SEND_SMS	27	27	18	28
WRITE_CALL_LOG	12	12	0	76
WRITE_CONTACTS	22	11	0	67
WRITE_EXTERNAL_STORAGE	6	41	47	6

Appboy [67] (now branded as *Braze*). *Our Services are not directed to individuals under the age of 13. We do not knowingly collect personal information from such individuals without parental consent and require our Customers to fully comply with applicable law in the data collected from children under the age of 13.*

3.4.2 Who Handles Sensitive Data and What for

Android does not force developers to explain to users whether a given permission is needed by the app's functionality and if it is also accessed by an embedded third-party SDK. Therefore, users cannot make informed decisions on whether to grant a given permission to the application or not based on which software component will gain access to it. To know who is responsible for the data collection enabled by a given permission, we statically analyze the bytecode of the most recent version of each parental control app to search for Android API calls protected by Android permission and then attribute these calls to the host application's code or to an embedded third-party SDK. For that, we updated the mapping between permissions and protected API calls originally made by Au et al. to incorporate recent changes in Android's permission model [46, 49]. We rely on LibRadar's package identification to attribute the protected API requests to the actual app, or to any embedded third-party SDK.

Table 3.4 summarizes the results of this analysis. For each API call protected by dangerous permissions, we report whether it is used only in the application code, only in the library code, or by both. We have an additional column reporting the percentage of uses for which we cannot attribute API calls to each software component for reasons such as: *i*) the permission being requested by the app developer in its Android manifest file but not being invoked in the code; *ii*) our method misses relevant API calls because of incomplete information in Au's mapping [46] [6]; and *iii*) the use of code obfuscation techniques. [7]

[6] Despite our manual efforts to complement the mapping and incorporate recent changes in the Android permission control model, we cannot guarantee its completeness, since some mappings are not yet properly documented in the Android API.

[7] Seven instances of apps reading unique identifiers like the MAC address, SSID, and IMEI were identified in obfuscated

A significant number of the calls to dangerous permission-protected methods are invoked **only by embedded third-party SDKs**. For instance, embedded SDKs access the fine location permission and the read phone state permission, [8]— 9% and 26% of the times respectively—when the host app does not require access to them. In other cases, SDKs piggyback on the set of permissions already requested by the host app: e.g., 55% of the times that fine location is requested by the app, a third-party SDK also access the permission. This suggests that users' personal data might be collected and processed for secondary usages beyond those offered by the app.

On a per-app basis, the apps *MMGuardian* and *Mobilefence* access users' geolocation only in application code but, for other apps, only in third-party code. Specifically, we observe that the library *com.proximity*—Google's proximity beacon—is embedded in the app *ShieldMyTeen*. This case might be justified for the need to leverage geo-fencing methods. The app *GPS Tracker* leaks the location in code that belongs to the library *Arity* [44], which is a platform that collects location information to better understand and improve mobility. While we find this library highly unusual for parental control apps, we find that this apps is a "Family GPS tracker" which can be used by parents as well. The developers are very explicit in their privacy policy about sharing data with this company for driving features present in the app.

After analyzing the potential misbehavior and privacy risks posed by parental control apps and their embedded SDKs, we now rely on dynamic analysis to bypass some of the static analysis' limitations, reporting evidence of the collection and dissemination of children's personal data by these third-party SDKs.

3.5 Dynamic analysis

The results of our static analysis show that some parental control apps and embedded third-party SDKs potentially collect and disseminate personal children data to the Internet. In this section, we use dynamic analysis methods to complement our static analysis and collect actual evidence of personal data dissemination to the Internet. To that end, we perform the following user actions to execute and test each app —or pair of apps if there is a parent or companion version—in our database:

1. We install the app and go through the setup process. We create a parent account when necessary, and grant consent to collect sensitive data when asked. In order to perform an apples-to-apples comparison, when prompted, we configure each app to monitor (or block) the activities of a child below 13 years of age. It is also important to note that we run all tests from a European country (Spain) and that when creating the account we are consenting to the privacy policy and terms of services of the app.
2. Due to scalability issues, we cannot test the whole spectrum of children actions that can be blocked or monitored by a parental control app. Therefore, we decided to focus on those that are more likely to be blocked based on their permission requests or advertised features. Specifically, we manually interact with the children device for five minutes as follows: (*i*) we visit a newspaper in the browser (this action is typically allowed for children), but also one pornographic website, and a gambling service (this type of traffic should be typically banned); (*ii*) we open a permitted child game and a potentially blacklisted dating app; (*iii*) we install and

code, thus they could not be attributed.

[8] A permission that allows reading unique device identifiers.

Table 3.5: Most popular third parties by apps contacting them

Third party	Type	# apps
Crashlytics	Crash Reporting, Analytics	23
Facebook Graph	Social Network, Ads, Analytics	15
Appsflyer	Analytics	5
Adjust	Ad Fraud, Marketing, Analytics	3
Google ads	Advertising	3
OneSignal	Push Notifications	3

uninstall a game; (iv) we take a picture with the default camera app and (v)—while the test phone does not have a SIM card—we go to the phone and SMS apps and attempt to make a phone call and send a message.

Our analysis estimates a lower bound of all the possible instances of personal data dissemination that might exist in our dataset of parental control apps when compared to our static analysis. Also, since the Android VPN API only allows one app to create a virtual interface at a given time [24], our method does not allow us to fully test 3 parental control apps that require access to the VPN interface to monitor the children's network communications. We still report any data dissemination that might happen prior to or during the VPN interface being setup.

3.5.1 Connections to Third-party Services

Our results confirm the high presence of network connections to third-party components in children-oriented apps as summarized in Table 3.5. We observe network connections to a variety of destinations in the recorded network flows. We find 49 different second-level domains across all apps, 18 of which are associated with third-party Advertisement and Tracking Services services according to the list developed by Razaghpanah et al. [230]. The most contacted domain is *Crashlytics*—now Firebase—which is a Google-owned bug reporting and analytics service contacted by 54.8% of apps. Every third-party domain found in our dynamic analysis experiments belongs to SDKs previously reported by our static analysis method. This also verifies some of our claims. For instance, dynamic analysis confirms that the app *GPS Tracker* contacts *Branch* services, an analytics platform to increase revenues through "links built to acquire, engage, and measure across all devices, channels, and platforms" [64] that is not supposed to be used in children-oriented software according to their own ToS.

3.5.2 Personal Data Collection and Dissemination

Using Lumen we identify 513 flows containing personal data (i.e., resettable and persistent unique identifiers, geolocation, WiFi and Access Point information which can be used as a proxy for geolocation, and list of packages installed) generated by 42 (91%) different apps. We note that the access to the WiFi AP is a known side-channel that has been used as a proxy to access users' geo-location [233].

The fact that we see private data in network traffic does not necessarily imply a privacy violation. This information might be uploaded to the cloud to deliver its intended service, and to inform the companion app or the parent directly through a web dashboard. In fact, 74% of these apps fall in the

Table 3.6: Data dissemination without consent

Data Dissemination Type	**Count (Unique SLD)**	
	1st parties	**3rd parties**
Android Advertisement ID	2	8
Android ID	4	11
AP MAC address	1	0
WiFi SSID	2	0
IMEI	4	1
Geolocation	0	1
Email	2	0

monitoring category, which are those that request a higher set of permissions. Yet, some types of data like unique IDs and location are extremely sensitive and should not be shared with third-party services, particularly those offering advertising and tracking services that do not comply with child privacy rules. We observe 4 apps disseminating the location to a third-party domain in. Examples of third parties collecting location information are analytics SDKs used for client engagement and app growth (*Amplitude*), and mobile push notification services (*OneSignal*). Other usages might be for providing a service to the parent, as in the case of mapping APIs (*Google Maps*). None of these providers are in Google's list of SDKs suitable for children oriented apps [128].

Google encourages developers to use the Android Advertisement ID (AAID) as the only user identifier [18]. The AAID is non-persistent and users can reset it or opt out of "Ads personalization" in their device. The combination of the AAID with other persistent identifiers without explicit consent from the user is also a violation of Google's Terms of Service [16]. Despite this, we find 24 apps collecting and sharing persistent identifiers such as the IMEI, and 58% apps uploading the AAID along with persistent identifiers, hence defeating the purpose of resettable IDs. This behavior has been previously reported for regular children apps published in the DFF program [238]. Looking further into this issue we find that half of the apps sending the AAID alongside another unique identifier do so to a third-party SDK.

3.5.3 User Consent

Both COPPA and GDPR state that companies must obtain verifiable parental consent before gathering data from children below the age limit (13 years of age for COPPA, 16 for GDPR) [77, 101]. App developers should obtain verifiable parental (or legal tutor) consent before collecting sensitive personal data from the minor using techniques such as sending an email, answering a set of questions, or providing an ID document or credit card details at runtime [102]. This legal requirement implies that informing parents or legal tutors about data collection practices on the privacy policy is not enough, especially if the app disseminates sensitive data to third-party services as previously discussed.

In our previous analysis, we consented to data collection in all our experiments by creating an account and operating the app impersonating a child. Nevertheless, we want to determine whether apps collect private data without parental consent. To do so, we rely on the automatic method previously proposed by Reyes et al. [238]: we launch each app and run it for five minutes *without* interacting with it. This implies that we *do not* actively consent to data collection and we do not carry out any of the children actions, opting instead to leave the app running with no input. As a result, any sensitive or

Table 3.7: Apps sending sensitive data without encryption

Application	Data Type	Destination
com.kidoz	AAID	kidoz.net
ru.kidcontrol.gpstracker	IMEI & Location	85.143.223.160
com.parentycontrol	Email	parentycontrol.com
com.safekiddo.kid	IMEI	safekiddo.com
com.kiddoware.kidsplace	Android ID	kiddoware.com
com.kiddoware.kidsafebrowser	Android ID & Hardware ID	kiddoware.com

personal data—particularly unique identifiers and geolocation—uploaded by the app to third parties may be a potential violation of COPPA and GDPR.

We find 67% apps disseminating personal data without explicit and verifiable consent. The information shared by these apps includes unique identifiers (i.e., the android serial id, the AAID, or the IMEI) and the location of the user (including alternative location methods such as the SSID or the AP MAC address which have been prosecuted by the U.S. FTC before [104, 238, 233]). 47% of all cases of non-consented data dissemination are going to a third party. Table 3.6 summarizes the data types disseminated by the tested apps, grouped by the type of data being disseminated. We note that none of these libraries are in Google's list of certified suitable for children SDKs [128]. We believe that some of these instances may correspond to developers being careless in the way that they integrate third-party SDKs, not making sure that the user has read and accepted their data collection methods before starting to collect data.

3.5.4 (Lack of) Secure Communications

Both the COPPA [101] rule and the GDPR [77] have clear provisions stating that developers must treat data about children with an appropriate degree of security [117, 151]. To assess that this is the case, we study whether parental control apps make use of encryption (e.g., TLS) to upload the data to the cloud. Table 3.7 summarizes the non-encrypted flows, sorted by pairs of apps and domains, as well as the type of sensitive data that is transmitted in the clear. We find instances of persistent identifiers that enable tracking of users, e.g., the IMEI and the AAID, or geolocation information, being sent in the clear. Finally, we observe one app (*Secure Kids*) uploading without encryption the list of installed packages (used at the server to enable parents to block unwanted apps) alongside an identifier-like string (DEV_ID). The app *com.kiddoware.kidsafebrowser* appears in our results because it is installed as the default web browser by one of the apps in the dataset.

3.6 Privacy Policy Analysis

Both GDPR and COPPA require app developers to provide clear terms of use and privacy policies to the end user. However, it is known that privacy policies may be difficult to understand—even yielding different interpretations—by average users [164, 216, 258, 295]. We manually inspect the privacy policies indexed by Google Play Store for each app in our corpus to analyze to what extent parental control app developers respect the transparency obligations enforced by regulation. To avoid biases introduced by ambiguous interpretations of the policies, two authors conduct independent manual analysis of each policy, and discuss with a third author in case of disagreement. We note that only

89% of app developers had published a privacy policy on their Google Play profile at the time of conducting our study in February 2018. We discuss the results of our privacy policy analysis along four axis, including examples of wording found in the privacy policies as examples:

General considerations. We first look at whether the policy can be found easily on Google Play, if it provides clear information about the company (i.e., location of company, main purpose, other apps), and if users are notified upon policy changes. We find that 95% of apps provide a direct link to their policies in their Google Play profile, and 10% of apps provide information about the companies. (e.g., *The "Company" or "We/us", with domicile at [...]*) However, despite the fact that changes in privacy policies over time are hard to monitor by users, only 20% of the apps state in their policies that they actively notify users upon policy changes.

Data collection practices. We find that most privacy policies (92.6%) report the kind of data being collected from minors. However, only a few of them (54%) clearly inform users of how these data are processed and for what purpose (e.g., *Your account and contact data are used for managing our relationship with you*). We also see that only roughly half (56%) of the policies portray how long collected data will be stored at the developer's servers.

Data sharing practices. Our static and dynamic analysis show that many parental control mobile applications rely on third-party SDKs offering crash reporting, analytics, or advertising services. When any embedded third-party service collects or processes personal data from users, both GDPR and COPPA require developers to list them on the privacy policy of the app. We find that 59% of apps talk about third-party service usage in their privacy policy, but only 24% of the apps list the type of data being sent to them. Finally, companies can also share or sell data to other companies (e.g., data brokers). Even though we see that 78% of policies talk about the possibility of sharing customer's data, they say that this will only happen as a result of company acquisition or legal compulsion.

Third-party service disclosure. We verify whether apps are transparent when they refer to third-party service usage in their policies by cross-checking these policies with our empirical findings regarding the usage of third-party SDKs and data dissemination (§ 3.5). Since GDPR introduced the need to name the third-party companies receiving personal data, we check this in the 25 apps for which we have privacy policies collected from after the GDPR became effective on the 25th of May of 2018. Table 3.8 shows the number of apps using a third-party service and how many of them actually report this in their privacy policy (e.g., *Google may use the Data collected to contextualize and personalize the ads of its own advertising network*). Only 28.0% name all the third-party services that we find during runtime, while 79% of the studied apps do not name any third parties that they share private data with. We note that the latter apps are potentially in violation of both COPPA and GDPR for not being clear about their data sharing partners. Furthermore, we find one app that shares location data with a third-party service, which can constitute a potential violation of the FTC COPPA rule which prohibits the collection of children's geolocation sufficient to identify a street name and city or town. While this app is not directed only at children, its company name is *Family Safe Productions*, hinting that it can be used for monitoring child locations. Nevertheless, they openly say in their policy that they can share location data with third parties (as reported in § 3.4.2). We note

Table 3.8: Third party presence in apps and on their privacy policy

Third party	Type	# apps	# apps listing them in policy
Crashlytics	Crash Reporting, Analytics	23	5
Facebook Graph	Social Network, Advertising, Analytics	15	3
Appsflyer	Analytics	5	1
Adjust	Ad Fraud, Marketing, Analytics	3	0
Google ads	Advertising	3	1
OneSignal	Push Notifications	3	1
Amplitude	Analytics	2	0
Help Shift	Customer service	1	0
Apptentive	Analytics, User Engagement	1	0
Branch	Analytics	1	0
Splunk	Analytics	1	0

that we did not find this behavior in our dynamic analysis experiments.

Regulatory compliance. We also study developers' awareness about different related regulation. Only 22% of apps claim COPPA compliance in their policy; and, while only 10% of policies talk about European legislation, 37% mention their compliance with local laws (e.g., *Our Privacy Policy and our privacy practices adhere to the United States Children's Online Privacy Protection Act ("COPPA"), as well as other applicable laws*).

User privacy rights. Finally, we check the rights and choices that users have about their data. To do so we look at the number of apps that allow users to opt out of the data collection practices without having to cease the use of the app, and how many apps give the user a chance to correct, delete, or access their data. We find that in 46% of apps users can opt out of data collection, where 63% of apps give users at least one of the above choices (e.g., *According to European Union applicable data protection legislation (GDPR), data subjects shall have the right to access to data, rectification, erasure, restriction on processing [...]*).

3.6.1 The GDPR Effect

We first analyzed the privacy policies at the beginning of this study, around three months before the GDPR became effective in May 2018. While we do not revisit the whole analysis, we use the results from our study to evaluate the evolution of the privacy policies for the 25 apps that share data with third parties and that are still available after the 25th of May 2018. Our goal is to learn if companies changed their policies, to understand the impact of the law in the specific case of parental control apps. Comparing the newer policies with the ones of our previous analysis, we find that 2 have major changes in their privacy policy after GDPR. Looking more in detail into the specific cases we find that one app initially did not name its data sharing partners in the privacy policy, and later added that info. Another app did not explain clearly the type of data being collected, and has now changed its policy to be clearer. However, we see that most policies are still unclear as they often omit names of third-party services used by the application. Thus, even when regulatory frameworks push to shed light into the third-party mobile ecosystem, we see the necessity for external auditing of mobile parental control applications beyond usability, and capability aspects.

3.7 Discussion

Recommendations for Public Bodies. Given the severe privacy risks revealed by our analysis of parental control apps (§ 3), we consider that privacy implications should be a fundamental part of any guide designed for parents. However, our findings show that privacy is not an aspect that is taking into account by public bodies when recommending tools to parents. Our dataset contains 5 of the 10 apps benchmarked in the SIP benchmark [89] by the European commission, which focuses on usability and resistance to children sidestepping attempts. Two of these apps follow privacy-risky practices: *Qustodio* (also mentioned on a large amount of online studies that recommend top parental control apps [220, 240]) shares data with third-parties without consent, and *Parentsaround* does so in addition of sharing unique identifiers with third-party services. We also compare our results to IS4K Cybersecurity, which lists a series of apps, enumerating their functionalities without any judgment on their suitability. Our dataset includes 6 out of the 10 apps in their list. Three of these apps share sensitive data with third parties without appropriate consent, and another app sends data unencrypted.

Recommendations for App Stores. We argue that app stores should take extra measures for verifying that applications directed at children comply with current legislation and treat children data with extreme care even if they are not in the designed for families program. While a complete and exhaustive analysis of apps could be unfeasible, static analysis showing the presence of analytics and advertisement libraries in apps that will be used by minors—including SDKs that prohibit their use in children apps—should raise concerns. Furthermore, simple dynamic analysis to verify that these apps use the most basic security measures, such as the use of encryption, would already help reducing the risks for children privacy. Much like in the case of children-oriented apps present in Google's DFF program, parental control apps should comply with children-oriented regulation and special provisions. By definition, they are directed at children. However, none of the apps analyzed in this study is listed on Google Play's DFF program. While previous work has shown that Android app's tend to collect plenty of user data [215, 230], in some cases even avoiding the security mechanisms implemented by the platform [233, 195], the fact that parental control apps potentially do not comply with specific regulation is much more worrisome—both for parents and minors.

Recommendations for parents. There is a big element of trust from parents towards parental control solution developers, which is broken when these services treat children data carelessly, share it with third-parties, or send it across the Internet unencrypted. This misbehavior and parents' inability to audit mobile applications calls for action from researchers and data protection agencies to understand the way in which these apps might be violating current legislation and invoke regulatory actions to fix these issues. Until that moment, some studies proposed the use of non-technical solutions for parental control [186]. We argue that, indeed, relying on these types of solutions would improve children's and parent's privacy alike. However, a more sensible middle ground for parents might be the use of blocking solutions in favor of monitoring ones. As we have shown in this thesis, the later tend to request more permissions and collect more personal data from children than the former.

Comparison to apps of different nature. In this thesis we show that parental control applications often misbehave posing privacy threats for children and even parents. However, we cannot say that they are worse in behavior than other Android apps, even children-oriented apps studied in the literature [230, 238]. Previous studies show that security and privacy misbehavior is a common trend in Android applications: they often request more permissions than needed [106], and include a large number of third-party SDKs [188, 230]. Regarding the privacy risks of children-oriented applications published in Google Play's Designed for Families program, Irwin et al. showed that almost 50% of the apps were in potential violation of the COPPA rule [238].

Much like in the case of children-oriented apps present in Google's DFF program, parental control apps should comply with children-oriented regulation and special provisions. By definition, they are directed at children. However, none of the apps analyzed in this study is listed on Google Play's DFF program. While previous work has shown that Android app's tend to collect plenty of user data [215, 230], in some cases even avoiding the security mechanisms implemented by the platform [233, 195], the fact that parental control apps potentially do not comply with specific regulation is much more worrisome – both for parents and minors.

In fact, there is a big element of trust from parents towards parental control solution developers, which is broken when these services treat children data carelessly, share it with third-parties, or send it across the Internet unencrypted. This misbehavior and parents' inability to audit mobile applications calls for action from researchers and data protection agencies to understand the way in which these apps might be violating current legislation and invoke regulatory actions to fix these issues. Until that moment, some studies proposed the use of non-technical solutions for parental control [186].

Limitations. Our app collection method relies on using the free search capabilities of the Google Play Store, which means that we did not explore every parental control app on the market. Nevertheless, we believe that the set of apps chosen is representative in popularity, developer characteristics, and behavior. Also, as our initial data collection started two years prior to us finishing and submitting this work, some apps were removed from the marketplace during our analysis and new apps had appeared. We repeated the data collection pipeline later on our study to find newly published apps, and we downloaded historical version of every app that we analyzed to always cover the same time-frame.

Despite combining static and dynamic analysis, we acknowledge that our analysis cannot guarantee full coverage of the code, app features, or the data flows. First, the static analysis will miss code paths implemented in native or obfuscated code. Second, LibRadar uses a database to identify third-party SDKs and cannot detect libraries that are not present in this database. Third, Lumen can miss flows that are not triggered by our pre-defined actions. Furthermore, we perform our dynamic analysis on the children app, and it may be that the companion parent app disseminates sensitive data over the network,. Additionally, our Man-in-the-middle proxy is unable to intercept TLS flows for 2 apps that might use techniques like TLS certificate-pinning [232]. Finally, our dynamic analysis has been executed from a testbed geolocated in the European Union. It would be necessary to verify our claims in the U.S. to further investigate potential COPPA violations.

Chapter 4 Security Analysis: Apps' Circumvention of the Android Permissions System

Previous work has shown that the Android permission system is not foul-proof, and that different actors can obtain unauthorized access to protected resources via different attacks [61, 206, 259]. One such example are covert and side channels that previous studies have discovered. However, covert and side channels on Android have remained more of a theoretical attack than a true threat to the privacy of users, due to their complicated and highly technical nature [84, 61, 172].

In line with the hypothesis of this thesis ($\mathcal{H}$ 1), we designed a pipeline for automatically discovering vulnerabilities in the Android permissions system through a combination of dynamic and static analysis, in effect creating a scalable honeypot environment. Namely, we use a highly instrumented Android version that allows us to crosscheck the type of data that is shared during runtime with the permissions that we granted to the app. This allows us to find mismatches for which we manually analyze the code of the app, in order to understand the cover or side channel used. Once we have discovered the way in which unauthorized access happens, we create a static fingerprint of the vulnerability and statically find other misbehaving apps and SDKs. We tested our pipeline on more than 88,000 apps and discovered covert and side channels used in the wild that compromise both users' location data and persistent identifiers, which we responsibly disclosed.

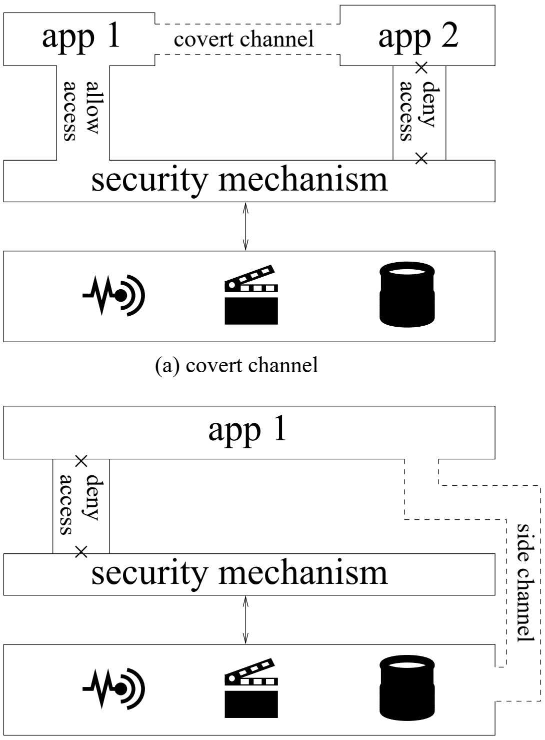
This chapter is organised as follows: we first explain how covert and side channels can be used to circumvent Android's permission model (§ 4.1), we then present our novel methodology based on static and dynamic analysis (§ 4.2) and report on the specific covert and side channels that we discover on the wild (§ 4.3). We conclude with a discussion about the limitations and future work (§ 4.4) and the legal implications of our findings (§ 4.5).

4.1 Circumvention of the Permission Model

Apps can circumvent the Android permission model in different ways, including apps colluding to share data without user awareness, or apps using unprotected resources to infer personal data from users [247, 7, 61, 206, 259, 257, 195, 194]. These practices are called covert and side channels, and Figure 10 illustrates the difference between these two types of attacks and shows how an app that is denied permission by a security mechanism is able to still access that information.

In this thesis, we build on a vast literature in the field of covert- and side-channel attacks for Android. However, while prior studies generally only reported isolated instances of such attacks or approached the problem from a theoretical angle, our work combines static and dynamic analysis to automatically detect real-world instances of mis-behaviors and attacks.

Covert Channels. A covert channel is a communication path between two parties (e.g., two mobile apps) that allows them to transfer information that the relevant security enforcement mechanism deems the recipient unauthorized to receive [69]. For example, imagine that AliceApp has been



(a) covert channel

(b) side channel

Figure 10: Covert and side channels. (a) A security mechanism allows `app1` access to resources but denies `app2` access; this is circumvented by `app2` using `app1` as a facade to obtain access over a communication channel not monitored by the security mechanism. (b) A security mechanism denies `app1` access to resources; this is circumvented by accessing the resources through a side channel that bypasses the security mechanism.

granted permission through the Android API to access the phone's IMEI (a persistent identifier), but BobApp has been denied access to that same data. A covert channel is created when AliceApp legitimately reads the IMEI and then gives it to BobApp, even though BobApp has already been denied access to this same data when requesting it through the proper permission-protected Android APIs.

In the case of Android, different covert channels have been proposed to enable communication between apps. This includes exotic mediums such as ultrasonic audio beacons and vibrations [84, 61]. Apps can also communicate using an external network server to exchange information when no other opportunity exists. Our work, however, exposes that rudimentary covert channels, such as shared storage, are being used in practice at scale.

Side Channels. A side channel is a communication path that allows a party to obtain privileged information without relevant permission checks occurring. This can be due to non-conventional unprivileged functions or features, as well as ersatz versions of the same information being available without being protected by the same permission. A classical example of a side channel attack is the timing attack to exfiltrate an encryption key from secure storage [172]. The system under attack is an algorithm that performs computation with the key and unintentionally leaks timing information—i.e., how long it runs—that reveals critical information about the key.

Side channels are typically an unintentional consequence of a complicated system. ("Backdoors" are intentionally-created side channels that are meant to be obscure.) In Android, a large and complicated API results in the same data appearing in different locations, each governed by different access control mechanisms. When one API is protected with permissions, another unprotected method may be used to obtain the same data or an ersatz version of it. Marforio et al. [191] proposed several scenarios to transmit data between two Android apps, including the use of UNIX sockets and external storage as a shared buffer. In our work we see that the shared storage is indeed used in the wild. Other studies have focused on using mobile noises [84, 247] and vibrations generated by the phone (which could be inaudible to users) as covert channels [7, 61]. Such attacks typically involve two physical devices communicating between themselves. This is outside of the scope of our work, as we focus on device vulnerabilities that are being exploited by apps and third parties running in user space.

Spreitzer et al. provided a good classification of mobile-specific side-channels present in the literature [259]. Previous work has demonstrated how unprivileged Android resources could be to used to infer personal information about mobile users, including unique identifiers [257] or gender [195]. Researchers also demonstrated that it may be possible to identify users' locations by monitoring the power consumption of their phones [194] and by sensing publicly available Android resources [307]. More recently, Zhang et al. demonstrated a sensor calibration fingerprinting attack that uses unprotected calibration data gathered from sensors like the accelerometer, gyroscope, and magnetometer [302]. Others have shown that unprotected system-wide information is enough to infer input text in gesture-based keyboards [257]. Research papers have also reported techniques that leverage lowly protected network information to geolocate users at the network level [206, 103, 2]. In this thesis, we extend previous work by reporting third-party libraries and mobile applications that gain access to unique identifiers and location information in the wild by exploiting side and covert channels.

4.2 Testing Environment and Analysis Pipeline

In order to discover how the Android permission system can be manipulated to access protected resources and data without user consent via covert and side channels, we rely on the instrumentation and processing pipeline depicted and described in Figure 11. In line with our hypothesis, this methodology combines the advantages of both static and dynamic analysis techniques to triage suspicious apps and analyze their behaviours in depth. We executed each app version individually on a physical mobile phone equipped with a customized operating system and network monitor. This testbed allows us to observe apps' runtime behaviours both at the OS and network levels. We can observe how apps request and access sensitive resources and their data sharing practices. We also have a comprehensive data analysis tool to de-obfuscate collected network data to uncover potential deceptive practices (§ 4.2.2).

Before running each app, we gather the permission-protected identifiers and data. We then execute each app while collecting all of its network traffic. We apply a suite of decodings to the traffic flows and search for the permission-protected data in the decoded traffic. We record all transmissions and later filter for those containing permission-protected data sent by apps not holding the requisite permissions. We hypothesize that these are due to the use of side and covert channels; that is, we are not looking for these channels, but rather looking for evidence of their use (§ 4.2.3). Then, we group the suspect transmissions by the data type sent and the destination where it was sent, because we found that the same data-destination pair reflects the same underlying side or covert

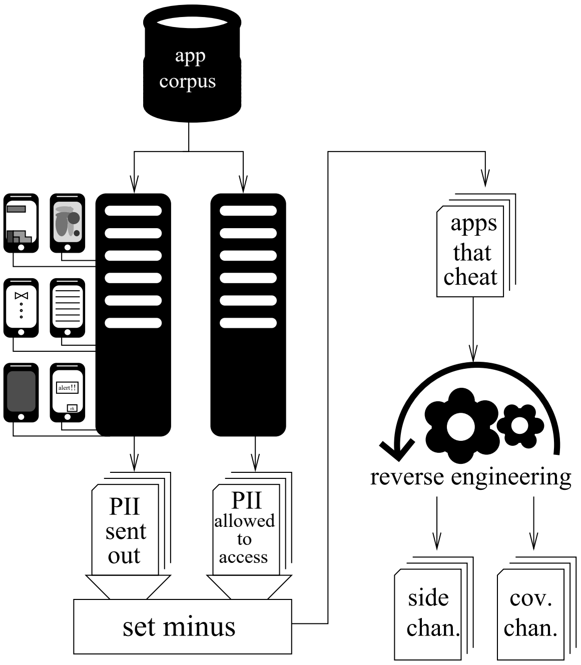


Figure 11: Overview of our analysis pipeline. Apps are automatically run and the transmissions of sensitive data are compared to what would be allowed. We manually analyze the code of those suspected of using a side or covert channel.

z

channel. We take one example per group and manually analyze its code to determine how the app gained permission-protected information without the corresponding permission (§ 4.2.4).

Finally, we fingerprint the apps and libraries found using covert- and side-channels to identify the static presence of the same code in other apps in our corpus. A fingerprint is any string constant, such as specific filename or error message, that can be used to statically analyze our corpus to determine if the same technique exists in other apps that did not get triggered during our dynamic analysis phase. Therefore, we use dynamic analysis to find evidence of potential covert and side channels by apps. Then, we rely on static code analysis to figure out the specific attack being used, and fingerprint the code to find other instances of such attacks across our dataset. We used this testing environment to find evidence of covert- and side-channel usage in 252,864 versions of 88,113 different Android apps, all of them downloaded from the U.S. Google Play Store using a purpose-built Google Play scraper.

4.2.1 App Collection

We wrote a Google Play Store scraper to download the most-popular apps under each category. Because the popularity distribution of apps is long tailed, our analysis of the 88,113 most-popular apps is likely to cover most of the apps that people currently use. This includes 1,505 non-free apps we purchased for another study [149]. We instrumented the scraper to inspect the Google Play Store to obtain application executables (APK files) and their associated metadata (e.g., number of installs, category, developer information, etc.).

As developers tend to update their Android software to add new functionality or to patch bugs [236], these updates can also be used to introduce new side and covert channels. Therefore, it is important to examine different versions of the same app, because they may exhibit different behaviours.

In order to do so, our scraper periodically checks if a new version of an already downloaded app is available and downloads it. This process allowed us to create a dataset consisting of 252,864 different versions of 88,113 Android apps.

4.2.2 Dynamic Analysis Environment

We implemented the dynamic testing environment described in Figure 11, which consists of about a dozen Nexus 5X Android phones running an instrumented version of the Android Marshmallow platform.[1] This purpose-built environment allows us to comprehensively monitor the behaviour of each of 88,113 Android apps at the kernel, Android-framework, and network traffic levels. We execute each app automatically using the Android Automator Monkey [23] to achieve scale by eliminating any human intervention. We store the resulting OS-execution logs and network traffic in a database for offline analysis, which we discuss in Section 4.2.3. The dynamic analysis is done by extending a platform that we have used in previous work [238].

Platform-Level Instrumentation. We built an instrumented version of the Android 6.0.1 platform (Marshmallow). The instrumentation monitored resource accesses and logged when apps were installed and executed. We ran apps one at a time and uninstalled them afterwards. Regardless of the obfuscation techniques apps use to disrupt static analysis, no app can avoid our instrumentation, since it executes in the system space of the Android framework. In a sense, our environment is a honeypot allowing apps to execute as their true selves. For the purposes of preparing our bug reports to Google for responsible disclosure of our findings, we retested our findings on a stock Pixel 2 running Android Pie—the most-recent version at the time—to demonstrate that they were still valid.

Kernel-Level Instrumentation. We built and integrated a custom Linux kernel into our testing environment to record apps' access to the file system. This module allowed us to record every time an app opened a file for reading or writing or unlinked a file. Because we instrumented the system calls to open files, our instrumentation logged both regular files and special files, such as device and interface files, and the `proc/` filesystem, as a result of the *"everything is a file"* UNIX philosophy. We also logged whenever an `ioctl` was issued to the file system. Some of the side channels for bypassing permission checking in the Android platform may involve directly accessing the kernel, and so kernel-level instrumentation provides clear evidence of these being used in practice.

We ignored the special device file `/dev/ashmem` (Android-specific implementation of asynchronous shared memory for inter-process communication) because it overwhelmed the logs due to its frequent use. As Android assigns a separate *user* (i.e., uid) to each app, we could accurately attribute the access to such files to the responsible app.

Network-Level Monitoring. We monitored all network traffic, including TLS-secured flows, using a network monitoring tool developed for our previous research activities [231]. This network monitoring module leverages Android's VPN API to redirect all the device's network traffic through a localhost service that inspects all network traffic, regardless of the protocol used, through deep-packet inspection and in user-space. It reconstructs the network streams and ascribes them to the originating app

[1] While as of this writing Android Pie is the current release [136], Marshmallow and older versions were used by a majority of users at the time that we began data collection.

by mapping the app owning the socket to the UID as reported by the `proc` filesystem. Furthermore, it also performs TLS interception by installing a root certificate in the system trusted certificate store. This technique allows it to decrypt TLS traffic unless the app performs advanced techniques, such as certificate pinning, which can be identified by monitoring TLS records and proxy exceptions [232].

Automatic App Execution. Since our analysis framework is based on dynamic analysis, apps must be executed so that our instrumentation can monitor their behaviours. In order to scale to hundreds of thousands of apps tested, we cannot rely on real user interaction with each app being tested. As such, we use Android's UI/Application Exerciser Monkey, a tool provided by Android's development SDK to automate and parallelize the execution of apps by simulating user inputs (i.e., taps, swipes, etc.).

The Monkey injects a pseudo-random stream of simulated user input events into the app, i.e., it is a UI fuzzer. We use the Monkey to interact with each version of each app for a period of ten minutes, during which the aforementioned tools log the app's execution as a result of the random UI events generated by the Monkey. Apps are rerun if the operation fails during execution. Each version of each app is run once in this manner; our system also reruns apps if there is unused capacity.

After running the app, the kernel, platform, and network logs are collected. The app is then uninstalled along with any other app that may have been installed through the process of automatic exploration. We do this with a white list of allowed apps; all other apps are uninstalled. The logs are then cleared and the device is ready to be used for the next test.

4.2.3 Personal Information in Network Flows

Detecting whether an app has legitimately accessed a given resource is straightforward: we compare its runtime behaviour with the permissions it had requested. Both users and researchers assess apps' privacy risks by examining their requested permissions. This presents an incomplete picture, however, because it only indicates what data an app *might* access, and says nothing about with whom it may share it and under what circumstances. The only way of answering these questions is by inspecting the apps' network traffic. However, identifying personal information inside network transmissions requires significant effort because apps and embedded third-party SDKs often use different encodings and obfuscation techniques to transmit data. Thus, it is a significant technical challenge to be able to de-obfuscate all network traffic and search it for personal information. This subsection discusses how we tackle these challenges in detail.

Personal Information. We define "personal information" as any piece of data that could potentially identify a specific individual and distinguish them from another. Online companies, such as mobile app developers and third-party advertising networks, want this type of information in order to track users across devices, websites, and apps, as this allows them to gather more insights about individual consumers and thus generate more revenue via targeted advertisements. For this reason, we are primarily interested in examining apps' access to the persistent identifiers that enable long-term tracking, as well as their geolocation information.

We focus our study on detecting apps using covert and side channels to access specific types of highly sensitive data, including persistent identifiers and geolocation information. Notably, the unauthorized collection of geolocation information in Android has been the subject of prior regulatory

Table 4.1: The types of personal information that we search for, the permissions protecting access to them, and the purpose for which they are generally collected. We also report the subsection in this thesis where we report side and covert channels for accessing each type of data, if found, and the number of apps exploiting each. The dynamic column depicts the number of apps that we directly observed inappropriately accessing personal information, whereas the static column depicts the number of apps containing code that exploits the vulnerability (though we did not observe being executed during test runs).

Data Type	**Permission**	**Purpose/Use**	**Subsection**	**Nº of Apps**		**Nº of SDKs**		**Channel Type**	
				Dynamic	**Static**	**Dynamic**	**Static**	**Covert**	**Side**
IMEI	READ_PHONE_STATE	Persistent ID	4.3.1	13	159	2	2	2	0
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.3.2	42	12,408	1	1	0	1
Email	GET_ACCOUNTS	Persistent ID	Not Found						
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found						
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found						
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3.3	5	355	2	10	0	2
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found						
GPS	ACCESS_FINE_LOCATION	Location Data	4.3.4	1	1	0	0	0	1

action [103]. Table 4.1 shows the different types of personal information that we look for in network transmissions, what each can be used for, the Android permission that protects it, and the subsection in this thesis where we discuss findings that concern side and covert channels for accessing that type of data.

Decoding Obfuscations. In our previous work [238], we found instances of apps and third-party libraries (SDKs) using obfuscation techniques to transmit personal information over the network with varying degrees of sophistication. To identify and report such cases, we automated the decoding of a standard suite of standard HTTP encodings to identify personal information encoded in network flows, such as gzip, base64, and ASCII-encoded hexadecimal. Additionally, we search for personal information directly, as well as the MD5, SHA1, and SHA256 hashes of it.

After analyzing thousands of network traces, we still find new techniques SDKs and apps use to obfuscate and encrypt network transmissions. While we acknowledge their effort to protect users' data, the same techniques could be used to hide deceptive practices. In such cases, we use a combination of code and static analysis to understand the precise technique. We frequently found a further use of AES encryption applied to the payload before sending it over the network, often with hard-coded AES keys.

A few libraries followed best practices by generating random AES session keys to encrypt the data and then encrypt the session key with a hard-coded RSA public key, sending both the encrypted data and encrypted session key together. To de-cipher their network transmissions, we instrumented the relevant Java libraries. We found two examples of third-party SDKs "encrypting" their data by XOR-ing a keyword over the data in a Viginère-style cipher. In one case, this was *in addition* to both using standard encryption for the data *and* using TLS in transmission. Other interesting approaches included reversing the string after encoding it in base64 (which we refer to as "46esab"), using base64 multiple times (basebase6464), and using a permuted-alphabet version of base64 (sa4b6e). Each new discovery is added to our suite of decodings and our entire dataset is then re-analyzed.

4.2.4 Finding Side and Covert Channels

Once we have examples of transmissions that suggest the permission system was violated (i.e., data transmitted by an app that had not been granted the requisite permissions to do so), we then analyze the code of the app to determine how it circumvented the permissions system. Finally, we use static analysis to measure how prevalent this practice is among the rest of our corpus.

Code Analysis. After finding a set of apps exhibiting behaviour consistent with the existence of side and covert channels, we manually analyze their code to determine how the app actually obtained information outside of the permission system. We note that this is a time consuming and not easily automated process, yet necessary. Since many of the transmissions are caused by the same SDK code, we only needed to find the code used for each unique *circumvention technique*: not every app, but instead for a much smaller number of unique SDKs. The destination endpoint for the network traffic typically identifies the SDK responsible.

During the code analysis process, our first step was to use apktool[36] to decompile and extract the smali bytecode for each suspicious app. This allowed us to analyse and identify where any strings containing PII were created and from which data sources. For some particular apps and libraries, our work also necessitated the analysis of C++ code; for we used IdaPro[1].

The typical process was to search the code for strings corresponding to destinations for the network transmissions and other aspects of the packets. This revealed where the data was already in memory, and then static analysis of the code revealed where that value first gets populated. As intentionally-obfuscated code is more complicated to analyze, we also added logging statements for data and stack traces as new bytecode throughout the decompiled app, recompiled it, and ran it dynamically to get a sense of how it worked.

Measuring Prevalence. The final step of our process was to determine the prevalence of the particular side or covert channel in practice. We used our code analysis to craft a unique fingerprint that identifies the presence of an exploit in an embedded SDK, which is also robust against false positives. For example, a fingerprint is a string constant corresponding to a fixed encryption key used by one SDK, or the specific error message produced by another SDK if the operation fails.

We then decompiled all of the apps in our corpus and searched for the string in the resulting files. Within smali bytecode, we searched for the string in its entirety as a `const-string` instruction. For shared objects libraries like Unity, we use the `strings` command to output its printable strings. We include the path and name of the file as matching criteria to protect against false positives. The result is a set of all apps that may also exploit the side or covert channel in practice but for which our instrumentation did not flag for manual investigation, e.g., because the app had been granted the required permission, the Monkey did not explore that particular code branch, etc.

4.3 Results

In this section, we present our results grouped by the type of permission that should be held to access the data; first we discuss covert and side channels enabling the access to persistent user or device IDs (particularly the IMEI and the device MAC address) and we conclude with channels used for accessing users' geolocation (e.g., through network infrastructure or metadata present in multimedia content).

Our testing environment allowed us to identify five different types of side and covert channels in use among the 88,113 different Android apps in our dataset. Table 4.1 summarizes our findings and reports the number of apps and third-party SDKs that we find exploiting these vulnerabilities in our dynamic analysis and those in which our static analysis reveals code that *can* exploit these channels. Note that this latter category—those that *can* exploit these channels—were not seen as doing so by our instrumentation; this may be due to the Automator Monkey not triggering the code to exploit it or because the app had the required permission and therefore the transmission was not deemed suspicious.

4.3.1 IMEI

The International Mobile Equipment Identity (IMEI) is a numerical value that identifies mobile phones uniquely. The IMEI has many valid and legitimate operational uses to identify devices in a 3GPP network, including the detection and blockage of stolen phones.

The IMEI is also useful to online services as a persistent device identifier for tracking individual phones. The IMEI is a powerful identifier as it takes extraordinary efforts to change its value or even spoof it. In some jurisdictions, it is illegal to change the IMEI [147]. Collection of the IMEI by third parties facilitates tracking in cases where the owner tries to protect their privacy by resetting other identifiers, such as the advertising ID.

Android protects access to the phone's IMEI with the READ_PHONE_STATE permission. We identified two third-party online services that use different covert channels to access the IMEI when the app does not have the permission required to access the IMEI.

Salmonads and External Storage. Salmonads is a "third party developers' assistant platform in Greater China" that offers analytics and monetization services to app developers [242]. We identified network flows to `salmonads.com` coming from five mobile apps that contained the device's IMEI, despite the fact that the apps did not have permission to access it.

We studied one of these apps and confirmed that it contained the Salmonads SDK, and then studied the workings of the SDK closer. Our analysis revealed that the SDK exploits covert channels to read this information from the following hidden file on the SD card: `/sdcard/.googlex9/.xamdecoq0962`. If not present, this file is created by the Salmonads SDK. Then, whenever the user installs another app with the Salmonads SDK embedded and with legitimate access to the IMEI, the SDK—through the host app—reads and stores the IMEI in this file.

The covert channel is the apps' shared access to the SD card. Once the file is written, all other apps with the same SDK can simply read the file instead of obtaining access through the Android API, which is regulated by the permission system. Beyond the IMEI, Salmonads also stores the advertising ID—a resettable ID for advertising and analytics purposes that allows opting out of interest-based advertising and personalization—and the phone's MAC address, which is protected with the ACCESS_NETWORK_STATE permission. We modified the file to store new random values and observed that the SDK faithfully sent them onwards to Salmonads' domains. The collection of the advertising ID alongside other non-resettable persistent identifiers and data, such as the IMEI, undermines the privacy-preserving feature of the advertising ID, which is that it can be *reset*. It also may be a violation of Google's Terms of Service [16],

Our instrumentation allowed us to observe five different apps sending the IMEI without permission to Salmonads using this technique. Static analysis of our entire app corpus revealed that six apps contained the `.xamdecoq0962` filename hardcoded in the SDK as a string. The sixth app had been granted the required permission to access the IMEI, which is why we did not initially identify it, and so it may be the app responsible for having initially written the IMEI to the file. Three of the apps were developed by the same company, according to Google Play metadata, while one of them has since been removed from Google Play. The lower bound on the number of times these apps were installed is 17.6 million, according to Google Play metadata.

Baidu and External Storage. Baidu is a large Chinese corporation whose services include, among many others, an online search engine, advertising, mapping services [53], and geocoding APIs [52]. We observed network flows containing the device IMEI from Disney's Hong Kong Disneyland park app (com.disney. hongkongdisneyland_goo) to Baidu domains. This app helps tourists to navigate through the Disney-themed park, and the app makes use of Baidu's Maps SDK. While Baidu Maps initially only offered maps of mainland China, Hong Kong, Macau and Taiwan, as of 2019, it now provides global services.

Baidu's SDK uses the same technique as Salmonads to circumvent Android's permission system and access the IMEI without permission. That is, it uses a shared file on the SD card so one Baidu-containing app with the right permission can store it for other Baidu-containing apps that do not have that permission. Specifically, Baidu uses the following file to store and share this data: `/sdcard/backups/.SystemConfig/ .cuid2`. The file is a base64-encoded string that, when decoded, is an AES-encrypted JSON object that contains the IMEI as well as the MD5 hash of the concatenation of "com.baidu" and the phone's Android ID.

Baidu uses AES in CBC mode with a static key and the same static value for the initialization vector (IV). These values are, in hexadecimal, `33303231323130326469637564696162`. The reason why this value is not superficially representative of a random hexadecimal string is because Baidu's key is computed from the binary representation of the ASCII string `30212102dicudiab`—observe that when reversed, it reads as `baidu cid 2012 12 03`. As with Salmonads, we confirmed that we can change the (encrypted) contents of this file and the resulting identifiers were faithfully sent onwards to Baidu's servers.

We observed eight apps sending the IMEI of the device to Baidu without holding the requisite permissions, but found 153 different apps in our repository that have hardcoded the constant string corresponding to the encryption key. This includes two from Disney: one app each for their Hong Kong and Shanghai (com.disney.shanghaidisneyland_goo) theme parks. Out of that 153, the two most popular apps were Samsung's Health (com.sec.android.app.shealth) and Samsung's Browser (com.sec.android.app.sbrowser) apps, both with more than 500 million installations. There is a lower bound of 2.6 billion installations for the apps identified as containing Baidu's SDK. Of these 153 apps, all but 20 have the READ_PHONE_STATE permission. This means that they have legitimate access to the IMEI and can be the apps that actually create the file that stores this data. The 20 that do not have the permission can only get the IMEI through this covert channel. These 20 apps have a total lower bound of 700 million installations.

4.3.2 Network MAC Addresses

The Media Access Control Address (MAC address) is a 6-byte identifier that is uniquely assigned to the Network Interface Controller (NIC) for establishing link-layer communications. However, the MAC address is also useful to advertisers and analytics companies as a hardware-based persistent identifier, similar to the IMEI.

Android protects access to the device's MAC address with the ACCESS_NETWORK_STATE permission. Despite this, we observed apps transmitting the device's MAC address without having permission to access it. The apps and SDKs gain access to this information using C++ native code to invoke a number of unguarded UNIX system calls.

Unity and IOCTLs. Unity is a cross-platform game engine developed by Unity Technologies and heavily used by Android mobile games [274]. Our traffic analysis identified several Unity-based games sending the MD5 hash of the MAC address to Unity's servers and referring to it as a `uuid` in the transmission (e.g., as an HTTP GET parameter key name). In this case, the access was happening inside of Unity's C++ native library. We analyzed the code of `libunity.so` to determine how it was obtaining the MAC address.

Reversing Unity's 18 MiB compiled C++ library is more involved than Android's bytecode. Nevertheless, we were able to isolate where the data was being processed precisely because it hashes the MAC address with MD5. Unity provided its own unlabelled MD5 implementation that we found by searching for the constant numbers associated with MD5; in this case, the initial state constants.

Unity opens a network socket and uses an `ioctl` (UNIX "input-output control") to obtain the MAC address of the WiFi network interface. In effect, `ioctl`s create a large suite of "numbered" API calls that are technically no different than well-named system calls like `bind` or `close` but used for infrequently used features. The behaviour of an `ioctl` depends on the specific "request" number. Specifically, Unity uses the SIOCGIFCONF[2] `ioctl` to get the network interfaces, and then uses the SIOCGIFHWADDR[3] `ioctl` to get the corresponding MAC address.

We observed that 42 apps were obtaining and sending to Unity servers the MAC address of the network card without holding the ACCESS_NETWORK_STATE permission. To quantify the prevalence of this technique in our corpus of Android apps, we fingerprinted this behaviour through an error string that references the `ioctl` code having just failed. This allowed us to find a total of 12,408 apps containing this error string, of which 748 apps do not hold the ACCESS_NETWORK_STATE permission.

4.3.3 Router MAC Address

Access to the WiFi router MAC address (BSSID) is protected by the ACCESS_WIFI_STATE permission. In Section 2, we exemplified side channels with router MAC addresses being ersatz location data, and discussed the FTC enacting millions of dollars in fines for those engaged in the practice of using this data to deceptively infer users' locations. Android Nougat added a requirement that apps hold an additional location permission to scan for nearby WiFi networks [137]; Android Oreo further required a location permission to get the SSID and MAC address of the connected WiFi network. Additionally, knowing the MAC address of a router allows one to link different devices that share Internet access,

[2] Socket ioctl get interface configuration

[3] Socket ioctl get interface hardware address

Table 4.2: SDKs seen sending router MAC addresses and also containing code to access the ARP cache. For reference, we report the number of apps and a lower bound of the total number of installations of those apps. We do this for all apps containing the SDK; those apps that *do not* have ACCESS_WIFI_STATE, which means that the side channel circumvents the permissions system; and those apps which *do* have a location permission, which means that the side channel circumvents location revocation.

	Contact	Incorporation	Total Prevalance		Wi-Fi Permission		No Location Permission	
SDK Name	**Domain**	**Country**	**(Apps)**	**(Installs)**	**(Apps)**	**(Installs)**	**(Apps)**	**(Installs)**
AIHelp	cs30.net	United States	30	334 million	3	210 million	12	195 million
Huq Industries	huq.io	United Kingdom	137	329 million	0	0	131	324 million
OpenX	openx.net	United States	42	1072 million	7	141 million	23	914 million
xiaomi	xiaomi.com	China	47	986 million	0	0	44	776 million
jiguang	jpush.cn	China	30	245 million	0	0	26	184 million
Peel	peel-prod.com	United States	5	306 million	0	0	4	206 million
Asurion	mysoluto.com	United States	14	2 million	0	0	14	2 million
Cheetah Mobile	cmcm.com	China	2	1001 million	0	0	2	1001 million
Mob	mob.com	China	13	97 million	0	0	6	81 million

which may reveal personal relations by their respective owners, or enable cross-device tracking.

Our analysis revealed two side channels to access the connected WiFi router information: reading the ARP cache and asking the router directly. We found no side channels that allowed for scanning of other WiFi networks. Note that this issue affects *all* apps running on recent Android versions, not just those without the ACCESS_WIFI_STATE permission. This is because it affects apps *without* a location permission, and it affects apps *with* a location permission that the user has not granted using the ask-on-first-use controls.

Reading the ARP Table. The Address Resolution Protocol (ARP) is a network protocol that allows discovering and mapping the MAC layer address associated with a given IP address. To improve network performance, the ARP protocol uses a cache that contains a historical list of ARP entries, i.e., a historical list of IP addresses resolved to MAC address, including the IP address and the MAC address of the wireless router to which the device is connected (i.e., its BSSID).

Reading the ARP cache is done by opening the pseudo file `/proc/net/arp` and processing its content. This file is not protected by any security mechanism, so any app can access and parse it to gain access to router-based geolocation information without holding a location permission. We built a working proof-of-concept app and tested it for Android Pie using an app that requests *no permissions*. We also demonstrated that when running an app that requests both the ACCESS_WIFI_STATE and ACCESS_COARSE_LOCATION permissions, when those permissions are denied, the app will access the data anyway. We responsibly disclosed our findings to Google in September, 2018.

We discovered this technique during dynamic analysis, when we observed one library using this method in practice: OpenX [214], a company that according to their website "creates programmatic marketplaces where premium publishers and app developers can best monetize their content by connecting with leading advertisers that value their audiences." OpenX's SDK code was not obfuscated and so we observed that they had named the responsible function `getDeviceMacAddressFromArp`. Furthermore, a close analysis of the code indicated that it would first *try* to get the data legitimately using the permission-protected Android API; this vulnerability is only used after the app has been explicitly denied access to this data.

OpenX did not directly send the MAC address, but rather the MD5 hash of it. Nevertheless, it is still trivial to compute a MAC address from its corresponding hash: they are vulnerable to a brute-force attack on hash functions because of the small number of MAC addresses (i.e., an upper bound of 48 bits of entropy).[4] Moreover, insofar as the router's MAC address is used to resolve an app user's geolocation using a MAC-address-to-location mapping, one need only to hash the MAC addresses in this mapping (or store the hashes in the table) and match it to the received value to perform the lookup.

While OpenX was the only SDK that we observed exploiting this side channel, we searched our entire app corpus for the string `/proc/net/arp`, and found multiple third-party libraries that included it. In the case of one of them, `igexin`, there are existing reports of their predatory behaviour [56]. In our case, log files indicated that after `igexin` was denied permission to scan for WiFi, it read `/system/xbin/ip`, ran `/system/bin/ifconfig`, and then ran `cat /proc/net/arp`. Table 4.2 shows the prevalence of third-party libraries with code to access the ARP cache.

Router UPnP. One SDK in Table 4.2 includes another technique to get the MAC address of the WiFi access point: it uses UPnP/SSDP discovery protocols. Three of Peel's smart remote control apps (tv.peel.samsung.app, tv.peel.smartremote, and tv.peel.mobile.app) connected to `192.168.0.1`, the IP address of the router that was their gateway to the Internet. The router in this configuration was a commodity home router that supports universal plug-and-play; the app requested the `igd.xml` (Internet gateway device configuration) file through port 1900 on the router. The router replied with, among other manufacturing details, its MAC address as part of its UUID. These apps also sent WiFi MAC addresses to their own servers and a domain hosted by Amazon Web Services.

The fact that the router is providing this information to devices hosted in the home network is not a flaw with Android *per se*. Rather it is a consequence of considering every app on every phone connected to a WiFi network to be on the trusted side of the firewall.

4.3.4 Geolocation

So far our analysis has showed how apps circumvent the permission system to gain access to persistent identifiers and data that can be used to infer geolocation, but we also found suspicious behaviour surrounding a more sensitive data source, i.e., the actual GPS coordinates of the device.

We identified 70 different apps sending location data to 45 different domains without having any of the location permissions. Most of these location transmissions were not caused by circumvention of the permissions system, however, but rather the location data was provided within incoming packets: ad mediation services provided the location data embedded within the ad link. When we retested the apps in a different location, however, the returned location was no longer as precise, and so we suspect that these ad mediators were using IP-based geolocation, though with a much higher degree of precision than is normally expected. One app explicitly used `www.googleapis.com`'s IP-based geolocation and we found that the returned location was accurate to within a few meters; again, however, this accuracy did not replicate when we retested elsewhere [224]. We did, however, discover one genuine side channel through photo EXIF data.

Shutterfly and EXIF Metadata. We observed that the Shutterfly app (com.shutterfly) sends precise geolocation data to its own server (`apcmobile.thislife.com`) without holding a location permission.

[4] Using commodity hardware, the MD5 for every possible MAC address can be calculated in a matter of minutes [165].

Instead, it sent photo metadata from the photo library, which included the phone's precise location in its exchangeable image file format (EXIF) data. The app actually processed the image file: it parsed the EXIF metadata—including location—into a JSON object with labelled `latitude` and `longitude` fields and transmitted it to their server.

While this app may not be intending to circumvent the permission system, this technique can be exploited by a malicious actor to gain access to the user's location. Whenever a new picture is taken by the user with geolocation enabled, any app with read access to the photo library (i.e., READ_EXTERNAL_STORAGE) can learn the user's precise location when said picture was taken. Furthermore, it also allows obtaining historical geolocation fixes with timestamps from the user, which could later be used to infer sensitive information about that user.

4.4 Limitations and Future Work

During the course of performing this research, we made certain design decisions that may impact the comprehensiveness and generalizability of this work. That is, all of the findings in this thesis represent lower bounds on the number of covert and side channels that may exist in the wild.

Our study considers a subset of the permissions labeled by Google as *dangerous*: those that control access to user identifiers and geolocation information. According to Android's documentation, this is indeed the most concerning and privacy intrusive set of permissions. However, there may be other permissions that, while not labeled as *dangerous*, can still give access to sensitive user data. One example is the BLUETOOTH permission; it allows apps to discover nearby Bluetooth-enabled devices, which may be useful for consumer profiling, as well as physical and cross-device tracking. Additionally, we did not examine all of the dangerous permissions, specifically data guarded by content providers, such as address book contacts and SMS messages.

Our methods rely on observations of network transmissions that suggest the existence of such channels, rather than searching for them directly through static analysis. Because many apps and third-party libraries use obfuscation techniques to disguise their transmissions, there may be transmissions that our instrumentation does not flag as containing permission-protected information. Additionally, there may be channels that are exploited, but during our testing the apps did not transmit the accessed personal data. Furthermore, apps could be exposing channels, but never abuse them during our tests. Even though we would not report such behavior, this is still an unexpected breach of Android's security model.

Many popular apps also use certificate pinning [232, 95], which results in them rejecting the custom certificate used by our man-in-the-middle proxy; our system then allows apps to continue without interference. Certificate pinning is reasonable behaviour from a security standpoint; it is possible, however, that it is being used to thwart attempts to analyse and study the network traffic of a user's mobile phone.

Our dynamic analysis uses the Android Exerciser Monkey as a UI fuzzer to generate random UI events to interact with the apps. While in our prior work we found that the Monkey explored similar code branches as a human for 60% of the apps tested [238], it is likely that it still fails to explore some code branches that may exploit covert and side channels. For example, the Monkey fails to interact with apps that require users to interact with login screens or, more generally, require specific inputs to proceed. Such apps are consequently not as comprehensively tested as apps amenable to automated exploration. Future work should compare our approaches to more sophisticated tools for automated exploration, such as Moran et al.'s Crashscope [201], which generates inputs to an app designed to trigger crash events.

Ultimately, these limitations only result in the possibility that there are side and covert channels that we have not yet discovered (i.e., false negatives). It has no impact on the validity of the channels that we did uncover (i.e., there are no false positives) and improvements on our methodology can only result in the discovery of more of these channels.

Moving forward, there has to be a collective effort coming from all stakeholders to prevent apps from circumventing the permissions system. Google, to their credit, have announced that they are addressing many of the issues that we reported to them [135]. However, these fixes will only be available to users able to upgrade to Android Q—those with the means to own a newer smartphone. This, of course, positions privacy as a luxury good, which is in conflict with Google's public pronouncements [222]. Instead, they should treat privacy vulnerabilities with the same seriousness that they treat security vulnerabilities and issue hotfixes to all supported Android versions.

Regulators and platform providers need better tools to monitor app behaviour and hold app developers accountable by ensuring apps comply with applicable laws, namely by protecting users' privacy and respecting their data collection choices. Society should support more mechanisms, technical and other, that empower users' informed decision-making with greater transparency into what apps are doing on their devices. To this end, we have made the list of all apps that exploit or contain code to exploit the side and covert channels we discovered available online [38].

4.5 Discussion

4.5.1 Privacy Expectations

In the U.S., privacy practices are governed by the "notice and consent" framework: companies can give *notice* to consumers about their privacy practices (often in the form of a privacy policy), and consumers can *consent* to those practices by using the company's services. While website privacy policies are canonical examples of this framework in action, the permissions system in Android (or in any other platform) is another example of the notice and consent framework, because it fulfills two purposes: (i) providing transparency into the sensitive resources to which apps request access (notice), and (ii) requiring explicit user consent before an app can access, collect, and share sensitive resources and data (consent). That apps can and do circumvent the notice and consent framework is further evidence of the framework's failure. In practical terms, though, these app behaviours may directly lead to privacy violations because they are likely to defy consumers' expectations.

Nissenbaum's "Privacy as Contextual Integrity" framework defines privacy violations as data flows that defy contextual information norms [209]. In Nissenbaum's framework, data flows are modeled by senders, recipients, data subjects, data types, and transmission principles in specific contexts (e.g., providing app functionality, advertising, etc.). By circumventing the permissions system, apps are able to exfiltrate data to their own servers and even third parties in ways that are likely to defy users' expectations (and societal norms), particularly if it occurs after having just denied an app's explicit permission request. That is, regardless of context, were a user to explicitly be asked about granting an app access to personal information and then explicitly declining, it would be reasonable to expect that the data then would not be accessible to the app. Thus, the behaviours that we document in this thesis constitute clear privacy violations. From a legal and policy perspective, these practices are likely to be considered deceptive or otherwise unlawful.

Both a recent CNIL decision (France's data protection authority), with respect to GDPR's notice and consent requirements, and various FTC cases, with respect to unfair and deceptive practices under U.S. federal law—both described in the next section—emphasize the notice function of the Android permissions system from a consumer expectations perspective. Moreover, these issues are also at the heart of a recent complaint brought by the Los Angeles County Attorney (LACA) under the California State Unfair Competition Law. The LACA complaint was brought against a popular mobile weather app on related grounds. The case further focuses on the permissions system's notice function, while noting that, "users have no reason to seek [geolocation data collection] information by combing through the app's lengthy [privacy policy], buried within which are opaque discussions of [the developer's] potential transmission of geolocation data to third parties and use for additional commercial purposes. Indeed, on information and belief, the vast majority of users do not read those sections at all" [264].

4.5.2 Legal and Policy Issues

The practices that we highlight in this thesis also highlight several legal and policy issues. In the United States, for example, they may run afoul of the FTC's prohibitions against deceptive practices and/or state laws governing unfair business practices. In the European Union, they may constitute violations of the General Data Protection Regulation (GDPR).

The Federal Trade Commission (FTC), which is charged with protecting consumer interests, has brought a number of cases under Section 5 of the Federal Trade Commission (FTC) Act [279] in this context. The underlying complaints have stated that circumvention of Android permissions and collection of information absent users' consent or in a manner that is misleading is an unfair and deceptive act [278]. One case suggested that apps requesting permissions beyond what users expect or what are needed to operate the service were found to be "unreasonable" under the FTC Act. In another case, the FTC pursued a complaint under Section 5 alleging that a mobile device manufacturer, HTC, allowed developers to collect information without obtaining users' permission via the Android permission system, and failed to protect users from potential third-party exploitation of a related security flaw [276]. Finally, the FTC has pursued cases involving consumer misrepresentations with respect to opt-out mechanisms from tailored advertising in mobile apps more generally [277].

Also in the United States, state-level Unfair and Deceptive Acts and Practices (UDAP) statutes may also apply. These typically reflect and complement the corresponding federal law. Finally, with growing regulatory and public attention to issues pertaining to data privacy and security, data collection that undermines users' expectations and their informed consent may also be in violation of various general privacy regulations, such as the Children's Online Privacy Protection Act (COPPA) [101], the recent California Privacy Protection Act (CCPA), and potentially data breach notification laws that focus on unauthorized collection, depending on the type of personal information collected.

In Europe, these practices may be in violation of GDPR. In a recent landmark ruling, the French data regulator, CNIL, levied a 50 million Euro fine for a breach of GDPR's transparency requirements, underscoring informed consent requirements concerning data collection for personalized ads [180]. This ruling also suggests that—in the context of GDPR's consent and transparency provisions—

permission requests serve a key function of both informing users of data collection practices and as a mechanism for providing informed consent [276].

Our analysis brings to light novel permission circumvention methods in actual use by otherwise legitimate Android apps. These circumventions enable the collection of information either without asking for consent or after the user has explicitly refused to provide consent, likely undermining users' expectations and potentially violating key privacy and data protection requirements on a state, federal, and even global level. By uncovering these practices and making our data public,[5] we hope to provide sufficient data and tools for regulators to bring enforcement actions, industry to identify and fix problems before releasing apps, and allow consumers to make informed decisions about the apps that they use.

[5] https://search.appcensus.io/

Chapter 5 LibSeeker: An Automatic, Agnostic and Hybrid approach for SDK Detection

As discussed in § 2.3, the ability to detect third-party SDKs in mobile apps and characterize their behavior is vital for the privacy analysis of Android apps. However, currently available tools might suffer from issues due to the way in which they operate. Most of these tools rely solely on static analysis and pre-defined SDK fingerprints, which results in a number of limitations. Since static analysis relies on inspecting the app's code and resources, detection tools can struggle to detect SDKs when the code is obfuscated or loaded dynamically at runtime. Furthermore, current state-of-the-art tools often rely on building a fingerprint of available SDKs code features to be able to detect SDK presence by detecting these fingerprints on the wild. This methodology has its own inherent limitations, as the lack of a correct and updated fingerprint hinders the ability of the tool to detect a given SDK. This is extremely important in an ever-changing ecosystem in which SDKs are constantly updated, new SDKs are released and others are merged together due to company acquisitions.

In this chapter, we propose a novel SDK detection method based in our Hypothesis ($\mathcal{H}$ 1). To do so, we first investigate the shortcomings of state-of-the-art tools, namely LibRadar, LibScout and Exodus (§ 5.1). We run an evaluation of their ability to detect SDKs on a set of ground-truth apps(§ 5.2). Using this knowledge, we present LibSeeker, a new SDK detection method that relies on a combination of signals extracted both statically and dynamically from a given app to confidently detect the presence of third-party code without the need of any a-priori knowledge (§ 5.3). We compare this solution against the three aforementioned static analysis tools, further showing their shortcomings when detecting SDK presence both on a dataset of 5k real world Android apps (§ 5.5). Finally, we demonstrate the utility of LibSeeker with two case studies: an analysis of SDK prevalence and permission piggybacking by SDKs using a dataset of 5k publicly available apps from the Google Play Store (§ 5.6. We conclude the chapter with a discussion (§ 5.8) of different ways in which we could improve the privacy and security risks introduced by third-party SDKs.

5.1 State-of-the art SDK Analysis Tools

To build an accurate methodology that advances the state-of-the-art SDK detection, we first need to make a comprehensive study of currently available tools. SDK detection is a widely studied topic, with a vast amount of proposed methodologies and tools [301]. These methodologies have evolved with time, as the first tools relied mainly in analyzing the class tree in search for third-party libraries, which made them highly susceptible to code obfuscation techniques [146, 252, 263, 74]. While most solutions rely on static analysis techniques, they include their own ways to cope with the shortcomings inherent to these techniques, more specifically the difficulty to analyze obfuscated code. LibId is a SDK detection tool that attempts to analyze obfuscated apps by relying on code features that stay consistent under code shrinking, control flow randomization, and package modification [303]. Namely, the authors create library fingerprints from *basic block signatures* which include all field related operations, methods calls and string presence. LibPecker relies on class dependencies, assuming that even if class names might be changed during obfuscation, their dependencies will remain the same [305]. Then, the authors create a profile for each library and for every third-party package found in an app and compare it against such profiles building a similarity score. The authors claim high precision and recall even if the app has been obfuscated with basic techniques such as ProGuard.

However, relying on a pre-computed set of library fingerprints to detect SDKs in apps has its own drawbacks and limitations. To overcome this issue, LibExtractor builds a dependency graph to detect potential third-party libraries, and then clusters those libraries that have the same code-base in an attempt to find one un-obfuscated instance [301]. However, their ability to detect candidate libraries can be hindered by the presence of advanced obfuscation techniques, such as control flow randomization and class renaming. In a recent study, Zhan et al. compared 5 state-of-the-art detection tools and showed that none of them presented an accuracy higher than 65% when detecting SDKs in apps [300]. They analyze the main cause for false positives and show that due to SDKs relying on pre-computed fingerprints, whenever an SDK partly includes code from another library, these tools would report both SDKs as they can find code that matches the profile of both SDKs. In terms of false negatives, the authors show that the main reasons are the use of code-obfuscation techniques and shortcomings inherent to the detection algorithm such as the lack of a given SDK profile for a tool.

Other researchers have opted for dynamic analysis techniques, inspecting the traffic generated by the app during runtime to map the hostnames contacted by the app to SDK companies. Razaghpanah et al. built Lumen [231], a traffic monitoring tool that relies on Android's VPN to be able to analyze app's traffic even when encrypted, to analyze the Android tracking ecosystem [230]. The authors developed a novel methodology to detect third-party and tracking at the traffic level, based on automatically obtaining information from online resources for domains that collect personal information to confirm that they belong to a company offering these services. This technique allows them to discover over 2k domains related to advertisement and tracking. Other authors have also relied on the use of Android's VPN to perform a person-in-the-middle "attack" to be able to analyze a device's network traffic [256, 235]. However, security changes in Android 11 [269] rendered this technique in-effective, as user certificates are no longer trusted by the OS, which disallows the analysis of encrypted traffic through the VPN service. Since these changes, authors have moved to new approaches to analyze the network communications of apps, mainly relying on dynamically hooking to the functions in charge of sending network requests to be able to analyze their content [78, 246, 86].

While these techniques can be very effective for the privacy analysis of Android apps, they are not best suited for the detection of SDK presence. The fact that a given piece of personal data was sent to a hostname does not necessarily mean that the SDK related to such hostname is included on the app. The advertisement and tracking ecosystem on Android is very dynamic, and a number of different actors might be contacted before an ad is shown to the user [55]. A clear limitation of traffic analysis for SDK detection are ad-network aggregators. Many mobile SDKs allow the developer to include different ad-networks within a single SDK to contact them at runtime in search for the most appropriate advertisement. This will lead to a single SDK contacting several SDK-related hostnames during runtime, leading to false positives.

In this thesis exhaustively analyze three static analysis tools: LibRadar [188], LibScout [48], and

Exodus [91]. We choose these three tools for several reasons, the most important one being that all of them are widely used by the research community. Both LibRadar and LibScout come from research efforts, and are very similar in their methodology. Both these tools aim to detect SDK presence, relying on creating fingerprints from a number of static analysis signals available in apps. On the other hand, Exodus is a tool developed by a NGO and aims to make privacy analysis available for all type of users. This tool also relies on static analysis, which means that it shares many of the limitations of the other two tools. In the rest of this section, we will take a deeper look at each of these tools individually.

LibRadar. LibRadar is a popular tool developed by Ma et al. to detect third-party SDK presence in Android apps that is widely used in the research literature [116, 309, 217, 219, 97, 118, 308, 286, 215, 255, 99, 149, 60]. LibRadar's methodology relies on two different parts. First, they have a feature extraction methodology that analyzes the different classes present in the app's smali code and builds a profile based on the frequency of Android API calls. These are instances in which a class relies on a function provided by Android itself, which often times remain un-obfuscated, making them more stable for library detection. Then, they run this extractor on 1M apps and apply a clustering mechanism to detect those classes that appear on more than 50 different apps, assuming that they are third-party libraries. Therefore, Libradar's accuracy is fully dependent on its ability to keep these profiles updated, as changes in these SDKs will tamper its ability to detect them. LibRadar includes a manual mapping of the top 200 libraries detected that includes the name of the SDK, the company behind this service and other relevant information such as the category of this library. Such categories include Advertisement, App Market, Development Aid, Development Framework, Digital Identity, Game Engine, GUI Component, Map/LBS, Mobile Analytics, Payment, Social Network and Utility. The main issue that comes from identifying an specific SDK is the need to keep updated static mappings, as the ecosystem of Android SDKs is highly volatile with new SDKs being released and popularized and companies acquiring other SDK providers or being merged. The issue with maintaining a classification of libraries are similar, as they need to be constantly updated to reflect changes in the behavior of a given SDK. Furthermore, it is common for SDKs to offer different functionalities or sub-products, meaning that these categories might not accurately reflect all of the capabilities of a given SDK [112].

LibScout. Developed by Backes et al., LibScout works similar to Libradar, as it also has a two-step process that builds fingerprints from specific library versions [48] and then looks for those fingerprints in real-world apps. LibScout extracts information about the libraries' code and builds a signature that is then used to match against the code of the analyzed app in an attempt to find an exact or partial match.To build such signatures, LibScout relies on what the authors call *fuzzy descriptor* which are method signatures in which any information that can be obfuscated (e.g., the name of the class) is replaced by placeholder. To build such profiles, LibScout must have access to the SDKs code, which is not always possible as many commercial SDKs are offered as compiled .jar files to protect their intellectual property. In their particular case, they gather the name of popular libraries and look in publicly available repositories (i.e., maven central, github or the SDK's webpage) for the libraries code. LibScout was built for security applications, as it has an unique feature which is the ability to detect different versions of the same SDK by creating one profile per version. Much like in the

Table 5.1: Summary of the methodology and limitations of the three tools used for comparison. $\star$: Fingerprint Coverage; ‡: Dead and Unused Code; †: Resilience Against Advanced obfuscation techniques.

Product	Method	Code-features	$\star$	‡	†	Papers using the tool
LibRadar	Static	Frequency of Android API calls	✓	✓		[116, 309, 217, 219, 97, 118, 308, 286, 215, 255, 99, 149, 60]
LibScout	Static	Method signatures	✓	✓		[215, 9, 205, 207, 12, 82, 253, 225, 208, 204]
Exodus	Static	Java classes in DEX files	✓	✓	✓	[9, 208]

previous case, LibScout's ability to detect SDKs is highly dependent on the profiles available and changes to SDKs can reduce its accuracy. Similarly, the tool can only detect those version for which they have a profile in their database. LibScout has been widely used in academic papers since it was first released [215, 9, 205, 207, 12, 82, 253, 225, 208, 204]

Exodus. Exodus is a french non-profit organization which strides to generate awareness about tracking on the Android apps ecosystem. While a complete description of Exodus' methodology is not available on a scientific paper, the project website explains partly its methodology [91]. Exodus builds fingerprints of trackers based both on code paths (e.g., "com.google.firebase") and hostnames (e.g., "firebase.google.com"). However, the exact way in which these fingerprints are generated is not documented. Nevertheless, as the code is open-source, we can analyze their SDK detection mechanism when applied to mobile apps. For each app, it extracts the Java classes embedded in each DEX file and compares them against their list of known tracker signatures using string matching. The main advantage of Exodus in contrast with most academic solutions is that it is regularly updated (i.e., the maintainers actively include signatures of new trackers when they are found). However, Exodus is a privacy product for privacy analysis, and as such, it only includes SDKs that they can classify as "trackers". [1] This means that many Android SDKs that fall in other categories (such as development support) can not be detected by Exodus.

5.2 Evaluation of State-of-the Art Tools

This analysis of Libradar [187], LibScout [48] and Exodus [91] allows us to find a number of shortcomings related to their methodology. In this section, we rely on a set of custom made apps to study how these shortcomings inherent to their methodology translate in the inability to detect SDKs in real apps. Namely, we design our test apps with the focus on measuring the following limitations of static analysis methods: [2]

Fingerprint Coverage ($\star$). We define fingerprint coverage as the ability to detect a given SDK because the tool has a valid fingerprint for it. All of these detection tools rely on this paradigm, in which there is a need for a pre-computed list of SDKs fingerprints to be able to detect their presence in apps. However, it is impossible to have complete coverage of a such vast ecosystem due to the large number of different third-party libraries available. While the authors of these tools made an attempt to be as comprehensive as possible and to have a wide number of libraries (i.e., 402 libraries

[1] Exodus' definition of tracker is "a piece of software whose task is to gather information on the person using the application, on how they use it, or on the smartphone being used" [90]

[2] Table 5.1 shows a summary of these limitations and their presence on the analyzed tools

for LibScout and 428 trackers for Exodus [226]) they are likely to miss unpopular libraries found on the long tail for which a fingerprint was never created. Furthermore, Exodus focuses specifically in tracking and advertising libraries so it is unpractical for non-privacy related analysis.

Furthermore, the SDK ecosystem is constantly changing. SDKs are constantly being updated, which might result on changes to its fingerprint, and its is also common to see new products being released. Therefore, these tools' ability to detect SDKs is highly co-related to their ability to maintain their profiles updated. However, LibScout profiles were last updated in July 2019, and while the authors provide scripts to generate new profiles, users need to be able to access the SDK's jar [3] and to manually generate a XML with metadata about the name and version of the SDK. LibRadar's profiles were last updated in 2018 and the authors do not provide a way to create new profiles. Therefore, these solutions are bound to miss newer SDKs and updated versions of SDKs as the original profile is likely to be outdated. Exodus is more recently updated and its open-source nature allows contributors to add new fingerprints for tracking services as they are discovered.

Dead and Unused Code (‡). This limitation refers to the inability of a given tool to detect whether a given SDK is actually being used by the app or not. It is well-known that most Android apps include code that is never used by the app itself, often times as part of legacy code that is no longer needed or as a consequence of copying code snippets from web-forums [106]. Since SDKs often include different types of functionalities, many apps only use a very small part of their included SDKs [167]. All of these tools treat the app as a whole, and make no distinction on whether the code that is matched to a given library profile is ever called from within the main app's code. While the SDK's code is indeed present in the app, this distinction is extremely important for most applications, such as security and privacy analysis of apps. Android's ProGuard includes a resource shrinker to reduce the number of unused code in the final release of an app [22], however upon manual verification we confirm that SDK code is present in the final release of an app even if the SDK is never used by the app.

Resilience Against Advanced Obfuscation (†). Previous work has shown that static analysis methods are prone to miss app behaviors in the presence of obfuscation techniques and dynamic code loading [303]. To avoid these issues, LibRadar and LibScout build their fingerprints using code features that remain unchanged when using the most common obfuscation methods. However, more advanced methods [33] that modify the app's call graph or that dynamically load code could temper with their ability to properly analyze a given app. In fact, Zhang et al. presented an analysis in which they showed that LibScout failed to detect most SDKs on apps which had been obfuscated [303]. On the other hand, Exodus relies on java classnames for its signatures. Any obfuscator that renames classes could hinder its ability to detect SDKs present in a given app.

5.2.1 Ground-truth Apps

After analyzing the limitations inherent to static analysis detection tools, we aim to measure how they can affect the ability of LibRadar, LibScout and Exodus to detect SDKs in real world applications. To that end, we need to build our own set of apps, as we cannot rely on publicly available since only the developer of the app itself knows which are the SDKs embedded on the app and which code

[3] LibRadar needs to access the SDKs code to be able to generate new fingerprints

obfuscation method is used. One can argue that, to overcome this limitation, we could extract the list of SDKs present in a given app by relying on the information disclosed on its privacy policy. However, this will only be valid for the subset of libraries collecting personal data and previous work has shown that privacy policies are often incomplete and inaccurate [212, 284, 208]. Therefore, we build a set of apps with the following properties and behaviors to be able to measure the shortcomings of these SDK detection tools in a fully controlled experiment:

Integrated SDKs. We build one app that includes, initializes and invokes nine popular and well-documented SDKs: `Firebase` [108], `Facebook` [92], `Adjust` [4], `IronSource` [161], `UnityAds` [275], `AdColony` [3], `Phunware` [221], `AppLovin` [43] and `Braze` [66]. We note that we choose popular SDKs to avoid manufacturing a test in which all SDK detection tools fail to find obscure or unpopular libraries. Therefore, the results showed in this experiment present only a lower-bound on potential false negatives by these tools. For each of these SDKs, we followed the official documentation to include the latest available version as of July 2022 in the app. Similarly, the code of the app is not obfuscated in order to get a baseline on the ability of each tool to correctly detect SDKs. When possible, we include different features from the SDK such as the declaration of custom permissions or services on the manifest file of the ground-truth app.

LibRadar fails to find all of the libraries that we have embedded into the un-obfuscated app as it only reports the tracking SDK `Supersonic Ads`, which was bought by `ironSource` in 2015. We take a deeper look at LibRadar's behavior when analyzing these apps and find that, regardless of using popular SDKs, LibRadar's fingerprints are so outdated that none of them match the SDKs present in the app. LibScout is only able to correctly detect the presence of `Firebase` and `Facebook`. This also shows that LibScout lacks profiles for newer versions of popular SDKs and thus it cannot detect their presence. In fact, LibScout's profiles are available to the user and we can see that only these two SDKs contain a profile on LibScout's database, explaining why they are the only ones reported. Finally, Exodus is able to report the presence of all the SDKs that are present in the test app except for Phunware. However, the report includes 15 total trackers, showing that Exodus is over reporting the presence of libraries. Some of this is explained by the fact that it reports different sub-products for Facebook (Ads, Analytics, Login and Share) but it also reports SDKs that are not at all present in the app. This the case of `Amazon Advertisement`, `IAB Open Measurement` and `Mintegral`. Upon manual investigation, we find that other SDKs are using parts of the code from these libraries, which lead to the presence of Java packages that match Exodus' tracker signatures.

Robustness Against Fake Signals. String-matching techniques can render false positives due to unused code or even comments in the app's code. To assess the robustness of our target tools against this behavior, we build an app that does not include any real SDK but instead contains strings that may trigger false positives. Namely, we include strings that are related to SDKs (e.g., "com.adjust"), a module of the app itself named "com.google.firebase" including a GET request to `firebase.com` and a fake custom permission that points to "IronSource" on the manifest. Finally, the app imports the Facebook SDK but never uses it, thus simulating dead code, which is common amongst Android apps [106]. This app will allow us to measure the amount of false positives reported by each solution, i.e., over-reporting of SDK presence).

We find that LibRadar does not report any of the SDKs for which we have added "fake" signals.

Nevertheless, seeing the inability of LibRadar to report the presence of SDKs in our first ground truth, we cannot know for sure whether this is an artifact of this inability or the methodology being sound against fake signals. On the other hand, LibScout reports the presence of several `Facebook` SDKs. While the code of the Facebook SDK is included on the app, this is never invoked from the main app's code. Since LibScout relies on a single signal from the app's code, it lacks the ability to detect if an SDK is actively used by the app, or if its presence is an artifact of dead or legacy code. Exodus also presents this issue, as it reports several `Facebook` products (namely Analytics, Login and Share) even though the main app's code does not use any of these features.

Robustness Against Advanced Obfuscation. We use the same app that we build for the SDK detection case but rely on Obfuscapk [33] to apply different obfuscation techniques to the app. One could argue that simple obfuscation such as the default ProGuard [57] are more commonly used by apps. However, the obfuscation techniques used by this tool are easy to bypass by most of currently used techniques. In fact, when using ProGuard to obfuscate the app, [4] we find no difference with the results provided for the un-obfuscated app by any of the tools. Therefore, we compile the app using the following techniques available within Obfuscapk: class renaming, field renaming and method renaming. Interestingly, LibRadar is still able to detect `SuperSonic` Ads but its overall inability to detect SDKs in our ground truth app prevents us from accurately measuring its ability to bypass obfuscation. LibScout is still able to detect `Facebook` but it is no longer capable of detecting `Firebase`. Finally, we find that Exodus' ability to detect SDKs is completely defeated by these obfuscation techniques, as it is no longer able to detect any SDK.

Takeaways Table 5.2 shows a summary of the accuracy of the three studied tools for each of the ground truth apps. As shown in our analysis, state-of-the-art tools leveraging static analysis are prone to miss SDKs because they need to rely on a-priori knowledge about and SDK to be able to detect it. Furthermore, they are prone to over report the presence of SDKs on the presence of dead and unused code as they make no distinction on whether a given SDK is actually used in the main app's code. Finally, we have shown that it is important to develop detection methods that are resilient against advanced obfuscation techniques, to avoid missing SDKs. This evaluation of currently available tools inform us to develop a novel methodology to is able to more effectively detect SDK presence, being more resilient against advanced obfuscation techniques and not relying on pre-calculated fingerprints of SDKs to be able to detect them.

5.3 LibSeeker's Methodology

Our analysis of the limitations of state of the art tools allows us to detect the different properties that an ideal third-party code detector should have the following properties:

- **SDK Agnostic:** We aim to build a solution that does not rely on a-priori knowledge about different libraries to be able to accurately detect them. To that end, we propose a new paradigm in which we rely on signals extracted from static and dynamic analysis to detect potential third-party libraries on an app. Then, this information should be interpreted via an analyst to decide

[4] We use the different techniques available to developers, include code shrinking, resource shrinking and any configuration specified in the documentation of the included SDKs

Table 5.2: Summary of the accuracy of the three tools against out set of ground-truth apps. TP means that the tool accurately reported the SDK. FN means that the tool was unable to report the SDK. TN means that the tool did not report the SDK despite the presence of fake signals. FP means that the tool reported the presence of an SDK because of fake signals. "-" means that the SDK was not present and that we did not include any fake signal for it.

SDK	All SDKs			Fake Signals			Obfuscated		
	LRadar	**LScout**	**Ex.**	**LRadar**	**LScout**	**Ex.**	**LRadar**	**LScout**	**Ex.**
Firebase	FN	TP	TP	TN	TN	TN	FN	TP	FN
Facebook	FN	TP	TP	TN	FP	FP	FN	FN	FN
Adjust	FN	FN	TP	TN	TN	TN	FN	FN	FN
IronSource	TP	FN	TP	-	-	-	TP	FN	FN
UnityAds	FN	FN	TP	-	-	-	FN	FN	FN
AdColony	FN	FN	TP	-	-	-	FN	FN	FN
Phunware	FN	FN	FN	-	-	-	FN	FN	FN
AppLovin	FN	FN	TP	-	-	-	FN	FN	FN
Braze	FN	FN	TP	-	-	-	FN	FN	FN

which of these are indeed a third-party SDK. This might seem counterproductive, however we argue that it is the only way to provide a tool that is SDK agnostic and that can be useful through the changes of a highly volatile ecosystem.

- **Robust Against False Positives:** As we discussed previously, many static analysis techniques can over-report SDK presence due to legacy and dead code. To that end, we rely on a combination of different static and dynamic analysis signals, together with a confidence metric that allow us to discard cases of over-reporting.
- **Obfuscation Resilient:** Our selection of different signals includes features that are obfuscation-resilient, thus helping us overcome basic obfuscation techniques. Only for those cases which rely on more advanced techniques, we employ a fingerprint based approach to de-obfuscate libraries that we have previously seen in other analysis. To limit the impact on our ability to provide an SDK agnostic tool, every time LibSeeker analyzes an app, it saves the profile of each un-obfuscated SDK to keep up with changes on the ecosystem.

To this end, we present LibSeeker a methodology that combines static and dynamic analysis in order to build a prototype SDK detector that satisfies all of the aforementioned properties.

5.3.1 Signal Extractor

As we discussed in § 2.4, static and dynamic analysis each provide their own advantages and limitations. In line with our Hypothesis ($\mathcal{H}$ 1), our methodology to detect third-party code relies on automatically extracting and reasoning about signals extracted using static and dynamic analysis that may be associated with a third-party SDK. When used in combination, static and dynamic analysis complement each other, increasing the confidence on the presence of a given SDK. For example, the presence of a classname in the app's code might lead to false positives as it this may never be used during runtime. However, dynamic analysis can provide further evidence of the use of this class if it is loaded at runtime.

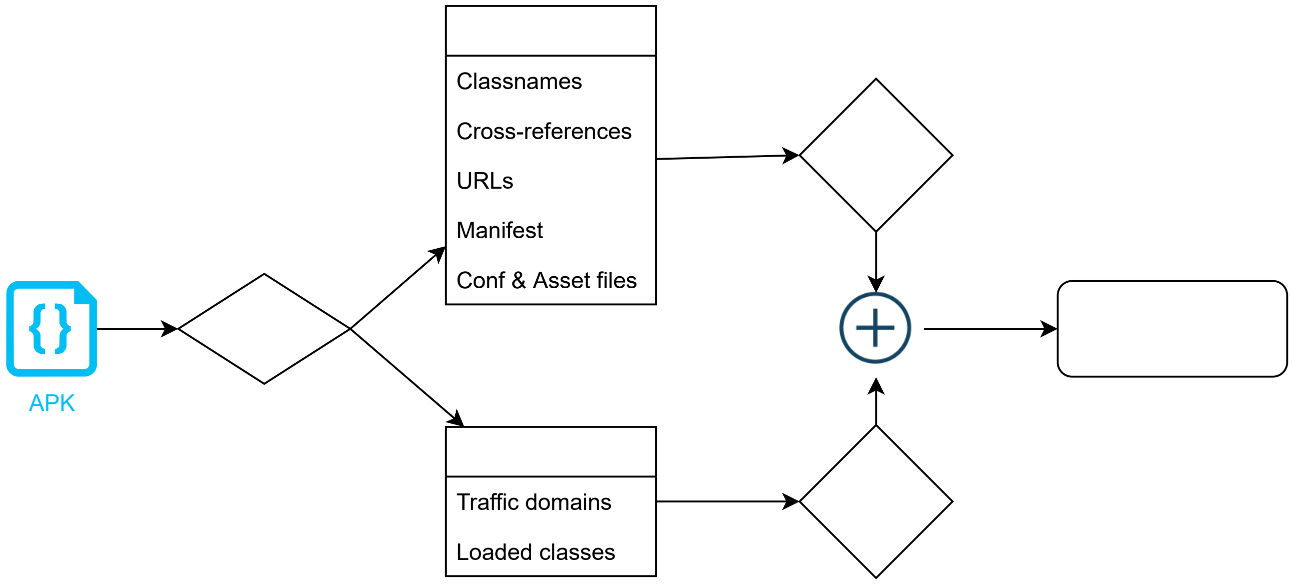


Figure 12: Overview figure of the pipeline for SDK detection

Static analysis

Static analysis allows us to collect signals that can be extracted from app's components without the need to actually run the app. We curate a set of signals that are often present when an SDK is embedded in an app, we generate this list by manually reading the documentation of a number of third-party SDKs. We note that, while extensive, this list is not exhaustive as there might be other signals that can be extracted statically from an app and that are not a part of our solution. However, this list of static analysis resources goes further than those used by previous work such as Exodus (§ 5.2). Namely, we extract the following signals, with some examples shown in Table 5.3:

- **Classnames.** The most obvious indication that a given SDK is embedded on an app is the fact that there are classes that belong to its namespace. Therefore, we rely on APKTool to extract all of the full paths for the classes that are included on the app. We discard those that share the namespace with the application as they would not belong to a third party. However, relying purely on the presence of a Java class for detection can lead to the over reporting of SDKs because of unused code. To deal with this, we also use Androguard to check whether there is a cross-reference to the class on the app's main code. A cross-reference means that some functionality from this external class is actually being used by the app itself.
- **URLs.** Many SDKs rely on online services to provide their service (e.g., downloading ads or uploading telemetry). To that end, we look for URLs hardcoded in the app's code. We first use Androguard to extract all the strings present in the code and then use regular expressions to identify those that are **urls**. As we have seen in our state-of-the-art analysis (§ 5.2), the presence of a string alone can lead to false positives, for example we might detect urls present in comments. To that end, we only keep those urls that can be mapped to one of the classes previously found.
- **Manifest info.** SDKs can often rely on different elements other than the code itself to present their services to the app, which need to often be declared in the app's manifest file. Therefore, we use a custom built manifest parser that extracts several signals from this file. We extract the **custom permissions** (§ 2.2) present in the manifest, and try to match them against any of the

classnames that we discovered previously. Some SDK providers declare their own custom permissions to provide extra functionality to the app's developer (e.g., `com.baidu.permission.*`). Similarly, SDKs can define their own **services**, **providers** and **receivers** to provide extra functionality for the app (e.g., `com.google.firebase.*`).

- **Configuration files.** Often times SDKs need extra assets and files, such as media or string files for their correct functioning. These are saved in specific folders that can be accessed when de-compiling the app. Therefore, we scan the whole directory structure of the APK and match the configuration files and folders against the potential SDKs for which we found classnames in the decompiled app. When available, this signal is extremely useful as it is often available even if the app is heavily obfuscated.

Dynamic analysis

In line with this thesis, we complement the static analysis signals with others extracted using dynamic analysis of the app's runtime behavior. It is important to note that, while dynamic analysis might miss behaviors due to the difficulty to fully exercise the app, whatever observation holds more weight as it is actual evidence of the app's behavior. For dynamic analysis, we got access to AppCensus [37], a commercial solution which runs Android apps in a highly instrumented version of Android, using random inputs to exercise the app and test out as many of the code paths as possible. AppCensus provides a number of useful dynamic analysis features, such as monitoring access to protected resources or the ability to observe network transmissions even if obfuscated [238, 233]. To do so, the instrumented device monitors both access to the filesystem as well as network traffic. Since the VPN interface is no longer a reliable source for network monitoring [269], AppCensus relies on Frida to inspect traffic by instrumenting the functions of the network interface. All apps were tested on un-rooted Pixel 3As phones, using a custom kernel and Android 10 image (i.e., avoiding anti-testing and anti-emulation techniques). The relevant pieces of information that we collect as signals for our detection method are:

- **Network communications.** As discussed previously, many SDKs rely on network transmissions to provide their functionality, for example advertisement SDKs need to fetch relevant ads from ad-networks [55]. Therefore, we rely on AppCensus' ability to monitor the traffic communications of apps, even when encrypted. It is important to note that the fact that a given domain is contacted is not enough by itself to detect the presence of an SDK. There are different types of SDKs (e.g., ad provider aggregators) that contact different ad networks even when the SDK is not included on the app. This goes to show the importance of putting static and dynamic analysis results in context together to get a better understanding on the third parties included in an app and avoid over-reporting. While the domain that is contacted by the app is highly relevant, there are other pieces of information that can be helpful to detect embedded SDKs. Namely, we also collect the User-Agent of each network flow. While most Android communications use the default User-Agent string, this can be modified by each library to show that a given connection is coming from their own SDK. On the other hand, there is not any rule on what the value of this string must be, and thus any party could spoof it and assign a value that falsely leads to a given SDK. Therefore, while not enough on its own, this value can help increase the confidence that a given SDK is present on an app.

Table 5.3: Examples of different signals found in real-world apps and the SDK to which they matched

Signal name	Signal value	SDK match
Classname	com.adcolony.sdk	com/adcolony
URLs	firebase.com/auth	com/firebase
Manifest info	com.adcolony.sdk.AdColonyInterstitialActivity	com/adcolony
Conf. files	/com/appsflyer/internal	com/appsflyer

- **Loaded classes.** We modified the Android Runtime (ART) Java virtual machine to record whenever a class object was loaded. We did this by instrumenting the `FindClass` method of the class linker whenever it failed to return a class from cache and loads it from storage. This tells us whenever the corresponding code is actually used at run time, and thus gives us evidence that particular classes we see statically in the app are actually used during execution. Of course, just because a class was not loaded during our testing does not mean that it is never loaded as in most cases we cannot exhaustively test every code path. We therefore rely on class loading evidence to increase our confidence on the presence of classes observed through static analysis.

5.3.2 Grouping signals

The signals presented in § 5.3.1 are disconnected, they come from different resources and might bring different levels of certainty. Therefore, we need to develop a method to group these signals together, mapping them to a specific SDK. While these signals can be matched directly to a path, e.g., a classname or a service found on the manifest, for others we need to develop specific grouping methods. To that end, we rely on a similarity metric based on the "gestalt pattern matching" algorithm [228] which does not take into account "junk", i.e., white spaces or blank lines. The result of this algorithm is a percentage ranging from 0 (completely different) to 1 (exactly the same). We run an empirical analysis to figure out what is a reasonable threshold for this ratio that reduces the number of false positives while still finding non-exact matches. To that end, we relied on 5 randomly picked publicly available apps for testing. For each pair of string and classname, we analyze the similarity score result and use this intelligence to choose an appropriate threshold, which we find to be 0.5 as it manages to catch most matches without resulting in false positives. A couple of examples are the URL `firebase.com` has a 0.66 ratio with the classname "com/firebase" and a 0.41 ratio with "com/firebird". Similarly, the services *FacebookService* and *FacebookLogin* match "com/facebook" (with a 0.52 and 0.58 value respectively) while *FirebaseService* has a value below the threshold (0.15). Therefore, for every non-trivial signal, we compare it against every classname previously found and if the matching value is above 0.5, we assign the signal to the same SDK as the classname.

5.3.3 Dealing with Advanced Obfuscation Techniques

The combination of static and dynamic analysis signals used by our methodology allows us to be resilient against the most common obfuscation techniques, like Proguard [21]. Most of these techniques lack the ability to change the Android manifest, asset files or even the runtime behavior of the app, meaning that most of the signals that we collect would still be present in the app. However, some apps are specially interested in protecting their intellectual property and rely on more robust obfuscation techniques, While previous work has shown that only about 25% of apps in Google Play are properly obfuscated [292], this number still deserves special attention from the point of view of an SDK detection tool. In fact, previous work has shown that most static analysis tools fail when presented with obfuscated apps [303].

Therefore, we build an extra tool for detecting third-party code in highly obfuscated apps. It is important to note that this is the only feature of LibSeeker that cannot work without any a-priory knowledge of SDKs code. Similar to previous work [187, 48], we take advantage of the fact that the same library might be obfuscated on some apps and unobfuscated in others. For each third-party library found on an app, we build a fingerprint based on code features that are likely to remain the same after being obfuscated. Namely, we rely on the signatures of the functions present (i.e., function name, return type and arguments) as well as the strings that appear on the libraries' code. This fingerprint allows us to create a database of fingerprints for non-obfuscated libraries. Then, whenever we encounter an obfuscated library [5] during our analysis, we can compare it against this database to find a match and de-obfuscate it. Therefore, to initially populate the database, we first run our fingerprinting solution on a subset of 1k apps gathered randomly from our dataset. Then, whenever we run LibSeeker, if there is a new fingerprint that is not on the database we store it, allowing us to keep track with changes on the ecosystem.

5.3.4 Building Confidence

Different signals might bring different certainty levels, and thus, when different SDKs are mapped to an SDK, not all of them can be treated equally. Namely, we build a confidence score, that needs to be calculated based on the type of signals that point towards the presence on an SDK. As we discussed before, previous detection tools can over-report SDKs that are not used by the app and relying on different signals that allow us to build up a confidence score allows LibSeeker to overcome this issue.

We rely on a confidence metric that ranges between 0 and 1 and increases when more signals are found. Namely, 0 means there were no signals for a given SDK and 1 means all possible signals were found. We build this score using an approach very similar to a weighted mean, in which different signals are assigned different weights. We calculate this weights based on two dimensions: 1) **signal strength**, which is an indication on how likely it is that an SDK is present in an app if the signal is found; and 2) **signal availability**, which reflects how often these signals appear on apps. It is important to take both of these characteristics into account, as assigning weights based only on signal strength can lead to false negatives, as some of the strongest signals are not commonly available on all apps (i.e., a service or custom permission related to a given SDK). In summary, we have strong signals (with a weight of 3 over the total) and weak signals (with a weight of 1 over the total). We classify most signals as weak (i.e., classname, related url, configuration file, manifest presence and user-agent string), with only a cross-reference in the main code, a network transmissions or a class loaded during runtime being considered a strong signal.

In order to calculate the confidence for each class and find which are high enough to be reported by the tool, we first need to merge all signals related to the same library together. To do so, we rely on a tree structure that starts with only the root ("/") node, and that keeps growing with every package found by one of our signal extraction mechanisms. To add new packages to the tree, we first

[5] We detect obfuscated libraries based on their name, as obfuscated libraries are renamed to non-legible strings such as "a/b/c" or "zz/xyz"

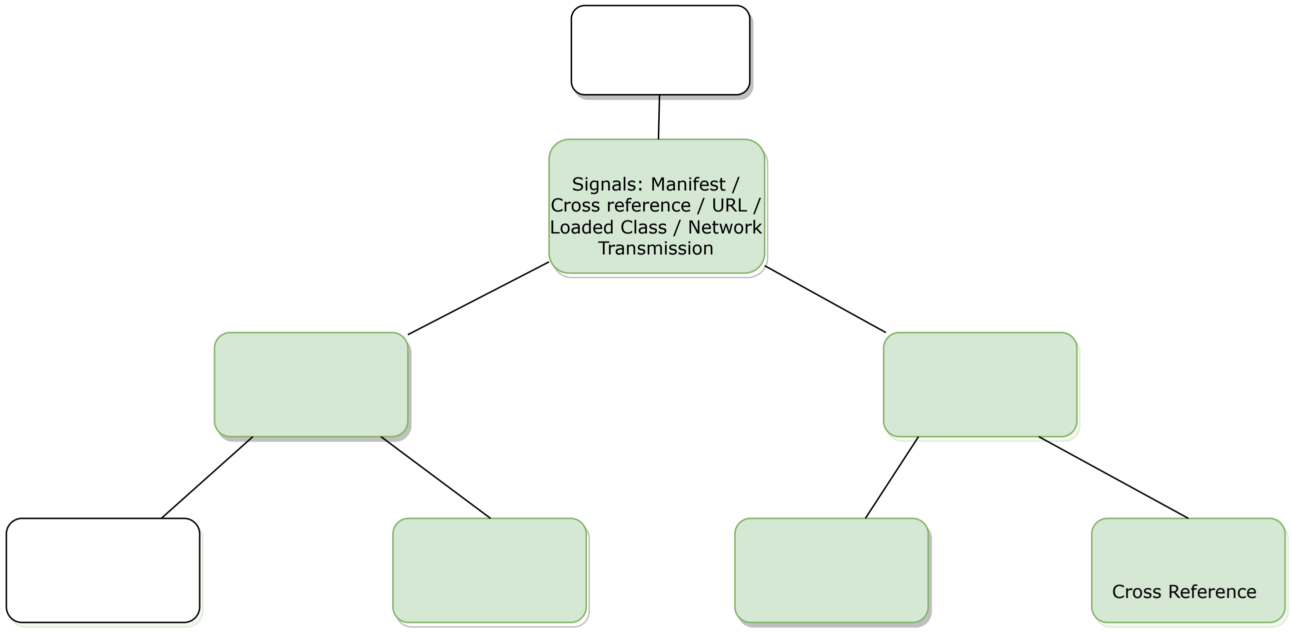

Signals: Manifest / Cross reference / URL / Loaded Class / Network Transmission
Cross Reference

Figure 13: Example of the tree structure that we use to store the different signals that increase the confidence for a particular node together with the signals found for each node. The nodes in green are above the confidence threshold.

represent the package as a tree itself, where each one of the subpackages is a leaf for the previous one (i.e., "com/google/firebase" would be represented as a tree with three nodes). Every node must be stored at a different level of the tree, for instance the "com" node would be stored on the second level, as a son of the root node "/". Therefore, for every signal, we start at the root level and traverse the tree looking for the lead that corresponds to the classname that the signal points to. Whenever we find that the current node already exists on the tree, we annotate it with the signal that has been mapped to the node. If a node with the same name does not exists yet, then we add a new node and annotate it with the current signal. We do this for every node of the package name, going down one level on the tree for each new node. At the end, we have a full representation of all packages found as a tree, in which each node contains all of the different signals that point towards the presence of a given package. Figure 13 shows an example tree including some examples nodes and the signals that were found for each one of them.

Once we have built a complete tree, we need to traverse and select those libraries for which we have enough signals to increase the confidence on its presence on the app above a given threshold. Note that we calculate the confidence value separately for the static signals (c_s) and the dynamic signals (c_d) and that the sum of both must be above this threshold. Therefore, the minimum value of the total confidence ($c_t = c_s + c_d$) is 0 and the maximum value is actually 2. Note that, because of the own limitations of dynamic analysis, and the fact that many SDKs do not contact any network domain, it is not uncommon for a SDK to lack dynamic analysis signals. This prevents us from calculating c_t as a product of c_s and c_d instead. We run an empirical analysis using different values for this threshold and found 0.5 to be the most fit for reducing the number of false positives while still being able to report SDKs for which all signals are not present.

To select the nodes to be presented as output, LibSeeker traverses the tree and for each node it looks at the c_t value of each children nodes. For any node that is above the threshold, it keeps going down the branch until it reaches a child that does not meet the confidence threshold. We note that we attempt to differentiate between different sub-products of the same library. Therefore, when we reach the last node of a branch with a value over c_t, we report any of the nodes at the same level that also meet the confidence value criteria. For example, if a tree has “com.facebook.login” and “com.facebook.ads” and both the “login” and “ads” nodes have a high enough confidence value, LibSeeker with report both of them separately. It is important to know that, in order to avoid relying on static lists such as mappings between library packages and library names, our technique outputs a list of classnames instead. This gives the ability to the user to decide whether or not to include an additional resource to go from classnames to library names. The example tree in Figure 13 shows the nodes above the confidence threshold in green. The final report for this example would show the following paths: “com.google.gms”, “com.facebook.ads” and “com.facebook.login”.

5.4 Dataset

In order to evaluate the capabilities of LibSeeker, we rely on a dataset of apps publicly available in the Play Store. To assemble this dataset, we leverage a repository collected in previous research efforts that has over 700k package names obtained from the Play Store and randomly select a subset of 5k apps for this study. Then, we use a custom-built crawler that downloads the latest available version of a given app (per package name), which we launched on June 2022. This dataset is representative of all type of apps available on the Play Store, in terms of popularity, developer type and user rating.

In terms of popularity, we see that over 57% of apps have more than 1M installs with approximately 1% of apps having over 1000M installs. On the other hand of the spectrum, we also have representation for low popularity apps, including around 3% of apps with 100 downloads or less. In terms of user rating we also observe differences across apps. While the average rating is 3.8 over 5 we observe that most apps are well rated by users with 64% of apps having a rating of 4 or above while only 9% of apps have a rating below 2. Finally, our dataset includes apps from a wide variety of developers. While most developers (69%) are responsible for one single app in our dataset, we also have several apps from well-known vendors such as Google (36 apps), Microsoft (17 apps) or Gameloft (15 apps).

5.5 Evaluation

In this section, we set out to compare LibSeeker to LibRadar, LibScout and Exodus. Due to the lack of ground truth to evaluate our new proposed methodology, we will rely on three different datasets: 1) the set of ground-truth apps discussed in § 5.2.1; 2) a dataset of 5k publicly available Android apps described in § 5.4, and 3) a subset of 25 randomly sampled apps from this dataset of public apps for which we will manually analyze the results.

5.5.1 LibSeeker’s Properties

In § 5.3 we presented a set of properties that the ideal SDK detection methodology would have. To validate whether LibSeeker fits this description, we rely on the set of ground-truth apps introduced in § 5.2.1 to run a controlled experiment and evaluate these different properties.

- **SDK Agnostic:** We test LibSeeker against an app that contains the latest available version of nine proprietary SDKs. Our tool works out of the box without the need to generate any profile from third-party libraries for the detection to work. We prove that this is indeed the case, as LibSeeker reports all SDKs integrated in our ground-truth app. Furthermore, it reports paths that potentially belong to different sub-products of the same SDK, as both `com/facebook/ads` and `com/facebook/appevents` are part of the results. Recall that we report different paths that belong to third parties, but leave the decision of which belong to the same SDK to an analyst. This change of paradigm not only allows us to report SDKs without any a-priori knowledge but it also provides useful information so that the analyst can differentiate which parts of a given SDK are used in the app, which can be very important for privacy analysis in SDKs that do not serve a single-purpose such as Facebook or Firebase.

- **Robust Against False Positives:** We use our ground-truth app which contains different types of fake signals to asses whether LibSeeker is able to differentiate whether an SDK is actually used by the app. We confirm that none of these SDKs are reported by LibSeeker, as the confidence for all of them are below the established threshold due to the lack of enough signals.
- **Obfuscation Resilient:** We test whether LibSeeker is able to resist complex obfuscation techniques by analyzing our ground-truth app that uses several obfuscation layers from Obfuscapk [6]. One of the issues with this tool is that it prevents the app from actually being able to execute, thus limiting our ability to collect dynamic analysis signals. However, LibSeeker ś anti-obfuscation techniques allow us to still detect enough static analysis signals and reports all of the SDKs present on the ground-truth app.

5.5.2 Open-world Evaluation

While our set of ground-truth apps allows us to make an initial assessment of LibSeeker's capabilities, a set of self-controlled apps is not enough to conclude that our methodology is more accurate and reliable than previously proposed solutions. To that end, we compare the results reported by each tool to those reported by LibSeeker for our dataset of 5k apps in order to have a macroscopic view of the differences across these tools.

Figure 14 shows two set of boxplots, the first one portrays the Jaccard similarity between the set of SDKs reported by LibSeeker and each of the other solutions. The Jaccard similarity measures the number of elements that are common across two sets, and the value goes from 0 (no common elements) to 1 (exactly equal). This ratio is calculated as the union of the sets divided by the intersection [163]. It is important to note that LibSeeker reports paths that should be inspected by an analyst to convert them to actual SDKs. To overcome for the potential differences between these results and the rest of tools, we rely on a mapping between path names and actual SDKs that we manually compiled using already available mappings, online resources and results of previous research. Namely, we build upon the mappings provided by LibRadar, augmenting them by manually looking for updated information about which SDKs relate to which code-paths. For those code-paths that are nor present in this mapping, we look at the most commonly present in our results and create a manual mapping from online resources such as forums and company databases. Note that this automatic mapping covers the role of the analyst, allowing us to make an apples to apples comparison between tools.

From the similarity scores, we can infer that Exodus is the tool that that presents the lowest sim-

[6] Namely method renaming, field renaming and class renaming

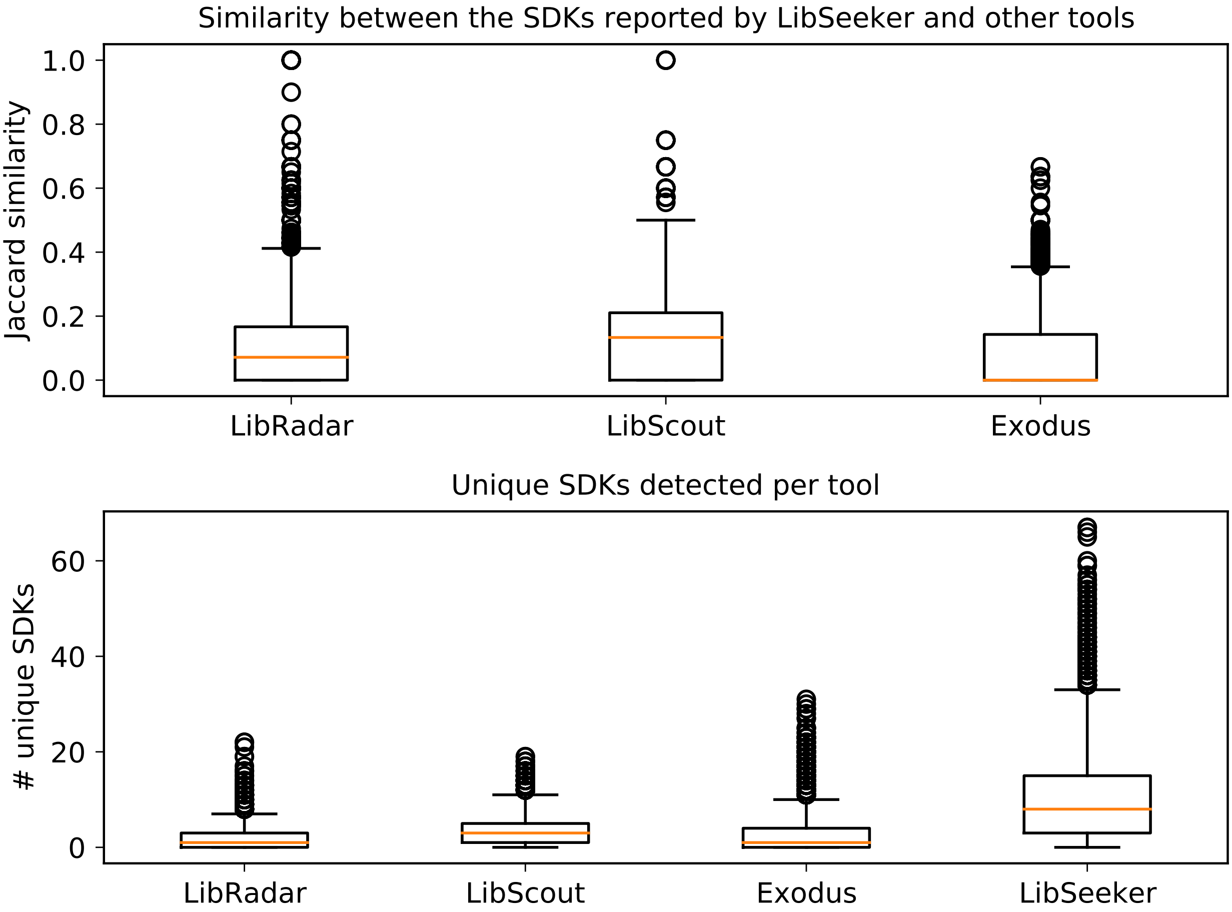


Figure 14: Boxplots showing the Jaccard similarity between the set of SDKs reported by LibSeeker and the rest of tools and the number of unique SDKs reported by each tool.

ilarity. This can be explained by the fact that Exodus does not report development support libraries, as this is a privacy tool that only reports those SDKs that they have labeled as “trackers”. The next tool in terms of low similarity is LibRadar, with a median of around 0.1 and a max value below 0.5, without accounting for outliers. As previously discussed, LibRadar’s profiles are very outdated (2018) which results in this tool systematically failing to detect newer versions of well-known SDKs. Finally, as LibScout’s profiles have been more recently updated than Libradar’s, we can see that the similarity with LibSeeker is also higher.

The second set of boxplots in Figure 14 shows the number of SDKs that were reported by one of the tools but not by LibSeeker. The plot for LibSeeker shows those SDKs that are **only** found by our tool. The second set of boxplots The fact that a given tool uniquely reports a given SDK can have two explanations. This can happen because the tool is more accurate than others and is able to uniquely detect the presence of a given third party; or because the tool is mistakenly reporting the presence of a third party due to some error on its methodology.

We can observe that LibSeeker uniquely reports more SDKs than all other solutions, with extreme cases of apps in which it finds 60 third-party components that none of the other solutions can find. We manually analyze the code of the app, due to its high number of reported SDKs to figure out whether LibSeeker is over-reporting. This apps is called *au.com.nexty.today*, an Australian app with over 100k downloads in the Play Store. We find clear examples of SDKs that are missed by the rest of tools, such as Twitter, Alibaba, OneSignal or ReactiveX. LibSeeker also reports a number of development support libraries as third parties, which appear to be missed by other tools, e.g., Android Animations Library, Apache HTTP, Fast JSON, Smooth Progress Bar and Floating Action Button. Upon further investigation, we find that the SDKs that LibRadar and LibScout fail to report do not match any of the profiles available in their database. Exodus, fails to report many of the SDKs because they are not labeled as "trackers" on its database.

We can also see that there are instances in which onther solutions report SDKs that are not reported by LibSeeker. On average, LibScout is the tool that reports a higher number of unique SDKs, but both LibRadar and Exodus have extreme cases with over 20 SDKs that LibSeeker did not report. To be able to understand whether this is a result of LibSeeker missing SDKs or other tools over-reporting due to the limitations of their methodology we need to take a deeper look at some of these apps individually. To that end, we randomly select a subset of 25 apps that we will use to further investigate these differences across solutions.

Microscopic View

While a full dataset of 5k apps allows us to obtain a macroscopic view of the differences between LibSeeker and the rest of tools, we cannot make any specific observation about why these differences exist. To that end, we randomly select a subset of 25 apps for which we manually verify all the mismatches between LibSeeker and the rest of tools and evaluate what is the source for the disagreement.

Figure 15 shows the Jaccard similarity score for each pair tools. These results allow us to make an important observation, the level of agreement across tools is extremely low with no pair of tools having a similarity index above 0.25 for the 25 analyzed apps. This shows that, differences in the methodologies of these tools can lead to very different results. LibScout and LibRadar's methodologies are very similar, resulting in a close Jaccard index for both of them, 0.16 and 0.17. Exodus' results are slightly closer to those of LibSeeker, with a value of 0.22 in similarity, despite being a privacy tool that only reports "trackers". This is a direct result of Exodus' profiles being actively updated to keep up with changes in the ecosystem.

We further explore the reason why these SDK detection tools present such a low similarity score in their results. To that end, Figure 16 shows the values of the intersection and the union between the set of reported SDKs by LibSeeker and each of the other detection tools as well as the unique number of SDKs that each one reports. Recall that the Jaccard index shows the value of the intersection divided by the union, and thus, the closer than these values are, the more elements both sets have in common. Furthermore, when the intersection value is close to the number of unique SDKs reported by one tool, this means that such tool is responsible for reporting most of the SDKs. In general, we can see that for most apps, the number of SDKs uniquely reported by LibSeeker is much closer to the value of the union that the number of SDKs reported by the other tool. Thus, as we have shown, in most cases LibSeeker is able to find SDKs that are missed by other solutions. Leveraging the reduced side of this dataset, we dive deeper into what causes these differences between LibSeeker and each specific tool.

LibRadar. Figure 16 shows that LibRadar does not have a high level of similarity with the results reported by LibSeeker. In fact, the average similarity for the 25 apps is below 0.2 with a maximum

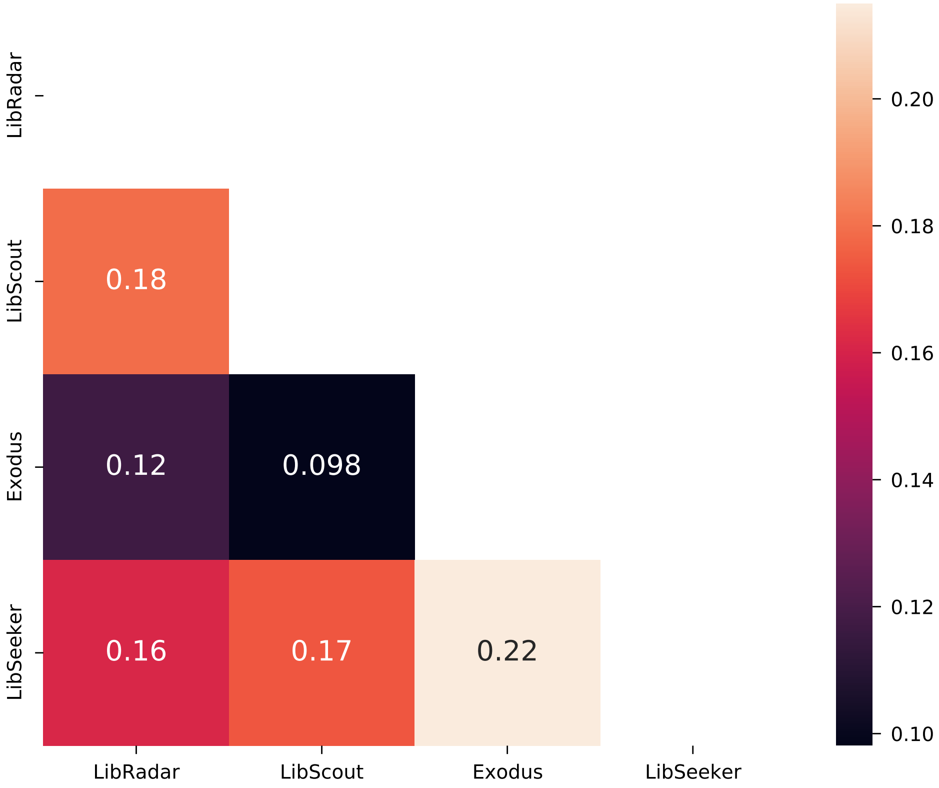


Figure 15: Heatmap showing the Jaccard similarity for each pair of the analyzed tools. The legend gradient highlights the value of this ratio that goes from 0 (no overlap between sets) to 1 (both sets are equal)

value of 0.75. LibRadar reports, on average, between two and three unique SDKs, with the most extreme case being 13. We first turn our attention to this outlier, and find that while there are three SDKs in common with LibSeeker's output (namely PlayHaven, Amazon and Tune), LibRadar reports SDKs for which we find no evidence of their use in the main app's code. This includes several Google related services, namely Google Ads, Google Play, Google Mobile Services (GMS) and Google Cloud Messaging (GCM). While we find that the app includes classes related to these SDKs, LibSeeker does not find any evidence about their actual usage by the app. As discussed in § 5.2.1, LibRadar does not make this distinction and reports the presence of these SDKs regardless of them being unused. We find that this is also the reason for LibRadar over-reporting Flurry, Apache Common, SLF4J and JsonPath.

We analyze which are the SDKs that are more often reported by LibRadar and not by LibSeeker and find that the most common are those related to Google products. Namely we find over-reporting of Google GMS in 6 apps, Google Ads in 5 apps and Google Play Services in 3 apps. In all of these apps we do find the relevant code for these SDKs, but see no further evidence of their usage by their man app. Other notable examples are the development support libraries Bolts and Glide which are over-reported in 5 apps. As previous work has shown, this over-reporting happens either because some of these SDKs code is included in other libraries, or because they are part of dead code that is

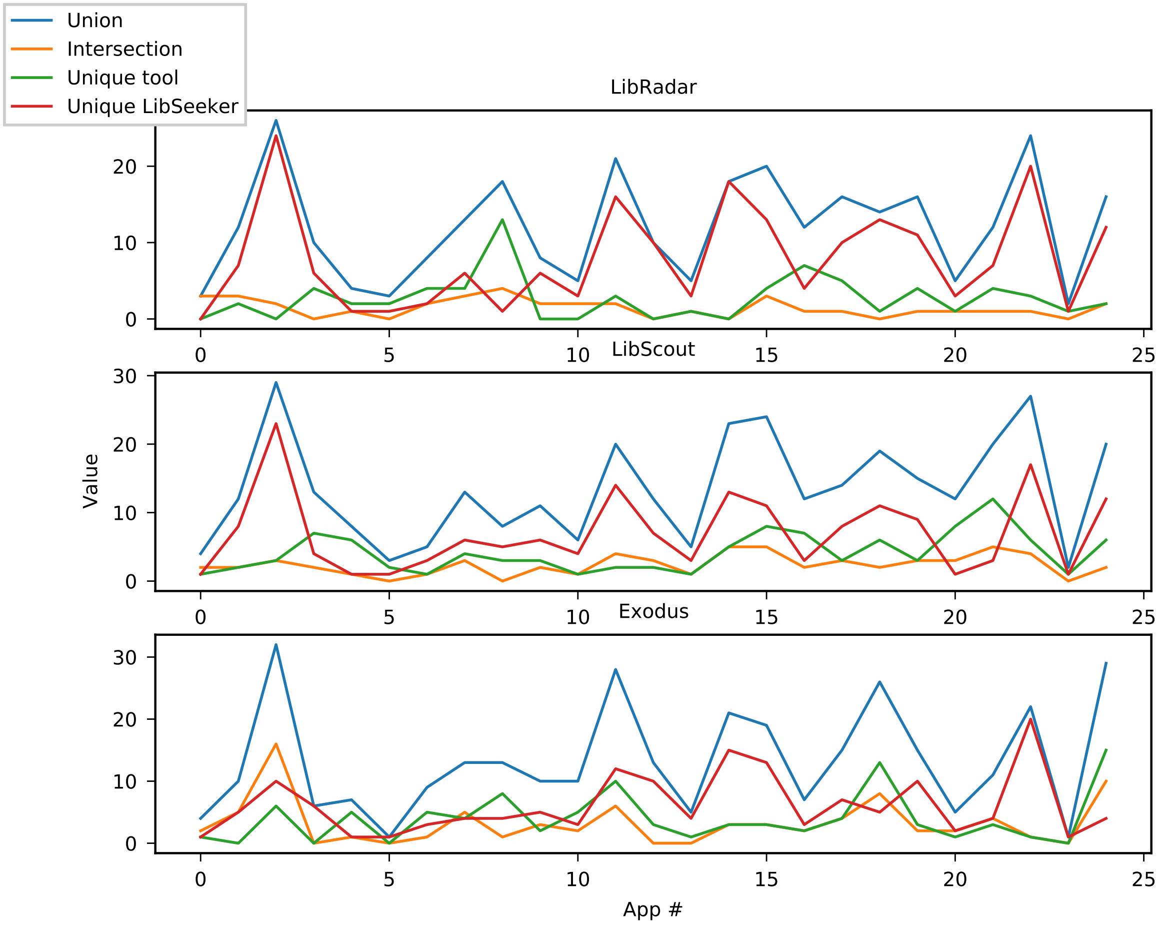


Figure 16: Union and intersection values for the set of reported SDKs by LibSeeker and the other detection tools and number of unique SDKs reported by each one of the solutions.

included but never used [167, 107]. Other examples of over-reporting by LibRadar include reporting *com/github* as an SDK, which is not enough to pin-point which library hosted in GitHub is actually included on the app.

We also look at the SDKs that are more commonly reported by LibSeeker but not by LibRadar. We observe the presence of well-known SDKs that are likely missed because of outdated fingerprints: Firebase (15 apps), Facebook (7 apps), Unity (6 apps), Appsflyer (4 apps) and InMobi (3 apps). By analyzing the runtime behavior of LibRadar when analyzing these apps, we confirm that these SDKs are not reported because the version found in the app cannot be matched to any of the fingerprints in LibRadar's database. The same issue happens with less known SDKs, that were not part of the initial set of libraries for which LibRadar generated a fingerprint. Examples of such SDKs are the paths "com/ljoy", "com/naver" and "io/reactivex".

LibScout. LibScout's results also differ greatly from LibSeeker, as the sets of SDKS reported by each tool have a jaccard similarity of 0.15 in average and a maximum value of 0.33. Figure 16 shows that, in most cases, LibSeeker uniquely reports the majority of SDKs that cause these differences across both solutions. On average, LibSeeker reports 9 unique SDKs per app while LibScout reports 4. To understand the reason behind LibScout not reporting SDKs, we first analyze which are the SDKs that it does not report more often. We find that Unity (7 apps), Amazon (5 apps), InMobi (5 apps) and Appsflyer (4 apps) are never reported by LibScout, indicating that this tools lacks appropriate profiles to be able to detect these SDKs. By analyzing LibScout's profiles we can find that indeed it lacks a profile for Unity, InMobi and some Amazon products such as the advertisement SDK. In the case of Appsflyer, we argue that the SDKs present in these apps were above the latest available version in LibScout's profiles [7].

We also look at those SDKs that are only reported by LibScout, to determine whether this tool is over-reporting SDKs or if LibSeeker might be missing results. Interestingly, we find that the majority of these mis-matches are caused by just three products: Facebook (15 apps), OkHTTP (15 apps), and Google GSON (14 apps). We manually look at the code of these apps and find that, the main issue is that LibScout is reporting the presence of these SDKs in apps in which there are not enough signals showing that the SDK is ever used by the main app. However, we also find instances in which LibScout reports Facebook in apps in which the code does not contain anything related to this SDK. Upon further analysis of LibScout's results, we find that part of the development support library Android X is matched to one of LibScout's Facebook profiles, leading to over-reporting in a high number of apps.

Exodus. Figure 15 shows that Exodus has the lowest similarity with each of the other tools but the highest with LibSeeker. As we have previously discussed (§ 5.1), Exodus is a privacy solution that only reports SDKs that are present in their "tracker" database. In fact, we see that LibSeeker consistently reports SDKs that are missed by Exodus for this reason, such as Picasso (3 apps), JSON org (3 apps), OkHTTP (3 apps), Google GSON (2 apps) and Retrofit (2 apps). We also find that Google Mobile Services is not reported by Exodus in the 17 apps in which LibSeeker finds its presence. Upon analyzing Exodus' "tracker" database we find that they lack a profile for this SDK, justifying its inability to detect it. There are also instances of SDKs for which Exodus has a profile but still fails to detect. The most commonly missed SDKs by Exodus are Firebase (3 apps), Ogury (2 apps) and Braze (1 app). When we analyze the apps in which these SDKs were found, we see that they use basic obfuscation techniques such as class renaming that defeat Exodus' ability to detect these SDKs as they no longer match their string-based profile matching. As discussed in § 5.1, Exodus' is more prone to miss SDKs in the presence of basic obfuscation techniques than other state-of-the-art tools.

Finally, we observe 7 cases of libraries that are reported by Exodus but not by LibSeeker, the most common one being Google Ads twice. Upon manual review of the code of these SDKs, we find that similar to other static analysis solutions, Exodus reports SDKs for which there is no evidence of their use by the main app's code.

5.6 Case studies

5.6.1 SDK Prevalence

One of the most common uses for SDK detection tools is to report on the ecosystem of SDKs in the wild, including the prevalence and popularity of SDKs across public apps. After showing the shortcomings of other state-of-the-art tools, we now present an SDK-centric view in which we rely

[7] At the time of writing the latest version is 6.9.2 and the profile for the latest version is 4.10.1

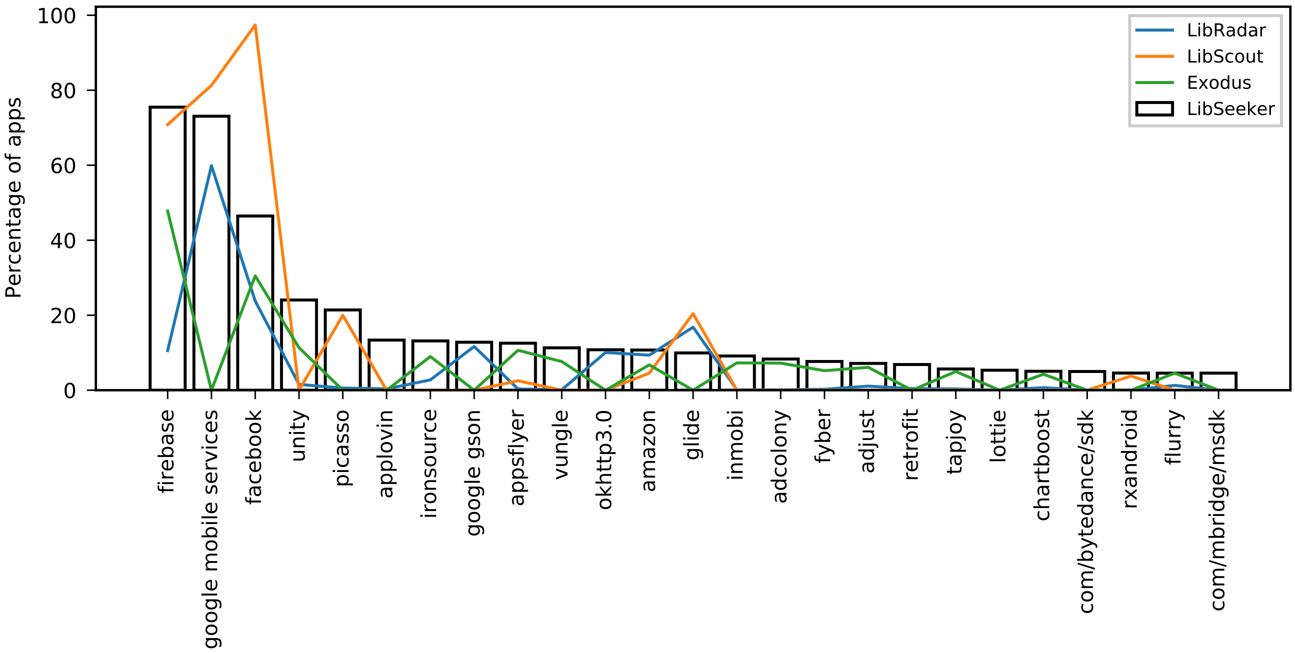


Figure 17: Prevalence according the different detection tools of the top 25 SDKs as reported by LibSeeker

on our manual map of paths reported by LibSeeker to the actual SDK behind them to study their prevalence on our dataset of 5k apps.

Figure 17 shows the top 25 SDKs reported by LibSeeker, including how prevalent these SDKs are according to the other three tools. This shows how the choice of a given SDK detection tool can highly alter the results of previous research, as we not only observe big differences with LibSeeker but also across different tools. Our results are in-line with previous analysis of the Android ecosystem of third-party libraries, with Google products (GMS and Firebase) being present in over 70% of the analyzed apps [230]. Rounding out the top five we observe Facebook (60%), Picasso (30%) and Unity (25%). This results also show that SDKs follow a long-tail distribution, in which a couple solutions are highly popular and then there are a number of less-popular SDKs with a similar prevalence number (around 10% of apps).

We observe that LibScout reports a similar distribution for the top 4 SDKs, but the prevalence numbers for LibRadar and Exodus are very different to those reported by LibSeeker. These differences across tools are further demonstrated when looking at the top 5 SDKs per each of these tools, which present clear differences. In the case of LibRadar these are: Google Mobile Services (60%), Facebook (23%), Bolts (20%), Glide (17%) and Google GSON (12%). For LibScout the top 5 SDKs by prevalence are: Facebook (98%), Google Mobile Services (81%), Firebase (70%), Google GSON (56%) and Retrofit (38%). Finally, for Exodus the results show: Firebase (47%), Google Ads (31%), Crashlytics (31%), Facebook (30%) and IAB (13%).

The ability to accurately study the prevalence of SDKs on Android apps is relevant for several reasons, on the one hand it allows to study the ecosystem and understand changes and trends. On the other hand, prevalence is important for the study of security and privacy, as once an issue is spotted in a given SDK knowing the number of apps in which the app is present can help assessing the number of users that might be affected by these issues.

Table 5.4: Permission escalation results

android.permission	App Only	SDK Only (ATS)	Both (ATS)
ACCESS_FINE_LOCATION	1%	88% (9%)	10% (23%)
ACCESS_COARSE_LOCATION	1%	88% (9%)	10% (23%)
ACCESS_NETWORK_STATE	1%	96% (4%)	2% (60%)
USE_FINGERPRINT	0%	89% (7%)	10% (5%)
VIBRATE	23%	66% (21%)	9% (0%)
CHANGE_WIFI_STATE	35%	50% (20%)	15% (0%)
ACCESS_WIFI_STATE	16%	72% (47%)	11% (22%)
ANSWER_PHONE_CALLS	0%	100% (66%)	0% (0%)
BLUETOOTH	20%	58% (15%)	20% (42%)
NFC	25%	75% (66%)	0% (0%)
GET_ACCOUNTS	44%	39% (33%)	15% (33%)
SET_WALLPAPER	66%	11% (0%)	22% (0%)
AUTHENTICATE_ACCOUNTS	77%	22% (0%)	0% (0%)
READ_PHONE_STATE	0%	95% (61%)	4% (0%)
BLUETOOTH_ADMIN	40%	40% (0%)	20% (0%)
KILL_BACKGROUND_PROCESSES	20%	60% (0%)	20% (0%)
WAKE_LOCK	50%	50% (0%)	0% (0%)
WRITE_SYNC_SETTINGS	0%	100% (0%)	0% (0%)
READ_SYNC_SETTINGS	0%	100% (0%)	0% (0%)

5.6.2 SDK Privilege Escalation

As we discussed in § 3.1, Android's permission model is app-centric, which means that SDKs run with the same privileges than the host app. This opens the door for SDKs to take advantage of the permissions granted to the app itself and use them for secondary purposes such as advertisement and tracking. Not only that, but some SDKs might be responsible for the request of a given permission that would otherwise not be needed by the app itself. To that end, we create a mapping of Android APIs (i.e., functions called within the app's code) to the AOSP permissions that protect them. Note that Google does not include such a mapping in their documentation. Therefore, we leverage three complementary sources: (*i*) the mappings implemented in Android Studio (Android's official IDE), which contains scripts to warn developers if they use an API without requesting the associated permission [125]; (*ii*) mappings extracted by parsing the AOSP source code to extract the methods that use the `@RequiresPermission` annotation, part of the AndroidX and Android support libraries [132, 131]; and (*iii*) the mappings generated by prior work (Axplorer [50]) for older Android versions (Android 5 and below). We acknowledge that, while we update the mappings released by prior work, our mappings might still be incomplete (something that we cannot measure due to the lack of ground truth). Armed with these mappings, we analyze the main code of each app and for every permission protected API call, we check whether it originated due to the app itself or a third-party SDK.

Table 5.4 shows for each permission used in by the app's in our dataset, whether the API that requires such permission was called by the app's code or by a third-party SDK. Furthermore, we include the percentage of SDKs that we were able to classify as advertisement and tracking (ATS), using a manual effort to map SDKs to their functionality according to the taxonomy presented in § 2.3.1. These results show that SDK piggybacking on the app's permissions is indeed a common scenario, as we see that location related permissions are used both by the app and an SDK in 20% of the cases. Similarly, in 5% of the apps that have access to the *READ_PHONE_STATE* (which allows access to permission identifiers) both the app and the SDK make use of this. However, the numbers in which only a given SDK is accessing the functionality enabled by a given permission is much higher. In over 80% of apps in which the location permissions are requested we only see evidence of a given SDK accessing these features. Similarly, in over 90% of the apps which request access to phone identifiers, we only see code from a third party accessing such features. Note that the fact that a given permission is used only by an SDK is not always an issue from a privacy perspective, as many third-party libraries are mean to help developers including code in their app (e.g., the Android Support Library for Android features or OkHTTP for network protocols). To that end, we include the number of SDKs that we are able to match to an advertisement and tracking service using a manual classification effort, based on previous research and intelligence gathered online. We see that 61% of the SDKs that are responsible for the *READ_PHONE_STATE* are related to tracking companies.

In fact, we take a deeper look at the usage of permissions on a per-SDK basis for those that we are able to classify as tracking related. For each of these tracking SDKs, we analyze the percentage of apps in which the SDK is the only one accessing a given permission and when it does so in conjunction with the actual app. Between those SDKs that access the *READ_PHONE_STATE* permissions we find that that Localytics and Mixpanel are accessing this permission uniquely in all apps in which we find them. Other examples are Adjust, InMobi and Amazon Ads which do so in over 75% of the apps. In terms of those SDKs that access the location we find that AppBrain, Verizon Ads, Baidu, MoPub, Appodeal are fully responsible for the request of this permission. Other examples InMobi or Adjust in which only the SDK accesses this permission in over 75% of the cases (with the other 25% being accessed both by the SDK and the app).

These results show that, while SDKs piggybacking of the app's permission is common, the usage of permissions only by an SDK is very extended. While previous work showed that app often times had more permissions than needed mainly due to legacy code and code copied from the Internet [106], our analysis shows that in most cases this over-privilege is a direct result of SDKs accessing data that would otherwise not be accessed by the app. We argue that one of the reasons for this behavior is the rise of the Ad-Tech ecosystem, as we see that those permissions directly related to advertisement and tracking (e.g., those related to the location of the user or to device identifiers) that are used only by SDKs in over 90% of the cases.

5.7 Limitations and Future Work

In this chapter, we presented a novel methodology that combines static and dynamic analysis to be able to accurately detect SDK presence without the need for a-priori knowledge. However, this tool has its own limitations:

- **Performance:** To be able to detect the usage of SDKs by the main app's code, LibSeeker looks for cross-references for each potential third-party package. This is a costly operation, specially for bigger apps and it can take in the order of hours to finish. One of the bottlenecks in this process is the use of Androguard [15], which could be solved by building a custom-made module for this operation that is more cost-efficient.
- **Anti-obfuscation:** While the combination of static and dynamic analysis signals used in our methodology allow us to detect most SDKs, even if obfuscated, to detect SDKs that rely on ad-

vanced obfuscation techniques, LibSeeker relies on a database of fingerprints of un-obfuscated libraries. To avoid having the same coverage issues than other static analysis SDK detection techniques, for every app that is analyzed by LibSeeker, this database is populated. Therefore, this tool automatically takes care of the process of keeping and up-to-date database of libraries. However, we acknowledge that we had to build this initial database using our dataset of 5k apps as part of the development process of LibSeeker.

- **Grouping signals:** As discussed in S 5.3, LibSeeker relies on a set of signals obtained from different sources which can be difficult to group together in order to make specific observations about SDK presence. To that end, rely on a measure of string similarity across signals, which we empirically test to make sure that the used threshold works appropriately. However, it is still possible that this methodology leads to cases in which a signals is wrongly matched or cases in which a match is missed.

Our methodology based on the combination of static and dynamic analysis techniques can be leveraged to further investigate security and privacy issues such as:

- **SDK versions:** Using a different combination of static and dynamic analysis signals could allow us to determine what is the precise version of an SDK being used by an app. One example of such a signal is the User-Agent, since some SDKs use their own modified version of this string that can contain versioning information. This information could be leveraged to study the rate at which app developers update SDKs, and study general trends such as whether some app updates only contain changes to the embedded SDKs.
- **Use of permissions by SDKs:** In this thesis, we have shown that SDKs are often to blame for the request of permissions by Android apps. By extending our dynamic analysis module to gather information about the type of data collected and accessed by different apps we could further investigate how this over-privilege translates in personal data collection and sharing by SDKs during runtime.

5.8 Discussion

5.8.1 Application Developers and SDK Providers

Intuitively, an app developer will select the third-party SDK that provides the best functionality, but the question remains whether the SDK's resources usage or data sharing practices are ever considered in that decision-making process. It is imperative that developers understand the risks that many SDKs might bring to users' privacy. Application developers are liable for any regulatory violation that occurs within their application, even those inflicted by third-party components. With new legislation such as the GDPR, the fines can add up to €20 million or four per cent of the company's worldwide annual revenue from the preceding financial year, whichever is higher [121]. It is critical that developers play a more central role in taking responsibility for their decisions to bundle third-party SDKs and that they follow the privacy-by design principles. For that, it is necessary to also put SDK providers under scrutiny, demanding more transparency about their data collection practices and purposes and about their business models. This could be complemented with contractual agreements between app developers and SDK providers allowing developers to have additional legal guarantees and a better understanding of the privacy implications associated with a given SDK.

5.8.2 Stricter App Store Policies

Mobile platform providers should strive to audit applications and SDKs in order to improve and safeguard users' privacy. The Google Play store has checks in place to improve the privacy of applications. In early 2019, Google forced applications requesting call and SMS permission to either be the default messaging or calling app or to submit a special form explaining why such permissions are necessary for the app [122]. Furthermore, in newer Android versions (Android 10) Google has added restrictions for using unique non resettable identifiers (such as the IMEI) and for accessing alternate methods to infer location without requesting the appropriate permission [28]. Google has also published a list of self-certified third-party SDKs suitable for children's applications [128]. This list is a great resource for developers of children oriented applications, as it reduces the search scope before making the decision to bundle a third-party SDK in their application. In this case, as these SDKs are self certified, developers must trust that those components do indeed comply with existing regulation. There is no public information on whether Google verifies the claims made by each provider. Similarly, it is still possible to find applications that do not carry a privacy policy and examples of incomplete policies [212]. While we acknowledge that a thorough privacy analysis of all applications submitted to the market is a technically complex and costly task, we believe that these enforcement mechanisms could benefit from including some of the auditing techniques developed by the research community. These policies are not specific to Android. Apple's App Store provides very exhaustive recommendations for app developers to minimize privacy damage to users [42]. These recommendations are focused on helping developers successfully pass their strict app review process prior to publication. In addition to minimum quality checks and recommendations—e.g.,, releasing bug-free software and offering appropriate content—these guidelines also discuss the need for including complete privacy policies (data access, and third-party SDKs, data minimization, and access to user data, among many others).

5.8.3 Changes in the permission model

Current permission models are app-centric by design so there is no separation between SDKs or apps accessing a given permission. Application developers only need to declare the permission in the app manifest file. Therefore, when a user grants the application permission to access a resource, the user has no information about whether an SDK or the app itself will exercise this permission. This goes in the opposite direction of current legislation, which is making strides towards better transparency and informing users about data collection and sharing practices, including the recipients of such data. Android mentions the use of an explanation before requesting a permission as a best practice but does not enforce it in published apps [26]. If developers had to justify the inclusion of a permission request, users could make a more informed decision on whether to grant such permission or not. Additionally, the operating system could monitor and inform users whenever a protected method has been invoked by the actual application or an embedded SDK, and whether this has been disseminated to a server hosted on the internet. Unless users know which company is collecting personal data in an app, they will not be able to exercise their data rights per current legislation.

5.8.4 Certifying Bodies and Regulatory Actions

Trusted certification authorities could independently validate and certify the data collection practices of SDK providers. Article 42 and recital 81 and 100 of the GDPR propose ways to ensure compliance by controllers and processors through certification mechanisms and data protection seals or marks [119]. However, previous certification attempts such as COPPA's Safe Harbor have proven ineffective. As revealed by academic research, many applications certified by certification authorities still incur into potential violations of the COPPA law [238]. It is unclear whether certification mechanisms can be effective in preventing deceptive behaviors and malpractices by third-party SDKs. The success of any certification scheme largely depends on the quality and depth of the certification process—i.e.,, the length for which the process is going to make sure that the SDKs are in compliance with any regulations and for how long they will be able to bring out any potential violations. The use of auditing tools could play a fundamental role in the validation of the claims made by SDK providers from a technical standpoint. Additionally, regulators must stay diligent and continue investigating any privacy malpractice on mobile applications. The FTC has previously acted towards SDK providers [105, 103], contributing to hold companies accountable when they do not respect users' privacy. These actions also have a valuable educational component. Once a regulatory action shows that a given company's behavior constitutes privacy malpractice, other companies with similar policies might take additional precautions to protect their brands, reputation, and business, and avoid regulatory scrutiny and fines.

Chapter 6 Conclusions and Future Work

In this thesis, we have proved the benefits of combining static and dynamic analysis for the security and privacy analysis of Android apps, showcasing these benefits with three case studies. Most prior research has relied either on studying potential privacy and security issues by analyzing the code of apps or showing evidence of their behavior via dynamic analysis of their runtime behavior. However, these analysis have limitations when used in isolation, such as the difficulty to analyze apps that are obfuscated or that contain dynamically loaded code when using code analysis, as well as the difficulty to derive from the app's code whether a given behavior happens during runtime or is part of dead code. Dynamic analysis can provide evidence of runtime behavior, but exercising apps to be able to find evidence of security and privacy issues can become a hard task given the lack of automatic testing techniques that can fully test an app. However, we have shown that a methodology that combines both analysis can provide sound, scalabale and complete analysis of the risks that Android apps can pose to users from a security, privacy and regulatory compliance analysis point of view. We have shown different ways in which the combination of both methodologies performs better than using only one method in isolation, by complementing the limitations of each method. We have shown ways in which static analysis can be used to drive the inputs used to dynamically test apps, using the potential risks discovered statically to generate appropriate inputs that can provide better coverage when dynamically testing apps. We have also proved how dynamic analysis findings can be later complemented with static code analysis that allows to detect instances of security risks that could not be empirically discovered during the analysis of runtime behavior. Finally, we have developed a methodology that relies both on static and dynamic analysis signals to solve well-known issues in state of the art SDK detection tools (e.g., improved accuracy when the code is obfuscated and reducing false positives due to dead and unused code).

The combination of static and dynamic analysis can be helpful for different actors other than researchers. For instance, automatic methods similar to those proposed on this thesis can be employed by app stores to provide a more complete testing of apps prior to their public release. This would improve their ability to detect harmful apps before any user can download them, helping to sanitize public app repositories. Similarly, external companies can rely on a methodology such as the one that we propose to provide certification to app developers. Prior work has shown that these certifications seldom times work correctly [238]. A good certification scheme based on a sound methodology can allow developers to test their apps prior to publication, showing good faith in an attempt to be transparent to users about their data collection practices. Furthermore, users could trust apps that have been eternally audited, allowing them to be informed in their decision of whether to trust the data collection practices of a given app before installing it.

More concretely, in this thesis we have proved the benefits of combining static and dynamic analysis methods through three different case studies:

- **Regulatory Compliance.** We present a regulatory compliance analysis of the parental control apps ecosystem in Android. To do so, we first use static analysis techniques to discover potential privacy issues by studying the permissions requested by these apps and third-party SDK embedded in them. Then, we build upon the insights generated through static analysis to drive the way in which we exercise the apps to analyze their runtime behavior, which allows us to obtain evidence of potential violations of current legislation. Our multilateral analysis of parental control apps brings to light a large number of undesirable behaviors regarding children's privacy. First, parental control applications request a large number of invasive permissions, which combined with the number of analytics and advertisement SDKs embedded in these apps pose a serious threat for children. Indeed, we find empirical evidence of data sharing practices with third parties: 72% of the apps share data with a third-party SDK (§ 3.5.2), and in 67% apps this sharing happens without explicit and verifiable parental consent (§ 3.5.3). We also find cases of apps uploading sensitive data to online servers without the use of proper encryption. Finally, our analysis of the apps' privacy policies reveals that they are far from clear about their data collection practices, and usually under-report on what data is shared with other services (§ 3.6). Such practices put in question the regulatory compliance of many of these apps.

- **Security Analysis.** We rely on a novel methodology that combines static and dynamic analysis techniques to find instances of Android apps and SDKs accessing permission-protected data without holding such permission. Namely, we find potential cases of un-authorized access to data analyzing network communications in search of personal data sharing for which we had not granted permission. We then manually analyze the code of one example per network destination, allowing us to detect the exact covert or side channel that allowed such un-authorized access. This allows us to build a static fingerprint of the code that allows apps and SDKs to perform such attack. We use this code fingerprint to search for other apps potentially taking advantage of these attacks. This novel technique allowed us to extensively test more than 88,000 apps finding a side and covert channels that potentially impact users in the hundreds of millions. We find evidence of third-party libraries provided by two Chinese companies—Baidu and Salmonads—independently making use of the SD card as a covert channel, so that when an app could read the phone's IMEI, it would store it for other apps that cannot (§ 4.3.1). We also showed that Unity was obtaining the device MAC address using `ioctl` system calls, which can be used to uniquely identify the device (§ 4.3.2). We found companies getting the MAC addresses of the connected WiFi base stations from the ARP cache, which can be used as a surrogate for location data (§ 4.3.3). Finally, we discovered one app that used picture metadata as a side channel to access precise location information despite not holding location permissions (§ 4.3.4).
- **SDK Detection.** We design a new SDK detection methodology that combines static and dynamic analysis techniques to extract different signals about third party presence from a given app without the need of any a-priori knowledge. Then, we rely on the combination of these signals to build a confidence score which allows us to detect whether third-party code present in an app is actually used. Contrary to prior work, we do not rely on pre-computed fingerprints of third-party SDKs to be able to detect them on the wild, allowing us to accurately detect third-party code regardless of the popularity of a given SDK. We compared the ability of our tool, LibSeeker, to three state-of-the-art detection tools (i.e., LibRadar, LibScout and Exodus), using both custom-made ground-truth apps and real world applications publicly available in the Play Store. We showed that our methodology is more accurate, as it does not over-report SDKs that

are not used by the main app. Furthermore, its ability to detect SDKs without a-priori knowledge allows LibSeeker to detect new versions of popular SDKs as well as unpopular SDKs that are often missed by other tools (§ 5.5). Finally, we showed that LibSeeker's abilities are helpful for the security and privacy analysis of Android apps via two case studies (§ 5.6). We studied the prevalence of SDKs in 5k apps downloaded from the Googles Play Store, showing that results can greatly vary depending on the detection tool of choice. Finally, we showed that most permissions are either only used by a given SDK, leading to over-privilege in apps; or used both by the app and SDK, suggesting that SDKs are leveraging the lack of privilege separation between apps and third parties in Android.

6.1 Future Work

The technologies presented in this thesis have a number of timely and relevant applications, that can help improve the privacy of users. Some examples of these applications are the automatic analysis of Android apps to analyze compliance with applicable regulation (e.g., GDPR and COPPA), validating developer claims in new self-declared data collection practices info in app stores [129, 41], and verifying external certification bodies such as ioXt [159]. In this Section, we expand on other possible avenues where our methodology can be helpful, outside of Android Privacy and Security research.

Tools for iOS Apps. While there is a long history of security and privacy research and analysis tools for Android apps, the amount of research on Apple's counterpart is considerably smaller. One of the main reasons for this difference is that, contrary to Android, iOS is a closed ecosystem, making it harder to create research tools for the analysis of apps. While there's an arsenal of tools for Android research, most iOS research has relied on ad-hoc methodologies that are harder to replicate. The next logical step after proving the benefits of a methodology based on the combination of static and dynamic analysis to hurdle their limitations when used in isolation is to use it as a baseline for iOS research. However, there is a need to start working towards having an open and proven set of tools that work well in this platform. There has already been some research in this avenue, showing that Apple's ecosystem is not necessarily better for privacy than Android's [173]. In fact, previous work has shown that both Apple and Android devices send information to their Apple and Google (respectively) upon set-up, even if the user has not opted-in to this data sharing [177].

The statement that Apple devices are better for the privacy of users, might not be completely accurate regardless of the privacy of their being a big part of Apple's marketing strategy [40]. The lack of accurate and scalable research methodologies in iOS prevents us from studying whether this holds true in their ecosystem apps and third-party libraries that share the same interest in obtaining personal data from users as in the Android ecosystem. In fact, preliminary results show that iOS apps are not more secure than their Android counterparts when it comes to SDK presence and data collection practices [173]. Therefore, we argue that there is a need to build new tools for being able to study the privacy and security of the Apple ecosystem. This would allow the research community to better understand the privacy and security issues of this ecosystem, put them in context with those already reported for Android, and adopt novel technologies such as the one presented in this thesis to further investigate and improve the privacy and security of phone users, regardless of their operative system of choice.

Smart TVs and IoT. Smart devices are increasingly becoming more common in our everyday life, including all types of commodities (e.g., smart fridges, smart lights and smart TVs). Similar to smartphones, these platforms allow the installation of third-party software and have access to very sensitive data about users (e.g., health data, location of the user or other devices in the house), which makes them potentially dangerous for the privacy of users. In fact, previous research has demonstrated the lack of security configurations [280, 218] in IoT devices and Smart TVs and the presence of a highly invasive tracker ecosystem [197, 283, 160].

Therefore, there is a need to create new methodologies that allow to study the behavior of these devices at scale. Interestingly, many of these devices are actually based on the Android and other Linux-based OSes, which means that existing methodologies can be applied to these new platforms. However, the testing of smart devices has a number of unique limitations, inherent to the way in which devices work. One limitation when trying to analyze the runtime behavior of these devices is the difficulty to generate user inputs. Contrary to Android apps, random input generators such as the Android Monkey cannot be used, and manually testing them is not a scalable solution. Previous work has relied on different solutions, such as building "smarter" monkeys that map the UI of apps to generate inputs [283] or that are optimized for specific devices such as smart TVs [197]. One way in which our methodology can be helpful to study smart devices, would be to perform code analysis of the device to understand the type of inputs that can triggers certain behavior during runtime and use it to drive dynamic analysis. In fact Redini et al. rely on a similar methodology, in which they extract valid fuzzing behaviors from analyzing the companion app of the device [234]. We acknowledge that, since many of these devices are proprietary, accessing their code can be an extremely complex manual effort, if even possible at all. Another limitations for the analysis of IoT devices is their interoperability, as they normally do not exist in isolation. It is common for smart devices to either communicate with its own companion app on the phone, or with different devices on the household. Therefore, these devices cannot be studied in isolation, and there is a need to create a test environment in which devices can communicate with each other. One way to overcome this issue is to create a test-bed with several IoT devices [241, 218] that can interact with each other. However, since there are a number of different manufacturers, there is a lack of methodologies that allow to analyze each device without the need for ad-hoc methodologies, hindering the scalablity of these methods.

Automatic Regulatory Analysis. As we have shown in this thesis, the use of a hybrid methodology that relies on both static and dynamic analysis can be beneficial for the regulatory analysis of apps. However, one of the main issues with regulatory compliance analysis is the need to interpret legal text, to be able to put in context with app's behavior. Prior work has shown that privacy policies are often hard to understand [164, 216, 258] and that they are often incomplete [212]. This makes it hard to rely on tools to automatically extract information from app's policies and use it to analyze whether they are in-line with the discovered behavior. There has been some advances in the Natural Language Processing (NLP) community to improve our capacity to automatically parse legal text [150, 14, 54]. However, most of these solutions are meant to either summarize the contents of the policy to users or to extract specific information (e.g., opt-out choices). There is a need for a methodology that extracts relevant information from a privacy policy and is able to put in context with the analysis of the app's behavior (i.e., reporting when a given piece of data is collected and not correctly informed on the policy).

We argue that, our methodology can be paired with the ability to automatically extract information from legal text to automatically analyze the regulatory compliance of Android apps. While a full assessment of compliance is complex as it requires from the interpretation of legal experts, many properties such as the proper use of secure network transmissions and the completeness of the disclosure about data sharing and collection practices on the privacy policy can be automatically tested. Namely, the testing of apps with static and dynamic analysis can allow to make observations about the app's behavior, which can then be put in context with the app's legal text.

Certification. The privacy of users has become a vital point of discussion in modern society, as shown by new privacy legislation across the globe [77, 261, 101] and new efforts from different companies to improve the privacy of their products. One such example are privacy labels which have recently been implemented both by Google's Play Store as their new "Data Shared" feature [129] and Apple's App Store "Privacy Nutrition Labels" [41] . The idea behind this feature is that developers can input data about the type of data that is collected by their app, helping users make an informed decision about whether to install it or not. However, whatever the developers declares should not be fully trusted, as they have an interest of portraying to be privacy friendly but also to collect personal data. Similar certification efforts exist for IoT applications, namely ioXt [159], which is an alliance that strives to make IoT products secure. As part of this mission, this alliance includes a certification process in which manufacturers and developers can get their products officially certified as being secure for end users.

However, previous research has shown that self-certification and official certifying bodies seldom times works. For instance, apps that had self-certifications about COPPA compliance were found to be in potential violation of this law [238]. We argue that there is a need to verify the validity of these certifications, which can be achieved with a methodology following the principles presented in this thesis, namely the combination of static and dynamic analysis for sound and complete privacy and security analysis. We believe that the distribution platform (i.e., App Stores) are on the perfect vantage point to implement such solutions to make sure that all published apps are being truthful in the way that they portray themselves to users. Most of these platforms already employ similar solutions to detect malware and apps that do not comply with their publication policies. Therefore, they could tools build with a methodology such as one proposed in this thesis to tackle these issues.

Bibliography

[1] IDA: About. *IDA pro*. https://www.hex-rays.com/products/ida/.

[2] J. P. Achara et al.
WifiLeaks: Underestimated privacy implications of the access wifi state Android permission.
Tech. rep. EURECOM+4302. Eurecom, May 2014.
URL: http://www.eurecom.fr/publication/4302.

[3] AdColony. *AdColony Android SDK*.
https://github.com/AdColony/AdColony-Android-SDK. (accessed 2022-07-20).

[4] Adjust. *Get Started with the Android SDK*.
https://help.adjust.com/en/article/get-started-android-sdk.
(accessed 2022-07-20).

[5] *Adtech to drive Internet advertising industry to $1 trillion in 2030, forecasts GlobalData*.
Accessed July 22nd, 2022.

[6] Airpush. *The future of Mobile Advertising*.
https://airpush.com/.

[7] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin.
"A new sensors-based covert channel on android".
In: *The Scientific World Journal* 2014 (2014).

[8] Rodney Alexander.
"How to protect children from internet predators: a phenomenological study".
In: *ANNUAL REVIEW OF CYBERTHERAPY AND TELEMEDICINE 2015* (2016), p. 82.

[9] Suzan Ali et al.
"Betrayed by the guardian: Security and privacy risks of parental control solutions".
In: *Annual Computer Security Applications Conference*. 2020, pp. 69–83.

[10] Alipay. *Trust makes it simple*.
https://intl.alipay.com.

[11] Alphabet. *Alphabet - Home Page*.
https://abc.xyz/.

[12] Omar Alrawi et al.
"The Betrayal At Cloud City: An Empirical Analysis Of {Cloud-Based} Mobile Backends".
In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 551–566.

[13] Amazon AWS. *Getting Started with Android*.
/aws.amazon.com/developers/getting-started/android/.

[14] Benjamin Andow et al.
"{PolicyLint}: Investigating Internal Privacy Policy Contradictions on Google Play".
In: *28th USENIX security symposium (USENIX security 19)*. 2019, pp. 585–602.

[15] Androguard. *Welcome to Androguard's documentation!*

https://androguard.readthedocs.io/en/latest/. (accessed 2022-08-20).

[16] Android. *Usage of Android Advertising ID*. https://play.google.com/intl/en-GB/about/monetization-ads/ads/ad-id/index.html.

[17] Android. *Android developer manual: permission model*. https://developer.android.com/guide/topics/permissions/overview. 2018.

[18] Android. *Best practices for unique identifiers*. https://developer.android.com/training/articles/user-data-ids.

[19] Android. *Permissions overview*. https://developer.android.com/guide/topics/permissions/overview.

[20] Android. *Play Protect*. https://www.android.com/play-protect/.

[21] Android. *proGuard*. Accesed September 1st, 2022.

[22] Android. *Shrink, obfuscate, and optimize your app*. https://developer.android.com/studio/build/shrink-code. (accessed 2022-09-23).

[23] Android. *UI/Application Exerciser Monkey*. https://developer.android.com/studio/test/monkey.

[24] Android. *VpnService*. https://developer.android.com/reference/android/net/VpnService.

[25] Android Developers. *App Manifest Overview*. https://developer.android.com/guide/topics/manifest/manifest-intro.

[26] Android developers. *App permissions best practices*. https://developer.android.com/training/permissions/usage-notes.

[27] Android Developers. *Permissions on Android*. https://developer.android.com/guide/topics/permissions/overview. Accessed: 30-12-2021.

[28] Android Developers. *Privacy changes in Android 10*. https://developer.android.com/about/versions/10/privacy/changes.

[29] Android Developers. *Use of SMS or Call Log permission groups*. https://support.google.com/googleplay/android-developer/answer/10208820?hl=en.

[30] Android Developers. *VPN Service*. https://developer.android.com/reference/android/net/VpnService.

[31] *Android malware anti-emulation techniques*. Accessed June 22nd, 2022.

[32] *AOL Confirms It Is Buying Millennial Media In $238M Deal To Expand In Mobile Ads*. Accessed June 3rd, 2022.

[33] Simone Aonzo et al. "Obfuscapk: An open-source black-box obfuscation tool for Android apps". In: *SoftwareX* 11 (2020), p. 100403. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100403. URL: http://www.sciencedirect.com/science/article/pii/S2352711019302791.

[34] Apache. *Apache Cordova*. https://cordova.apache.org/.

[35] APKPure. *Homepage*. https://apkpure.com/.

[36] Apktool. *Apktool: A tool for reverse engineering Android apk files*. https://ibotpeaches.github.io/Apktool/.

[37] Appcensus. *Homepage*. https://www.appcensus.io/.

[38] AppCensus Inc. *Apps using Side and Covert Channels*. https://blog.appcensus.mobi/2019/06/01/apps-using-side-and-covert-channels/. 2019.

[39] Appium. *Mobile App Automation Made Awesome*. appium.io.

[40] Apple. *Privacy*. Accesed September 6th, 2022.

[41] Apple. *Privacy Labels*. Accesed September 6th, 2022.

[42] Apple Store. *App Store Review Guidelines*. https://developer.apple.com/app-store/review/guide-lines/.

[43] AppLovin. *AppLovin Integration*. https://dash.applovin.com/documentation/mediation/android/getting-started/integration. (accessed 2022-07-20).

[44] Arity. *Arity*. https://www.arity.com.

[45] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *PLDI 2014*. 2014.

[46] Kathy Wain Yee Au et al. “PScout: Analyzing the Android Permission Specification”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. ACM, 2012.

[47] G. S. Babil et al. “On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices”. In: *2013 International Conference on Security and Cryptography (SECRYPT)*. 2013, pp. 1–8.

[48] 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*. 2016, pp. 356–367.

[49] Michael Backes et al. “On Demystifying the Android Application Framework: Re-visiting Android Permission Specification Analysis”. In: *USENIX Security 2016*. 2016, pp. 1101–1118. ISBN: 978-1-931971-32-4.

[50] Michael Backes et al. “On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis”. In: *Proceedings of the USENIX Security Symposium*. 2016.

[51] Baidu. *Android SDK*. lbsyun.baidu.com/index.php?title=androidsdk.

[52] Baidu. *Baidu Geocoding API*. https://geocoder.readthedocs.io/providers/Baidu.html. Accessed: February 12, 2019. 2019.

[53] Baidu. *Baidu Maps SDK*. http://lbsyun.baidu.com/index.php?title=androidsdk. Accessed: February 12, 2019. 2019.

[54] Vinayshekhar Bannihatti Kumar et al. "Finding a choice in a haystack: Automatic extraction of opt-out statements from privacy policy text". In: *Proceedings of The Web Conference 2020*. 2020, pp. 1943–1954.

[55] Muhammad Ahmad Bashir and Christo Wilson. "Diffusion of User Tracking Data in the Online Advertising Ecosystem." In: *Proc. Priv. Enhancing Technol.* 2018.4 (2018), pp. 85–103.

[56] Bauer, A. and Hebeisen, C. *Igexin advertising network put user privacy at risk*. https://blog.lookout.com/igexin-malicious-sdk. Accessed: February 12, 2019. 2019.

[57] Richard Baumann, Mykolai Protsenko, and Tilo Müller. "Anti-proguard: Towards automated deobfuscation of android apps". In: *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*. 2017, pp. 7–12.

[58] BBC News. *Web porn: Just how much is there?* https://www.bbc.com/news/technology-23030090. 2013.

[59] Felix Beierle et al. "TYDR–Track your daily routine. Android App for tracking smartphone sensor and usage data". In: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2018, pp. 72–75.

[60] Eduardo Blázquez et al. "Trouble Over-The-Air: An Analysis of FOTA Apps in the Android Ecosystem". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1606–1622.

[61] K. Block, S. Narain, and G. Noubir. "An autonomic and permissionless Android covert channel". In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2017, pp. 184–194.

[62] Theodore Book, Adam Pridgen, and Dan S Wallach. "Longitudinal analysis of android ad library permissions". In: *arXiv preprint arXiv:1303.0857* (2013).

[63] Boomerang. *Spin Browser*. https://useboomerang.com/spin/.

[64] Branch.io. *Homepage*. https://branch.io.

[65] Branch.io. *Terms of Service*. https://branch.io/policies/#terms-and-conditionss.

[66] Braze. *Android SDK Integration*. https://www.braze.com/docs/developer_guide/platform_integration_guides/android/initial_sdk_setup/android_sdk_integration/. (accessed 2022-07-20).

[67] Braze (formerly AppBoy). *Privacy*. https://www.braze.com/privacy/.

[68] *Build location-aware apps*. Accessed July 22nd, 2022.

[69] S. Cabuk, C. E. Brodley, and C. Shields. “IP covert channel detection”. In: *ACM Transactions on Information and System Security (TISSEC)* 12.4 (2009), p. 22.

[70] Paolo Calciati and Alessandra Gorla. “How Do Apps Evolve in Their Permission Requests?: A Preliminary Study”. In: *MSR ’17*. IEEE Press, 2017.

[71] Paolo Calciati et al. “Automatically granted permissions in Android apps: An empirical study on their prevalence and on the potential threats for privacy”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 114–124.

[72] Paolo Calciati et al. “What Did Really Change with the new Release of the App?” In: *MSR 2018*. 2018.

[73] R. Chatterjee et al. “The Spyware Used in Intimate Partner Violence”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[74] Kai Chen, Peng Liu, and Yingjun Zhang. “Achieving accuracy and scalability simultaneously in detecting application clones on android markets”. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 175–186.

[75] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. “Automated test input generation for Android: Are we there yet?” In: *arXiv preprint arXiv:1503.07217* (2015).

[76] Andrea Continella et al. “Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis.” In: *NDSS*. 2017.

[77] Council of European Union. *General Data Protection Regulation 679/2016*. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN. 2016.

[78] Huajun Cui et al. “TraceDroid: A Robust Network Traffic Analysis Framework for Privacy Leakage in Android Apps”. In: *International Conference on Science of Cyber Security*. Springer. 2022, pp. 541–556.

[79] CulebraTester. *Android UI Testing Simplified*. culebra.dtmilano.com.

[80] Cyberbullying Research Center. *Summary of Our Cyberbullying Research (2004-2016)*. https://cyberbullying.org/summary-of-our-cyberbullying-research.

[81] Adrian Dabrowski et al. “Measuring cookies and web privacy in a post-gdpr world”. In: *International Conference on Passive and Active Network Measurement*. Springer. 2019, pp. 258–270.

[82] Jiarun Dai et al. “{BScout}: Direct Whole Patch Presence Test for Java Executables”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1147–1164.

[83] Jan Dekelver et al. “Design of mobile applications for people with intellectual disabilities”. In: *CREATIVITY IN INTELLIGENT TECHNOLOGIES AND DATA SCIENCE, CIT&DS 2015* 535 (2015), pp. 823–836.

[84] Luke Deshotels. “Inaudible Sound as a Covert Channel in Mobile Devices.” In: *USENIX WOOT*. 2014.

[85] Michalis Diamantaris et al. “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*. 2019, pp. 37–48.

[86] Alexander Druffel and Kris Heid. “Davinci: Android app analysis beyond frida via dynamic system call instrumentation”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2020, pp. 473–489.

[87] Matthew S Eastin, Bradley S Greenberg, and Linda Hofschire. “Parenting the internet”. In: *Journal of communication* 56.3 (2006), pp. 486–504.

[88] William Enck et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. In: *TOCS* 32.2 (2014), p. 5.

[89] Europen Comission. *Benchmarking of parental control tools for the online protection of children*. https://www.sipbench.eu/index.cfm/secid.1/secid2.3. 2017.

[90] Exodus Privacy. *Tracker*. https://reports.exodus-privacy.eu.org/en/info/trackers/. (accessed 2022-10-4).

[91] Exodus Privacy. *What Exodus Privacy does*. https://exodus-privacy.eu.org/en/page/what/.

[92] Facebook. *Get Started with Facebook for Android*. https://developers.facebook.com/docs/android/getting-started/. (accessed 2022-07-20).

[93] Facebook. *Github: Hermes*. https://github.com/facebook/hermes.

[94] Facebook Developers. *Facebook Graph API*. https://developers.facebook.com/docs/graph-api/.

[95] S. Fahl et al. “Why Eve and Mallory love Android: An analysis of Android SSL (in) security”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012.

[96] Ming Fan et al.
"An empirical evaluation of GDPR compliance violations in Android mHealth apps".
In: *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE. 2020, pp. 253–264.

[97] Shehroze Farooqi et al.
"Understanding incentivized mobile app installs on google play store".
In: *Proceedings of the ACM internet measurement conference*. 2020, pp. 696–709.

[98] Parvez Faruki et al.
"Evaluation of android anti-malware techniques against dalvik bytecode obfuscation".
In: *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE. 2014, pp. 414–421.

[99] Álvaro Feal et al. "Angel or devil? a privacy study of mobile parental control apps".
In: *Proceedings on Privacy Enhancing Technologies* 2020.2 (2020), pp. 314–335.

[100] ÁLVARO FEAL et al.
"Don't Accept Candy from Strangers: An Analysis of Third-Party Mobile SDKs".
In: *Data Protection and Privacy: Data Protection and Artificial Intelligence* (2021), p. 1.

[101] Federal Trade Comission.
Children's Online Privacy Protection Act, (15 U.S.C. 6501, et seq.,)
https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule. 1998.

[102] Federal Trade Comission.
Get Parents' Verifiable Consent Before Collecting Personal Information rom Their Kids.
https://www.ftc.gov/tips-advice/business-center/guidance/childrens-online-privacy-protection-rule-six-step-compliance#step4.

[103] Federal Trade Comission. *Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission*.
https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked. Accessed: 12-01-2021.

[104] Federal Trade Comission. *Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission*.
https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked. 2016.

[105] Federal Trade Comission. *Video Social Networking App Musical.ly Agrees to Settle FTC Allegations Th at it Violated Children ' s Privacy Law '*.
www.ftc.gov/news-events/press-releases/2019/02/video-social-networking-app-musically-agrees-settle-ftc. 2016.

[106] Adrienne Porter Felt et al. "Android permissions demystified".
In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 627–638.

[107] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the eighth symposium on usable privacy and security*. 2012, pp. 1–14.

[108] Firebase. *Add Firebase to your Android project*. https://firebase.google.com/docs/android/setup. (accessed 2022-07-20).

[109] Firebase. *Firebase A/B Testing*. https://firebase.google.com/docs/ab-testing.

[110] Firebase. *Firebase Clould Messaging'*. https://firebase.google.com/docs/cloud-messaging.

[111] Firebase. *Firebase Craslytics*. https://firebase.google.com/docs/crashlytics.

[112] *Firebase products*. Accessed June 7th, 2022.

[113] Fortumo. *Global direct carrier billing platform*. https://fortumo.com/.

[114] Yanick Fratantonio et al. "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop". In: *S&P*. 2017.

[115] J. Gamba et al. "An Analysis of Pre-installed Android Software". In: *S&P*. 2020.

[116] Julien Gamba et al. "An analysis of pre-installed android software". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1039–1055.

[117] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. "Show Me How You Move and I Will Tell You Who You Are". In: *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS*. SPRINGL '10. ACM, 2010.

[118] Yuhao Gao et al. "Demystifying illegal mobile gambling apps". In: *Proceedings of the Web Conference 2021*. 2021, pp. 1447–1458.

[119] GDPR. *Certification - Article 42*. www.privacy-regulation.eu/en/article-42-certification-GDPR.htm.

[120] GDPR Info. *GDPR Article 32*. URL: https://gdpr-info.eu/art-32-gdpr/.

[121] GDPR.eu. *What are GDPR fines?* https://gdpr.eu/fines/.

[122] GDPR.eu. *What are GDPR fines?* https://gdpr.eu/fines/.

[123] M. Georgiev et al. "The most dangerous code in the world: validating SSL certificates in non-browser software". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 38–49.

[124] C. Gibler et al. “AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale”.
In: *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*. TRUST’12.
Vienna, Austria, 2012. URL: http://dx.doi.org/10.1007/978-3-642-30921-2_17.

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

[126] Google. *Families*.
https://play.google.com/about/families/.

[127] Google. *Firebase*.
firebase.google.com/.

[128] Google. *Google Play certified ad networks program*.
https://support.google.com/googleplay/android-developer/answer/9283445.

[129] Google. *Provide information for Google Play’s Data safety section*.
Accesed September 6th, 2022.

[130] Google. *Providing a safe and secure experience for our users*.
https://android-developers.googleblog.com/2018/10/providing-safe-and-secure-experience.html.

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

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

[133] Google Developers. *AdMob*.
https://developers.google.com/admob.

[134] Google Developers. *Google Pay*.
https://developers.google.com/pay.

[135] Google, Inc. *Android Q privacy: Changes to data and identifiers*. https://developer.android.com/preview/privacy/data-identifiers#device-identifiers.
Accessed: June 1, 2019.

[136] Google, Inc. *Distribution Dashboard*. https://developer.android.com/about/dashboards.
Accessed: June 1, 2019. 2019.

[137] Google, Inc. *Wi-Fi Scanning Overview*.
https://developer.android.com/guide/topics/connectivity/wifi-scan#wifi-scan-permissions. Accessed: June 1, 2019.

[138] Google Maps Help. *Introducing Live View*.
https://support.google.com/maps/thread/11554255?hl=en.

[139] Google Maps Platform. *Overview*.
https://developers.google.com/maps/documentation/android-sdk/intro.

[140] Google Play. *Families*. https://play.google.com/about/privacy-security-deception/.

[141] Google Play. *Kid Control Dev profile*. https://play.google.com/store/apps/dev?id=6687539553449035845.

[142] Google Play. *Privacy, Security and Deception*. https://play.google.com/about/privacy-security-deception/.

[143] Google Play. *Yoguesh Dama profile*. https://play.google.com/store/apps/dev?id=5586168019301814022.

[144] Google Play Store — FamilySafety Production. *GPS Phone Tracker*. https://play.google.com/store/apps/details?id=com.fsp.android.c. 2018.

[145] M. I. Gordon et al. “Information Flow Analysis of Android Applications in DroidSafe.” In: *NDSS*. Vol. 15. 2015, p. 110.

[146] Michael C Grace et al. “Unsafe exposure analysis of mobile in-app advertisements”. In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. 2012, pp. 101–112.

[147] United Kingdom of Great Britain and Northern Ireland. *Mobile Telephones (Re-programming) Act*. http://www.legislation.gov.uk/ukpga/2002/31/introduction. 2002.

[148] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. “Code reuse in open source software”. In: *Management science* 54.1 (2008), pp. 180–193.

[149] Catherine Han et al. “The price is (not) right: Comparing privacy in free and paid apps”. In: *Proceedings on Privacy Enhancing Technologies* 2020.3 (2020).

[150] Hamza Harkous et al. “Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning”. In: *USENIX Security 18*. USENIX Association, 2018.

[151] Herald Sun. *Police warn photos of kids with geo-tagging being used by paedophiles*. https://www.heraldsun.com.au/technology/news/photograph-uploads-put-kids-at-risk/news-story/9ef00e4105cb1d38d8f5acb77d6c7433. 2012.

[152] here.com. *Homepage*. www.here.com/.

[153] *How to use Play Console*. Accessed July 22nd, 2022.

[154] IAPP. *GDPR Article 8*. URL: https://iapp.org/resources/article/the-eu-general-data-protection-regulation/\#A8.

[155] IAPP. *GDPR Recital 38*. https://iapp.org/resources/article/the-eu-general-data-protection-regulation/#R38.

[156] ICO. *Individual Rights*. Accessed August 24th, 2022.

[157] INCIBE. *Internet Segura For Kids (Spanish)*. https://www.is4k.es/. Accessed: 17-01-2022.

[158] Internet Safety 101. *Internnet Safety*. https://internetsafety101.org/.

[159] ioXt. *The Global Standard for IoT Security*. https://www.ioxtalliance.org/. (accessed 2022-09-14).

[160] Umar Iqbal et al. “Your Echos are Heard: Tracking, Profiling, and Ad Targeting in the Amazon Smart Speaker Ecosystem”. In: *arXiv preprint arXiv:2204.10920* (2022).

[161] IronSource. *Android SDK Integration*. https://developers.is.com/ironsource-mobile/android/android-sdk. (accessed 2022-07-20).

[162] *IronSource to acquire mobile advertising and app monetization company Tapjoy*. Accessed June 3rd, 2022.

[163] Paul Jaccard. “The distribution of the flora in the alpine zone. 1”. In: *New phytologist* 11.2 (1912), pp. 37–50.

[164] Carlos Jensen and Colin Potts. “Privacy policies as decision-making tools: an evaluation of online privacy notices”. In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 2004, pp. 471–478.

[165] Jeremi M. Gosney. *Nvidia GTX 1080 Hashcat Benchmarks*. https://gist.github.com/epixoip/6ee29d5d626bd8dfe671a2d8f188b77b. Accessed: June 1, 2019. 2016.

[166] Qiwei Jia et al. “Who leaks my privacy: Towards automatic and association detection with gdpr compliance”. In: *International Conference on Wireless Algorithms, Systems, and Applications*. Springer. 2019, pp. 137–148.

[167] Yufei Jiang et al. “RedDroid: Android application redundancy customization based on static analysis”. In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2018, pp. 189–199.

[168] Haojian Jin et al. “Why are they collecting my data? inferring the purposes of network traffic in mobile apps”. In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2.4 (2018), pp. 1–27.

[169] jPush. *Product Introduction*. https://docs.jiguang.cn/en/jpush/guideline/intro/.

[170] Ari Juels. “Targeted advertising... and privacy too”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2001, pp. 408–424.

[171] Kiddoware. *Kiddoware homepage*. https://kiddoware.com/.

[172] P. Kocher, J. Jaffe, and B. Jun. “Differential power analysis”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[173] Konrad Kollnig et al. "Are iPhones Really Better for Privacy? Comparative Study of iOS and Android Apps". In: *arXiv preprint arXiv:2109.13722* (2021).

[174] Konrad Kollnig et al. "Before and after GDPR: tracking in mobile apps". In: *arXiv preprint arXiv:2112.11117* (2021).

[175] Butler W Lampson. "A note on the confinement problem". In: *Communications of the ACM* (1973).

[176] Pierre Laperdrix et al. "The Price to Play: a Privacy Analysis of Free and Paid Games in the Android Ecosystem". In: *Proceedings of the ACM Web Conference 2022*. 2022, pp. 3440–3449.

[177] Douglas J Leith. "Mobile Handset Privacy: Measuring The Data iOS and Android Send to Apple And Google". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2021, pp. 231–251.

[178] Li Li et al. "Iccta: Detecting inter-component privacy leaks in android apps". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 280–291.

[179] Commission Nationale de l'Informatique et des Libertés (CNIL). *Data Protection Around the World*. https://www.cnil.fr/en/data-protection-around-the-world. Accessed: September 23, 2018. 2018.

[180] Commission Nationale de l'Informatique et des Libertés (CNIL). *The CNIL's restricted committee imposes a financial penalty of 50 Million euros against Google LLC*. 2019.

[181] Martina Lindorfer et al. "Andrubis–1,000,000 apps later: A view on current Android malware behaviors". In: *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE. 2014, pp. 3–17.

[182] M. Liu et al. "Identifying and Analyzing the Privacy of Apps for Kids". In: *Proc. of ACM HotMobile*. 2016.

[183] Xing Liu et al. "Privacy risk analysis and mitigation of analytics libraries in the android ecosystem". In: *IEEE Transactions on Mobile Computing* 19.5 (2019), pp. 1184–1199.

[184] Sonia Livingstone et al. "Risks and safety for children on the internet: the UK report". In: *Politics* 6.1 (2010).

[185] Sonia Livingstone et al. "Risks and Safety on the Internet: The Perspective of European Children. Full FINDINGS". In: (Jan. 2011).

[186] Livingstone, Sonia and Helsper, Ellen. "Parental Mediation of Children's Internet Use". In: *Journal of Broadcasting & Electronic Media - J BROADCAST ELECTRON MEDIA* 52 (Nov. 2008), pp. 581–599. DOI: 10.1080/08838150802437396.

[187] Ziang Ma et al. “Libradar: fast and accurate detection of third-party libraries in android apps”. In: *Proceedings of the 38th international conference on software engineering companion*. 2016, pp. 653–656.

[188] Ziang Ma et al.
“LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps”.
In: *ICSE 2016*. ACM, 2016.

[189] Mary Madden et al. “Teens, social media, and privacy”.
In: *Pew Research Center* 21 (2013), pp. 2–86.

[190] D. Maiorca et al.
“Stealth attacks: An extended insight into the obfuscation effects on android malware”.
In: *Computers & Security* 51 (2015), pp. 16–31.

[191] Claudio Marforio et al.
“Analysis of the communication between colluding applications on modern smartphones”.
In: *Proceedings of the 28th Annual Computer Security Applications Conference*. 2012, pp. 51–60.

[192] Kay Mathiesen. “The Internet, children, and privacy: the case against parental monitoring”.
In: *Ethics and Information Technology* 15.4 (2013), pp. 263–274.

[193] Célestin Matte, Nataliia Bielova, and Cristiana Santos.
“Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe’s transparency and consent framework”.
In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 791–809.

[194] Y. Michalevsky et al. “PowerSpy: Location Tracking Using Mobile Device Power Analysis.”
In: *USENIX Security Symposium*. 2015, pp. 785–800.

[195] Yan Michalevsky, Dan Boneh, and Gabi Nakibly.
“Gyrophoone: Recognizing speech from gyroscope signals”.
In: *23rd {USENIX$}$ Security Symposium (${$USENIX$}$ Security 14)*. 2014, pp. 1053–1067.

[196] Nariman Mirzaei et al. “Reducing combinatorics in GUI testing of android applications”.
In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 559–570.

[197] Hooman Mohajeri Moghaddam et al.
“Watching you watch: The tracking ecosystem of over-the-top tv streaming devices”.
In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 131–147.

[198] Israel J Mojica et al. “A large-scale empirical study on software reuse in mobile apps”.
In: *IEEE software* 31.2 (2013), pp. 78–86.

[199] Monica Anderson. *Parents, Teens and Digital Monitoring*.
https://stirlab.org/wp-content/uploads/2018/06/2017_Wisniewski_ParentalControl.pdf.

[200] MoPub. *Powerful App Monetization*. www.mopub.com/.

[201] K. Moran et al. "CrashScope: A Practical Tool for Automated Testing of Android Applications". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017.

[202] New York Times. *Uber Hid 2016 Breach, Paying Hackers to Delete Stolen Data*. https://www.nytimes.com/2017/11/21/technology/uber-hack.html. 2017.

[203] Yi Ying Ng et al. "Which Android app store can be trusted in China?" In: *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE. 2014, pp. 509–518.

[204] Duc Cuong Nguyen et al. "Short text, large effect: Measuring the impact of user reviews on android app security & privacy". In: *2019 IEEE symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 555–569.

[205] Duc Cuong Nguyen et al. "Up2dep: Android tool support to fix insecure code dependencies". In: *Annual Computer Security Applications Conference*. 2020, pp. 263–276.

[206] L. Nguyen et al. "Unlocin: Unauthorized location inference on smartphones without being caught". In: *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*. IEEE. 2013, pp. 1–8.

[207] Trung Tin Nguyen et al. "Measuring user perception for detecting unexpected access to sensitive resource in mobile apps". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 578–592.

[208] Trung Tin Nguyen et al. "Share First, Ask Later (or Never?) Studying Violations of {GDPR's} Explicit Consent in Android Apps". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 3667–3684.

[209] Helen Nissenbaum. "Privacy as contextual integrity". In: *Washington Law Review* (2004).

[210] Midas Nouwens et al. "Dark patterns after the GDPR: Scraping consent pop-ups and demonstrating their influence". In: *Proceedings of the 2020 CHI conference on human factors in computing systems*. 2020, pp. 1–13.

[211] Ofcom: UK broadband, home phone and mobile services regulator. *Children and parents: Media use and attitudes report 2018*. https://www.ofcom.org.uk/__data/assets/pdf_file/0024/134907/Children-and-Parents-Media-Use-and-Attitudes-2018.pdf. 2018.

[212] Ehimare Okoyomon et al. "On the ridiculousness of notice and consent: Contradictions in app privacy policies". In: *Workshop on Technology and Consumer Protection (ConPro 2019), in conjunction with the 39th IEEE Symposium on Security and Privacy*. 2019.

[213] Marten Oltrogge et al.
"The rise of the citizen developer: Assessing the security impact of online app generators".
In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 634–647.

[214] OpenX. *Why We Exist*. https://www.openx.com/company/. 2019.

[215] Elleen Pan et al.
"Panoptispy: Characterizing Audio and Video Exfiltration from Android Applications".
In: *Proceedings on Privacy Enhancing Technologies* 2018.4 (2018), pp. 33–50.

[216] Harshvardhan J Pandit, Declan O'Sullivan, and Dave Lewis.
Queryable Provenance Metadata For GDPR Compliance. 2018.

[217] Elias P Papadopoulos et al.
"The long-standing privacy debate: Mobile websites vs mobile apps".
In: *Proceedings of the 26th International Conference on World Wide Web*. 2017, pp. 153–162.

[218] Muhammad Talha Paracha et al.
"IoTLS: understanding TLS usage in consumer IoT devices".
In: *Proceedings of the 21st ACM Internet Measurement Conference*. 2021, pp. 165–178.

[219] Elkana Pariwono et al. "Don't throw me away: Threats Caused by the Abandoned Internet Resources Used by Android Apps".
In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 147–158.

[220] PCMag. *The Best Parental Control Software of 2019*.
https://uk.pcmag.com/parental-control-monitoring/67305/the-best-parental-control-software.

[221] Phunware. *Android SDK Integration Guide - Core*.
https://developer.phunware.com/display/DD/Android+SDK+Integration+Guide+-+Core. (accessed 2022-07-20).

[222] Sundar Pichai. *Privacy Should Not Be a Luxury Good*. The New York Times.
https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html. 2019.

[223] Michael Plotnick, Charles Eldering, and Douglas Ryder. *Behavioral targeted advertising*. US Patent App. 10/116,692. 2002.

[224] I. Poese et al. "IP geolocation databases: Unreliable?"
In: *ACM SIGCOMM Computer Communication Review* 41.2 (2011), pp. 53–56.

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

[226] Exodus Privacy. *Trackers*. Accesed March 2nd, 2022.

[227] A. Rao et al. "Meddle: middleboxes for increased transparency and control of mobile traffic".
In: *Proceedings of the 2012 ACM conference on CoNEXT student workshop*. ACM. 2012, pp. 65–66.

[228] Ratcliff, John and Metzener, David. *Pattern Matching: the Gestalt Approach*. https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5. (accessed 2020-03-23).

[229] Abbas Razaghpanah et al. "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem". In: *The 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. 2018.

[230] Abbas Razaghpanah et al. "Apps, Trackers, Privacy and Regulators: A Global Study of the Mobile Tracking Ecosystem". In: *Network and Distributed System Security Symposium*. Feb. 2018.

[231] Abbas Razaghpanah et al. "Haystack: In Situ Mobile Traffic Analysis in User Space". In: *CoRR* (2015).

[232] Abbas Razaghpanah et al. "Studying TLS usage in Android apps". In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM. 2017, pp. 350–362.

[233] Joel Reardon et al. "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System". In: *28th {USENIX$}$ Security Symposium (${$USENIX$}$ Security 19)*. 2019, pp. 603–620.

[234] Nilo Redini et al. "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 484–500.

[235] J. Ren et al. "ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic". In: *Proceedings of the ACM SIGMOBILE MobiSys*. 2016.

[236] Jingjing Ren et al. "Bug Fixes, Improvements,... and Privacy Leaks". In: *NDSS* (2018).

[237] Reuters. *Facebook loses Belgian privacy case, faces fine of up to $ 125 million*. www.reuters.com/article/us-facebook-belgium/facebook-loses-belgian-privacy-case-faces-fine-of-up-to-125-million-idUSKCN1G01LG.

[238] Irwin Reyes et al. ""Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale". In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (2018), pp. 63–83.

[239] Israel J Mojica Ruiz et al. "Understanding reuse in the android market". In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE. 2012, pp. 113–122.

[240] SafeWise. *Best Parental Control Apps and Software Buyers Guide*. https://www.safewise.com/resources/parental-control-filters-buyers-guide/.

[241] Said Jawad Saidi et al. "A haystack full of needles: Scalable detection of iot devices in the wild". In: *Proceedings of the ACM Internet Measurement Conference*. 2020, pp. 87–100.

[242] Salmonads. *About Us*. http://publisher.salmonads.com. 2016.

[243] Samsung. *Knox SDK*. https://seap.samsung.com/sdk/knox-android.

[244] G. Sarwar et al. “On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices.” In: *SECRYPT*. Vol. 96435. 2013.

[245] Sarah Schafer. *With Capital in Panic, Pizza Deliveries Soar*. The Washington Post. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm. 1998.

[246] Christian Schindler et al. “Privacy leak identification in third-party Android libraries”. In: *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*. IEEE. 2022, pp. 1–6.

[247] R. Schlegel et al. “Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.” In: *NDSS*. Vol. 11. 2011, pp. 17–33.

[248] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP ’10. IEEE Computer Society, 2010.

[249] Screentime Labs. *Screentime homepage*. https://screentimelabs.com/.

[250] James Sellwood and Jason Crampton. “Sleeping android: The danger of dormant permissions”. In: *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*. 2013, pp. 55–66.

[251] 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*. 2015, pp. 1–6.

[252] Shashi Shekhar, Michael Dietz, and Dan S Wallach. *Separating Smartphone advertising from applications*. Rice University, 2012.

[253] Laura Shipp and Jorge Blasco. “How private is your period?: A systematic analysis of menstrual app privacy policies.” In: *Proc. Priv. Enhancing Technol.* 2020.4 (2020), pp. 491–510.

[254] Benjamin Shmueli and Ayelet Blecher-Prigat. “Privacy for children”. In: *Colum. Hum. Rts. L. Rev.* 42 (2010), p. 759.

[255] Anastasia Shuba and Athina Markopoulou. “Nomoats: Towards automatic detection of mobile tracking”. In: *Proceedings on Privacy Enhancing Technologies* 2020.2 (2020).

[256] Anastasia Shuba et al. “Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices”. In: *Workshop on Wireless of the Students, by the Students, & for the Students*. ACM. 2015.

[257] Laurent Simon, Wenduan Xu, and Ross Anderson. “Don’t interrupt me while I type: Inferring text entered through gesture typing on Android keyboards”. In: (2016).

[258] Rocky Slavin et al.
"Toward a framework for detecting privacy policy violations in android application code".
In: *International Conference on Software Engineering*. ACM. 2016.

[259] R. Spreitzer et al.
"Systematic classification of side-channel attacks: a case study for mobile devices".
In: *IEEE Communications Surveys & Tutorials* (2017).

[260] StartApp. *Mobile, Fulfilled*.
www.startapp.com/.

[261] State of California Department of Justice. *California Consumer Privacy Act (CCPA)*.
https://oag.ca.gov/privacy/ccpa. Accessed: 18-08-2022.

[262] Statista. *Mobile Internet*.
https://www.statista.com/topics/779/mobile-internet/. 2018.

[263] Ryan Stevens et al. "Investigating user privacy in android ad libraries".
In: *Workshop on Mobile Security Technologies (MoST)*. Vol. 10. Citeseer. 2012,
pp. 195–197.

[264] COUNTY OF LOS ANGELES SUPERIOR COURT OF THE STATE OF CALIFORNIA.
COMPLAINT FOR INJUNCTIVE RELIEF AND CIVIL PENALTIES FOR VIOLATIONS OF THE UNFAIR COMPETITION LAW. http://src.bna.com/EqH. 2019.

[265] Vincent F. Taylor and Ivan Martinovic.
"To Updae or Not to Update: Insights From a Two-Year Study of Android App Evolution".
In: *ASIA CCS '17*. ACM, 2017.

[266] The Guardian. *Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach*.
www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election.

[267] The New York Times. *Time to Build a National Data Broker Registry*.
www.nytimes.com/2019/09/13/opinion/data-broker-registry-privacy.html.

[268] The Verge. *There are over 3 billion active Android devices*.
https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. Accessed: 28-12-2021.

[269] HTTP ToolKit. *Android 11 tightens restrictions on CA certificates*.
Accesed September 1st, 2022.

[270] L. Tsai et al. "Turtle Guard: Helping Android Users Apply Contextual Privacy Preferences".
In: *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*.
USENIX Association, 2017.

[271] Güliz Seray Tuncay et al. "Resolving the predicament of android custom permissions".
In: (2018).

[272] Twitter4J. *Homepage*.
https://developers.facebook.com/docs/graph-api/.

[273] Unity. *Monetize your game*. https://unity.com/solutions/unity-ads.

[274] Unity. *Unity for all*. unity.com.

[275] Unity Ads. *Installing the Unity Ads SDK for Android*. https://docs.unity.com/ads/InstallingTheAndroidSDK.html. (accessed 2022-07-20).

[276] U.S. Federal Trade Commission. *In the Matter of HTC America, Inc.* https://www.ftc.gov/sites/default/files/documents/cases/2013/07/130702htcdo.pdf. 2013.

[277] U.S. Federal Trade Commission. *In the Matter of Turn Inc.* https://www.ftc.gov/system/files/documents/cases/152_3099_c4612_turn_complaint.pdf. 2017.

[278] U.S. Federal Trade Commission. *Mobile Security Updates: Understanding the Issues*. https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf. 2018.

[279] U.S. Federal Trade Commission. *The Federal Trade Commission Act. (FTC Act)*. https://www.ftc.gov/enforcement/statutes/federal-trade-commission-act.

[280] Junia Valente and Alvaro A. Cardenas. "Security &#38; Privacy in Smart Toys". In: *IoTS&#38;P '17*. ACM, 2017.

[281] Raja Vallée-Rai et al. "Soot – a Java Bytecode Optimization Framework". In: *CASCON*. IBM Press, 1999.

[282] Pelayo Vallina et al. "Tales from the porn: A comprehensive privacy analysis of the web porn ecosystem". In: *Proceedings of the Internet Measurement Conference*. 2019, pp. 245–258.

[283] Janus Varmarken et al. "The tv is smart and full of trackers: Measuring smart tv advertising and tracking". In: *Proceedings on Privacy Enhancing Technologies* 2020.2 (2020).

[284] Luca Verderame et al. "On the (un) reliability of privacy policies in android apps". In: *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2020, pp. 1–9.

[285] Timothy Vidas and Nicolas Christin. "Evading android runtime analysis via sandbox detection". In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 2014, pp. 447–458.

[286] Haoyu Wang et al. "Beyond google play: A large-scale comparative study of chinese android app markets". In: *Proceedings of the Internet Measurement Conference 2018*. 2018, pp. 293–307.

[287] Haoyu Wang et al. "Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets". In: *IMC '18*. ACM, 2018.

[288] Wenyu Wang et al. “An empirical study of android test generation tools in industrial cases”. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, pp. 738–748.

[289] Fengguo Wei, Sankardas Roy, and Xinming Ou. “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps”. In: *ACM Transactions on Privacy and Security (TOPS)* 21.3 (2018), pp. 1–32.

[290] Xuetao Wei et al. “Permission Evolution in the Android Ecosystem”. In: *ACSAC ’12*. ACM, 2012.

[291] Lukas Weichselbaum et al. “Andrubis: Android malware under the magnifying glass”. In: *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001* (2014), pp. 1–10.

[292] Dominik Wermke et al. “A large scale investigation of obfuscation use in google play”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 222–235.

[293] P. Wijesekera et al. “Android Permissions Remystified: A Field Study on Contextual Integrity”. In: *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 2015.

[294] P. Wijesekera et al. “The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017.

[295] Shomir Wilson et al. “The creation and analysis of a website privacy policy corpus”. In: *Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016.

[296] Pamela Wisniewski et al. “Parental Control vs. Teen Self-Regulation: Is there a middle ground for mobile online safety?” In: *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. ACM. 2017, pp. 51–69.

[297] Benjamin Yankson, Farkhund Iqbal, and Patrick C. K. Hung. “Privacy Preservation Framework for Smart Connected Toys”. In: *Computing in Smart Toys*. Ed. by Jeff K.T. Tang and Patrick C. K. Hung. Springer International Publishing, 2017.

[298] Michele L Ybarra, Kimberly J Mitchell, and Josephine D Korchmaros. “National trends in exposure to and experiences of violence on the Internet among children”. In: *Pediatrics* (2011).

[299] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. “Real-time bidding for online advertising: measurement and analysis”. In: *Proceedings of the seventh international workshop on data mining for online advertising*. 2013, pp. 1–8.

[300] Xian Zhan et al. “Automated third-party library detection for android applications: Are we there yet?” In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2020, pp. 919–930.

[301] Xian Zhan et al. “Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review”. In: *IEEE Transactions on Software Engineering* (2021).

[302] J. Zhang, A. R. Beresford, and I. Sheret.
"SensorID: Sensor Calibration Fingerprinting for Smartphones".
In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[303] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann.
"Libid: reliable identification of obfuscated third-party android libraries". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 55–65.

[304] Junbin Zhang et al.
"Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe".
In: *IEEE Transactions on Software Engineering* (2021).

[305] Yuan Zhang et al.
"Detecting third-party libraries in android applications with high precision and recall".
In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 141–152.

[306] Nan Zhong and Florian Michahelles. "Google Play is Not a Long Tail Market: An Empirical Analysis of App Adoption on the Google Play App Market". In: *SAC*. ACM, 2013.

[307] X. Zhou et al.
"Identity, location, disease and more: Inferring your secrets from android public resources".
In: *Proc. of 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM. 2013.

[308] Tong Zhu et al. "Dissecting Click Fraud Autonomy in the Wild". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 271–286.

[309] Onur Zungur, Gianluca Stringhini, and Manuel Egele.
"Libspector: Context-aware large-scale network traffic analysis of android applications".
In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2020, pp. 318–330.