Informatics

Dynamic iOS Privacy Analysis: Verifying App Store Privacy Labels

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Wolfgang Jirout, BSc
Matrikelnummer 01525606

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.techn. Martina Lindorfer, BSc

Signiert von:	Thomas Jirout
Datum:	07.12.2021 17:16:37

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

www.a-trust.at
TRUST einfach sicher

Signiert von:	Martina Lindorfer
Datum:	07.12.2021 16:54:37

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

www.a-trust.at
TRUST einfach sicher

Wien, 7. Dezember 2021

Thomas Wolfgang Jirout   Martina Lindorfer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Dynamic iOS Privacy Analysis: Verifying App Store Privacy Labels

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Wolfgang Jirout, BSc

Registration Number 01525606

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.techn. Martina Lindorfer, BSc

Vienna, 7th December, 2021

Thomas Wolfgang Jirout

Martina Lindorfer

Technische Universität Wien

A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Thomas Wolfgang Jirout, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. Dezember 2021

Thomas Wolfgang Jirout

Danksagung

Danke Michael, meinen Freunden, meinen Eltern und baldigen Schwiegereltern für eure kontinuierliche Unterstützung, die Gespräche, den nötigen Freiraum und die Motivation, meinen Weg beizubehalten. Ohne gestern kein heute.

Kurzfassung

Smartphones speichern große Mengen persönlicher Informationen, wie Fotos, Kontakte oder Standort-Daten, an einem zentralen Ort. Nutzer*innen können den Applikationen ("Apps") auf ihren Geräten Zugriff zu diesen Informationen gestatten, jedoch gibt es danach kaum eine Möglichkeit um festzustellen, *wie* die Daten von der App verwendet werden. Mit dem Ziel, diese Nutzung für die Anwendenden transparenter zu gestalten, hat Apple kürzlich die sogenannten App Privatsphäre Details ("Privatsphäre-Angaben") im iOS App Store eingeführt. Seither sind die Anbieter*innen von Apps verpflichtet, in einem standardisierten Format anzugeben, wie sie oder Dritte die von ihnen gesammelten Daten speichern und verwenden. Eine große Schwachstelle dieses Systems ist es, dass diese Angaben seitens Apple nicht überprüft werden.

Ziel dieser Arbeit war es deshalb, zu evaluieren, wie dynamische Analysemethoden dazu verwendet werden können, um das Daten-Nutzungsverhalten von Apps zu analysieren und die Privatsphäre-Angaben zu verifizieren.

Zu diesem Zweck wurde eine automatisierte Analyse-Plattform entwickelt, die mittels dynamischer Instrumentation während der Laufzeit einer App sensible Methodenaufrufe analysieren und den Netzwerk-Datenverkehr auswerten kann. Durch die Automatisierung dieses Prozesses wird es ermöglicht, eine solche Privatsphäre-Untersuchung an einer größeren Menge von Apps durchzuführen.

Die Performanz der Plattform wurde im Zuge von drei Fallstudien mit insgesamt 120 Apps bewertet. Hierbei lag der Fokus darauf, Missbrauch der "Geräte-ID"-Angabe im App Store zu identifizieren. Diese Angabe muss von App-Anbieter*innen gemacht werden, wenn ihre App auf die Werbe-ID zugreift und diese sendet, da sie dadurch die Möglichkeit erhalten, Nutzer*innen-Verhalten app-übergreifend zu analysieren. Unsere Ergebnisse zeigen, dass die Plattform eine solche Deklaration in 70% der Fälle verifizieren konnte. Zusätzlich konnten wir feststellen, dass 24% einer Gruppe bestehend aus 50 Apps, die explizit angegeben hatten gar keine Daten zu sammeln, es dennoch taten.

Unsere Erkenntnisse legen nahe, dass die Einführung der App Privatsphäre Details ein effektiver Schritt ist, um die Verwendung von sensitiven Daten transparenter zu machen. Jedoch zeigen sie auch, dass deren Überprüfung notwendig ist, um sicherzustellen, dass Nutzer*innen langfristig nicht durch falsche oder unverifizierte Angaben ihr vertrauen darin verlieren.

Abstract

Smartphones bundle large amounts of our personal data, like photos, contact information, and location data, in a single place. Users can grant applications (“apps”) access to this data, but they can hardly control or verify *how* the data is used afterwards. In an attempt to make the collection of privacy-sensitive data more transparent to the user, Apple has recently introduced so-called App Privacy Details (“privacy labels”) on the App Store. They require iOS apps to disclose which kinds of data types they or their third-party partners collect and how they store and use it. One major shortcoming of the system is that these declarations are not verified or audited by Apple, opening the possibility for potential misuse.

The aim of this thesis was to evaluate how dynamic mobile analysis methods can be used to investigate an app’s privacy-related behavior at run time in an automated manner, and how they could be used to verify declared privacy labels.

Using Dynamic Instrumentation with API Method Call Interception as well as Network Analysis, we created an analysis platform capable of conducting such an analysis on a larger number of apps in an automated manner with minimal interaction by the researcher.

To evaluate the real-world performance of the platform, we used it to analyze the privacy behavior of 120 apps in three different case studies. Specifically, we focused on identifying misuse of the “Device ID” label, which apps need to declare if they collect the device’s Advertising Identifier, a valuable data point that allows tracking services to cross-reference user data across apps. Our results show that our platform could successfully verify a declared privacy label for the Advertising Identifier in 70% of cases. Furthermore, we also discovered that among 50 apps who declared not to collect any data, 24% of them still did.

We conclude that privacy labels are an important step towards making the use of sensitive data more transparent to the user, but more work needs to be done to ensure that users do not lose their trust in them caused by inaccurate and unvalidated declarations.

Contents

CHAPTER 1

Introduction

Smartphones store large amounts of personal information about ourselves and others. Sensitive information, such as location data, contacts, photos, messages, etc., is stored in a single place and thus requires protection. Modern mobile operating systems, such as iOS or Android, use permission systems to manage access to these kinds of information [HRE+19, FHE+12]. In practice, this means that installed applications ("apps") can only access certain private information if the user has granted access to it.

However, this system has two important tradeoffs: First, after the user has allowed an app to access certain private data, every piece of code included in the application can access the sensitive information, too [HRE+19]. This especially includes all third-party libraries and frameworks that might have been included by the developer. As a result, if a user granted a navigation app access to location data, any analytics or advertising library would get access to this sensitive information as well. Second, even though users can manage access to their data through the permission system, they have hardly any means to detect, measure or control *how* the data is used afterwards [RRL+16]. Following our previous example, an analytics library would thus be able to bundle together any private data it has access to through its host app and send it to a tracking server, without any knowledge or possible control by the user.

In an effort to give users more insights into *how* their data is used by app providers, Apple has recently introduced so-called App Privacy Details[1] ("privacy labels") on the App Store. For every published or updated app on the App Store after December 8, 2020 [Cam20], developers are now required to specify which kinds of personal information are collected,[2] how the data is used, whether it is stored in a user-identifiable manner as well

[1] App Store Privacy Details, https://developer.apple.com/app-store/app-privacy-details/ (accessed Nov. 9, 2021)

[2] "Collecting" roughly refers to transmitting and storing user data off the device; Exact definition of "collect" provided on the Apple website at: https://developer.apple.com/app-store/app-privacy-details/#data-collection (accessed Nov. 9, 2021)

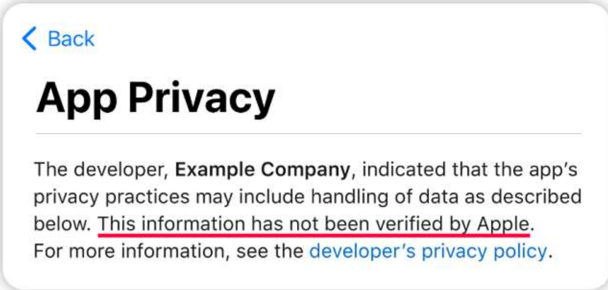

Figure 1.1: When viewing an app's Privacy Details on the App Store, Apple shows a small disclaimer clarifying that the declared labels are not verified – a major shortcoming of this system.

as whether it is used for tracking.[3] For example, an app may specify that it is collecting the user's *phone number* for *Product Personalization*, stores this data *linked to the user* but *does not use it for tracking*.

Unfortunately, one of the shortcomings of these new privacy labels is that the information provided by the developers is not verified or audited by Apple [Dor21] (Figure 1.1). Multiple recent news reports have already analyzed the plausibility of some apps' published labels and found many of them to be questionable or inaccurate [Dor21, Fow21].

Given this absence of transparency, it is important to develop research tools to analyze an app's privacy behavior and verify declared labels. Since App Privacy Details have only been introduced quite recently, there is, to the best of our knowledge, no scientific literature available on the verification of them, yet. However, the core concepts of such an analysis, namely static and dynamic analysis are commonly applied to perform security and privacy analyses on the Android and iOS platforms. Static analysis involves extracting app information without executing it, e.g., by decompiling the binary and inspecting the source code, while dynamic analysis involves evaluating an app's behavior during run time [KSB$^+$21], e.g., by analyzing network traffic while interacting with the app.

Overall, the goals of this thesis were twofold: (1) Investigate how dynamic analysis methods could be used to analyze an app's privacy-behavior at run time and verify an app's declared privacy labels. (2) Develop a workflow that could perform such an analysis in an automated manner. Due to the large number of privacy labels to be verified, we focused our attention on verifying or detecting misuse of the Advertising Identifier, a valuable unique identifier that has shown to be of high interest for apps

[3]"Tracking" roughly refers to linking collected user data with third-party data; Exact definition of "tracking" on the Apple website at: https://developer.apple.com/app-store/app-privacy-details/#user-tracking (accessed Nov. 9, 2021)

[AH13, RRL+16, KSB+21], since it allows them to cross-reference user data across apps (see also Section 3.1). Apps collecting it need to declare the "Device ID" privacy label.[4]

The main contribution of this thesis is a platform capable of performing Dynamic Instrumentation with API Method Invocation Interception to determine the privacy-behavior of apps behind the scenes, UI automation to simulate user input and network analysis to detect sent privacy-sensitive data (limited to the Advertising Identifier for this study). All of these tasks, including app installation, uninstallation, reporting and logging of analysis files are being executed autonomously, allowing researchers to investigate larger number of apps in sequence without the need for manual interaction during the analysis. Furthermore, this work contributes to an ongoing larger project which aims to develop a comparative privacy analysis toolchain for both iOS and Android.

The remainder of this document is structured as follows: In Chapter 2 we present related work to our study. Chapter 3 goes on to introduce the fundamentals of smartphone application analysis and the challenges of such an analysis on the iOS platform. Our methodology is presented in two chapters: Chapter 4 introduces the individual methods used, while Chapter 5 goes into detail about how we combined them to create an autonomous analysis platform. The results of three case studies conducted to evaluate the real-world performance of our platform are presented and discussed in Chapter 6. Finally, Chapter 7 concludes this thesis with a summary and a discussion about limitations and future work.

[4] Apple defines that the "Device ID" privacy label is to be used for identifiers "such as the device's advertising identifier, or other device-level ID": https://developer.apple.com/app-store/app-privacy-details/#data-type (accessed Nov. 10, 2021)

CHAPTER 2

Related Work

In this chapter we provide an overview of related work in the area of privacy research on mobile operating systems. While the focus of such analysis has mostly been on the Android platform in the last decade [KHRW19, OBE16], there are still a considerable amount of noteworthy publications investigating the iOS platform or performing comparative studies for both operating systems.

After the release of the iPhone in 2007 and the introduction of the App Store in 2008, one of the first studies evaluating the privacy implications of third-party iOS apps was published by Egele et al. in 2011 [EKKV11]. Specifically, they were interested to find out whether Apple's App Review, a vetting mechanism that every published app on the App Store has to pass, was effective at ensuring high privacy standards. They developed PiOS, a static analysis tool with the goal of detecting cases where private data could be leaked to third parties. By that time, they concluded that overall the 1,407 apps they investigated respected the user's privacy, but over 100 of them leaked the device's UDID, which is a valuable identifier for ad and tracking services.

One of the first comparative studies of the iOS and Android platforms regarding access to sensitive user data was published by Han et al. in 2013 [HYG+13]. Their study, which compared access to security-sensitive APIs of 1,300 cross-platform apps appearing on both the Android on iOS platform, revealed the importance of effective mechanisms implemented to protect user privacy, rather than relying on trust alone. Unlike today, where both operating systems have well-established permission systems requiring apps to ask users before they can access sensitive data, only Android had such a system by that time [HYG+13]. Apple instead relied on their vetting system and until iOS 5 simply allowed all apps to access sensitive data, without any notice to the user. As a result, the study revealed that 73% of the investigated iOS apps accessed more security-sensitive APIs than their Android counterparts, who had to explicitly ask their users beforehand.

Shortly after that, Agarwal et al. released the first large case study investigating the use of privacy-sensitive data specifically on iOS [AH13]. They developed ProtectMyPrivacy, a dynamic analysis tool that integrated into the system and then ran "in the wild" on real users' phones participating in the study. The system was capable to detect access to privacy-sensitive data and offered users a choice to grant permission or replace the information the app could see with artificial data. Overall, they gathered valuable information regarding the data apps were most interested in by analyzing 225,685 apps used by 90,621 users in the study over a period of nine months. They revealed that 48.8% of apps were interested in the device identifier, location data (13.2%) and users' contacts (6.2%).

Since then, Apple has continued to strictly narrow down the information an app can access without asking for permission and has even blocked the use of the device identifier since iOS 5 [Sch11]. However, apps could still use the Advertising Identifier instead. The only difference to the device ID is that it can be manually disabled by the user or re-generated at any time. In a recent change, however, with the release of iOS 14.5 in April 2021, Apple now requires every app to ask the user for explicit permission to use this identifier as well, a major leap for user privacy, as we will discuss later in Section 3.5 and 6.5.

A few years later, in 2015, when the App Store had already accumulated 1.3 million apps, Deng et al. investigated the quality of Apple's App Review process [DSZX15]. In their study, they created IRiS, a process capable of statically analyzing an app and detecting the use of private iOS APIs, which is prohibited by the App Store guidelines. Since all apps need to pass Apple's App Review, the researchers should not have been able to find released apps violating that rule. Their analysis of 2019 apps, however, revealed that 7% were still accessing private APIs, including ones that enabled them access to privacy-sensitive user data.

In 2016, Orikogbo et al. addressed the fact that *static* iOS analyses had so far been mostly performed on relatively small datasets, while Android studies had already evaluated hundreds of thousands of apps [OBE16]. Specifically, they claimed that the main reason for this imbalance was the fact that Android apps could easily be downloaded and evaluated, but the iOS platform's digital rights management made it challenging to aggregate a large set of target apps. To address this issue, they created CRiOS, a fully automated system capable of retrieving a large number of apps from the App Store. They used their system to download and decrypt 43,404 iOS apps, which took about four weeks. They then performed static analysis to evaluate included third-party libraries and concluded that apps on average contained only 39.8% of developer-authored content with the remaining 60.2% being library classes. Since all third-party libraries contained in an app gain the same permissions as the app itself [HRE+19], this means that included libraries, e.g., used for analytics or advertising, may leak sensitive data without the user or developer knowing about it.

Since then, papers analyzing various aspects of Android and iOS privacy leaks and user tracking have become more common [RNVR+18, BLV+18, HYG+13], which often times included results that were relevant for both platforms.

In 2018, Binns et al. evaluated the prevalence of third-party trackers by statically analyzing 959,426 Android apps and revealed that more than 90% of them contained known tracking hosts, with the median of apps including hosts of 5 distinct tracking companies [BLV+18]. Keeping this information in mind, Razaghpanah et al. analyzed the mobile advertising and tracking ecosystem using real-world usage data of 14,599 Android apps captured from study participants. They identified that Google and Facebook had the highest prevalence in apps with a presence of 73% and 31%, respectively [RNVR+18]. Most importantly, the remaining 8 out of the top 10 tracking services (present in around 80% of apps) reserved the right to share their data with other third-parties (e.g., with each other) in their privacy policy, posing a major risk to user privacy.

In order to address issues of user privacy being at risk by invisible behind-the-scenes privacy practices, Apple has recently introduced App Privacy Details ("privacy labels") on the App Store, with the aim to make the use of sensitive data more transparent. Ideas to introduce such kinds of labels have already been discussed scientifically in recent years [KBCR09, CWC+16]. For their verification, dynamic analysis methods are necessary, since static analysis alone cannot verify real data flows, given that apps are not executed [KSB+21].

Attempts to detect sensitive data leaving the device have been made over the years [RRL+16, CFL+17, SLA+16, ROF+19], but not yet with the focus of validating privacy labels, given their novelty. In 2016, Ren et al. developed a dynamic network-level analysis system capable of detecting leaks of personally identifiable information (PII) with high precision and revealed that apps are vastly interested in device identifiers (> 50% of the top 100 apps) [RRL+16]. Unfortunately, since approaches like these relied on plaintext network flow analysis and both Android and iOS apps are nowadays required to use HTTPS (encrypted traffic) [Bur18, Con16], other means of analyzing network traffic had to be investigated, e.g., using differential analysis [CFL+17] or machine-in-the-middle proxies [SLA+16, KSB+21], like we did in our study.

All things being considered, our work contributes to current research efforts by combining multiple up-to-date technologies and dynamic analysis methods into a single analysis platform. This platform is capable of performing privacy analysis on a physical iOS device autonomously and can be used to verify Apple's recently introduced App Privacy Details. It can further act as the foundation for future studies and is designed to be extendable to integrate additional analysis methods. In contrast to recent work by Kollnig et al. [KSB+21], which performs dynamic analysis by running an app for 30 seconds, our platform additionally simulates user interaction and is therefore able to trigger and accept permission requests, which allows us to investigate the app's privacy-behavior further after access was granted.

CHAPTER 3

Background

3.1 Mobile Privacy and Tracking

There are many different definitions of "privacy" [MM17], e.g., the Oxford English Dictionary defines it as the "State or condition of being free from being observed or disturbed by other people", whereas Alan F. Westin in 1967 defined it as the state of having "limited access to a consumer's information" [MM17].

Especially the latter definition is striking in today's world where smartphones are omnipresent. We place our trust in these devices and their operating systems for storing large amounts of personal information and keeping them safe. While we can mostly control what kinds of data can be accessed by the apps installed on our phones, we have little insight into how our data is processed or with whom it is shared once access has been granted.

3.1.1 Permissions and Sandboxing

Modern smartphone operating systems use sandboxing and permission systems to restrict the capabilities of an app and control which information can be accessed. Sandboxing refers to running all apps on the smartphone isolated from each other [MP15]. In general, this means that apps can not communicate with apps from different developers or access each other's files. Additionally, access to personal user data (e.g., contacts, photos, etc.) is only possible after the user has been asked for permission.

Once an app has been granted permission, all third-party libraries included in the app gain access to the unlocked sensitive data as well.

3.1.2 User Tracking

The vast majority of apps we use today include third-party libraries for advertising and/or tracking ("ATS") [RNVR+18]. Developers include these services for the purpose of getting a better understanding of their users (e.g., for analytics), to tailor the user experience or to serve (personalized) ads [BLV+18, RNVR+18]. By being present in a large number of apps, ATS can connect various kinds of user information and build a rich user profile, especially if the apps that include them contain different kinds of data access permissions. Since most of these operations happen in the background, user can be tracked across apps (or even devices) without their knowledge or consent [RNVR+18]. Given that most of the top advertising services reserve the right to share user data with other third-parties (e.g., with each other) [RNVR+18], they can gather vast amounts of information about their users without hardly any control or knowledge by them, posing a threat to user privacy.

Since apps cannot communicate with each other due to sandboxing, ATS require unique identifiers to attribute received data to a specific user on their servers. Both Android and iOS offer so-called advertising identifiers to fulfill this need. Specifically, every app can request this identifier from the operating system, which resolves to the same value for all apps installed on the system. After obtaining it, the ATS can send it to their servers, along with data they have collected (e.g., product interactions, preferences, or device information). Once they receive the same advertising identifier from two or more apps, they can use it to cross-reference the information and attribute it to one user profile.

Android and iOS offer these identifiers to make it easy for advertisers to create user profiles, while also giving users a choice to re-generate or opt-out from providing the identifier. Since iOS 14.5, Apple has also introduced App Tracking Transparency to make use of this identifier more transparent and opt-in, as detailed later in Section 3.5.

3.2 iOS Overview

3.2.1 App Structure Overview

The only official source for installing iOS apps is the App Store. Each app is delivered as an `.ipa` file [TTX+20], which contains the "Application Bundle". The executable contained in the bundle is signed and encrypted using Apple's FairPlay digital rights management system [KSB+21].

Once the app is executed, it gets decrypted in-memory by the system and two dedicated folders for the app are generated: one for the application bundle and one for the app's data [TTX+20]. Both folders have a random name and, following the sandboxing approach described above, apps can *only* directly access and manage their own folder.

The application bundle also includes the `info.plist` file, a key-value storage containing important metadata, such as the app's displayed name, version, build number and desired permissions.

3.2.2 App Permissions and Entitlements

During run time, apps can request various permissions from the iOS operating system in order to gain access to, e.g., the photo library, the user's contacts, or location data. Before an app is able to so, it has to "declare" the desired permission in the `info.plist` file by providing a *Usage Description*. The usage description contains a string with information that will be presented to the user once the app requests access, e.g., a photo editing app may have the `info.plist` key `NSPhotoLibraryUsageDescription` with the value "Access to your photo library is required to edit and save your photos".

In order to request a permission at run time, Apple offers various system libraries or *Frameworks* responsible for each data point. For example, the *PhotoKit* Framework is responsible for managing access to the photo library. While each framework has its own specific methods to be used to gain access, the process usually comprises of two steps: (1) checking the authorization status, and (2) requesting authorization. In the first step, a developer can check if the required permission has already been granted by the user (e.g., if this is not the first time the app is executed). If not, the app can use one of the framework's provided methods to ask for permission. The system will then display a dialogue to the user, showing the text defined in the respective usage description, giving them a choice to accept or deny the request. After that, the app is informed about the decision an can react accordingly.

Certain permissions/privileges can only be requested or used by the app if they have been additionally been approved beforehand by Apple during App Review. These permissions are called *Entitlements*. This includes access to the most privacy-sensitive permissions or specific execution modes of the app, e.g., access to health data, background execution of the app or access to discover and communicate with devices on the Wi-Fi network.

3.2.3 Jailbreaking

Even though sandboxing, the permission system and other mechanisms provide enhanced security and privacy for users, these features may pose a restriction for implementing certain use cases. Particularly, for dynamic analysis, apps can not be analyzed or inspected in the background at run time due to the locked-down nature of the system. *Jailbreaking* or *rooting* refers to the process of circumventing these restrictions by escalating privileges in order to gain root access to the device [KHRW19], often times by exploiting security bugs in the kernel or firmware.

Once root access has been obtained, the system can be configured and extended like any other UNIX-based operating system. For example, it allows access to the device via SSH. Most jailbreaking tools also install Cydia, a third-party App Store or package manager, which allows to conveniently install and run apps or code by third-parties to modify the system [EKKV11]. Naturally, modifying or disabling security principles on the platform can drastically reduce the system's security or privacy of the user, and is thus discouraged by Apple [App18].

For this thesis, we needed to use a jailbroken iPhone in order to be able to inspect apps at run time using Frida, as described in Sections 3.6 and 4.1.

3.3 Mobile App Analysis

The methods used for the analysis of apps are commonly distinguished as being either *static* or *dynamic* [KSB+21]. In this section, we provide a brief introduction into both types as well as their benefits and drawbacks.

3.3.1 Static Analysis

Static analysis methods include all kinds of inspections that can be performed without executing an app. In practice, this may include binary analysis, inspection of de-compiled code, or code snippet fingerprinting (i.e., looking for known code parts or patterns in an app). Specifically, using static analysis, it is easy to discover if an app may, for example, include a third-party library known to be used for advertising or tracking.

Scalability. One major benefit of static analysis compared to dynamic analysis is that it scales very well [KSB+21]. This is due to the fact that the same set of checks can be performed for many apps at once. For example, checking whether a de-compiled code contains any *known* dangerous code paths or commands can be verified at scale.

Obfuscation and dynamic code execution. In practice, certain methods can be applied by developers to prevent static analysis or make results less useful. Code obfuscation, for example, aims to make it harder to understand or predict the execution of certain code paths. This could prevent a static analyzer from identifying known dangerous code patterns in practice. Additionally, studies have shown that up to 30% of apps render code dynamically at runtime [ROF+19], meaning that the actual behavior of an app may be significantly different than what its underlying static code suggests.

Expected results. Static analysis offers great insights about the theoretical capabilities of an app. It is important to note, however, that not all code paths contained in an app are necessarily also executed at run time. An app may, for example, only use a small fraction of an included third-party library, which may not trigger a privacy leak in practice, while the pertinent code to do so would be included in the app's source code. Due to the aforementioned limitations, results obtained from static analysis need to be interpreted with caution, since it may include false positives.

3.3.2 Dynamic Analysis

Dynamic analysis refers to all kinds of analysis mechanisms executed during the run time of an application. Usually, this involves executing an app in an environment that is ideally both realistic but also allows researchers to evaluate its behind-the-scenes behavior. Simulating a real-world environment is important, as some (potentially malicious) apps may behave differently if they detect being analyzed or run in a simulator [CFL+17].

Scalability. In contrast to static analysis, the biggest challenge of dynamic approaches is scalability. Apps usually require user interactions to execute the various code paths contained in it. This is why manual dynamic analysis in a lab environment is only feasible if the number of to-be-tested apps is relatively small, given that installing, using and analyzing apps manually takes a substantial amount of time. For larger case studies, apps can either be tested "in the wild" by real users or user interaction simulators (so-called UI fuzzers) may be used, which simulate user input by (semi-)randomly interacting with the user interface. While UI fuzzing improves the scalability of dynamic analysis, one of the major shortcomings is its tendency to produce limited results in certain cases, e.g., if an app requires a login directly after startup [ROF+19].

Expected results. Discovered privacy leaks using dynamic analysis can be interpreted as a lower bound of possible leaks performed by the tested app, since there may always be code paths that have not been triggered during the (simulated) analysis period. Thus, in contrast to static analysis, this method does not produce false positives, but rather may return false negatives [ROF+19].

3.4 App Store App Privacy Details

Since December 8, 2020, Apple requires all app developers to disclose their app's privacy practices in the form of App Privacy Details[1] on the App Store [Cam20]. In effect, these Privacy Details are standardized labels appearing on each app's App Store page. An example is shown in Figure 3.1. They aim to give users an easy to understand overview of how privacy-sensitive user data *collected* by the app is treated behind the scenes. Collecting data roughly refers to *transmitting* user data off the device and *storing it in an accessible way* for longer periods than necessary to fulfill the current request in real time. In broad terms, developers need to disclose *what kinds of data* they collect, for what *purpose*, whether the data is stored *in a user-identifiable manner or not* and, most importantly, whether it is *shared with third-parties or combined with third-party data for tracking purposes*.[2]

Types of collected data. Apple groups the kinds of privacy-sensitive data that may be collected into 13 different categories[3] (along with a 14th "other" category), such as Contact Info, Location, User Content and Identifiers. For each of the categories, apps must select one or more specific items contained in it that best describe the collected data, e.g., for "Identifiers", apps need to specify whether they use the "User ID" and/or "Device ID"; for "Contact Info", they need to specifically declare whether Name, E-Mail address, phone number, physical address or "Other User Contact Info" is collected.

[1] https://developer.apple.com/app-store/app-privacy-details/ (accessed Oct. 21, 2021)

[2] An exact definition of "data collection", "tracking" and "third-party data" can be found on Apple's website at: https://developer.apple.com/app-store/app-privacy-details/#data-collection (accessed Oct. 21, 2021)

[3] As of Oct. 21, 2021

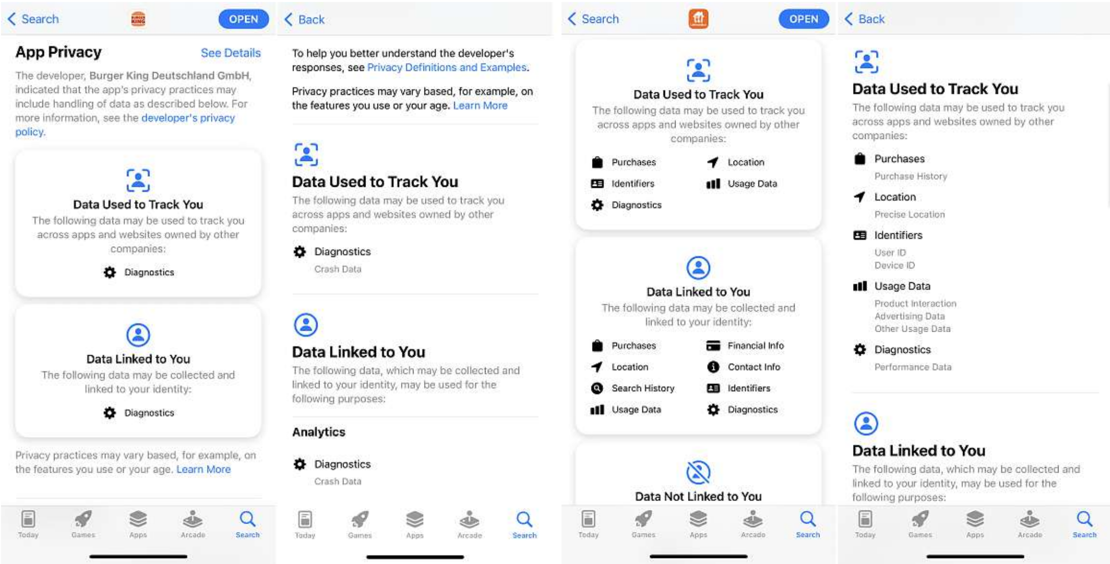

Figure 3.1: Example App Privacy Details of the Burger King Österreich and Lieferservice.at app. For each app, both the overview and the details page are shown. (captured on Nov. 28, 2021)

Linking data to the user. For each data type, apps need to declare whether they store collected data in a way that can be tracked back to the user or not. As an example, an app may specify in the category "Usage Data" that "Product Interaction" data is collected. This kind of data may be collected just for the general purpose of counting how often a specific app feature is used by customers, in which case it would be sufficient to collect this data anonymously without storing which user exactly uses this feature. On the other hand, if a developer's intent was to *personalize* the app experience for each individual user, they would need to associate the collected usage data with the User ID, hence they would need to declare that they store product interaction linked to the user.

Data use. As outlined in the above example, merely specifying the kinds of data and whether they are stored linked to the user does not convey the *intent* or *purpose* of the data collection, which is why Apple requires developers to also declare one or more purposes *for each data point.* Developers can assign whether they use it for "Analytics", "Product Personalization", "Third-Party Advertising", "Developer's Advertising and Marketing", "App Functionality" or "Other Purposes".

Tracking. Up to this point, developers have specified the *type* of data, how they *store* it and for what *purpose*. Finally, Apple asks for each data point whether the collected data is also shared or combined with third-party data, given that this practice can have serious privacy implications for the user and is usually hidden somewhere in the Privacy Policy and therefore hard to identify for users.

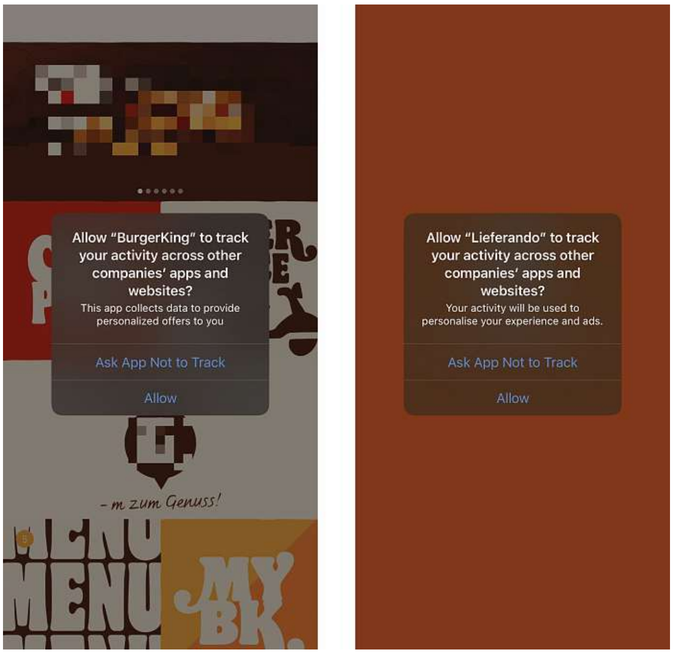

Figure 3.2: Example App Tracking Transparency tracking requests included in the Burger King Österreich and Lieferservice.at app (captured on Nov. 17, 2021)

In summary, App Privacy Details are all about *transparency*; their introduction has given users unprecedented insights into the privacy behavior of apps and allows them to evaluate and compare them in a structured and standardized manner. One of the shortcomings is that they are currently not audited or verified by Apple. In Section 4.2 we evaluate whether and how such a validation process could be accomplished using static and dynamic analysis.

3.5 App Tracking Transparency

Along with the introduction of App Privacy Details in 2020, Apple also introduced the new App Tracking Transparency (ATT) policy. It requires developers to ask their users for explicit approval to be tracked, which is a major policy change in favor of user privacy.

Prior to the introduction of ATT, apps could simply access the Advertising Identifier by default, but users had the option to opt-out from providing the identifier in the system settings. In the new system that went into effect with the release of iOS 14.5 in April 2021, access changed to being opt-in for each app individually, and users also have the choice to turn off tracking completely (i.e., automatically denying tracking permission requests) in system settings.[4] In order to obtain access, apps now have to declare a new permission

[4] "Upcoming AppTrackingTransparency requirements", Apple Developer News, Online, https://developer.apple.com/news/?id=ecvrtzt2 (accessed Nov. 17, 2021)

in their `info.plist` file (`NSUserTrackingUsageDescription`) and explicitly ask their users if they agree to be tracked using a system dialogue at run time (see examples in Figure 3.2). If the app tries to access the AdId before asking the user or after access has been denied, the system merely returns zeros instead of the identifier.

Given that advertising and tracking companies rely heavily on the AdId, e.g., for displaying personalized ads, major companies such as Facebook protested about the change. Facebook claimed that the change would be “devastating for small businesses” [War20]. However, Apple still continued with the introduction, receiving praise, e.g., from the Electronic Frontier Foundation (EFF), which called it “a solid step in the right direction” [GC21].

By the time of our analysis in November 2021, ATT had already been in effect for about half a year. Therefore, we investigated in our analysis how many apps had already adopted the new system and whether they actively asked the user for tracking permission.

3.6 Dynamic Instrumentation on iOS

Frida[5] is a powerful open source toolkit that enables dynamic instrumentation on iOS and Android devices. As such, it enables us to monitor and control running apps. Specifically, it acts as the foundation that allows us to perform API method call interception, as described later in Section 4.1. In this section, we provide an introduction into the necessary basic concepts of dynamic instrumentation using Frida. Figure 3.3 provides an overview of the involved parts and their responsibilities.

3.6.1 Taking Control Over a Process

Using the linux/unix command `ptrace`, Frida is able to gain control of an iOS app. In simplified terms, it pauses a running thread in the target process, allocates memory to insert a bootstrapper,[6] which then loads and executes an `.so` (shared object/library) file so that *frida-agent* is started on a separate thread [RT15]. Control is then passed back to the paused thread, who continues execution without noticing that anything has changed. However, since frida-agent is now running “inside the app”, it can now be monitored and controlled from the outside.

3.6.2 Dynamic Instrumentation with Frida Scripts

The setup process described above is executed by the Frida Core. It communicates with the device and sets up the frida-agent in a target app. The Core is written in C, but there are multiple wrappers available, which make using Frida simple with popular tools such as Python or Node.js.

[5]https://frida.re (accessed Oct. 21, 2021)
[6]Short snippet of executable statements used to set up the injection.

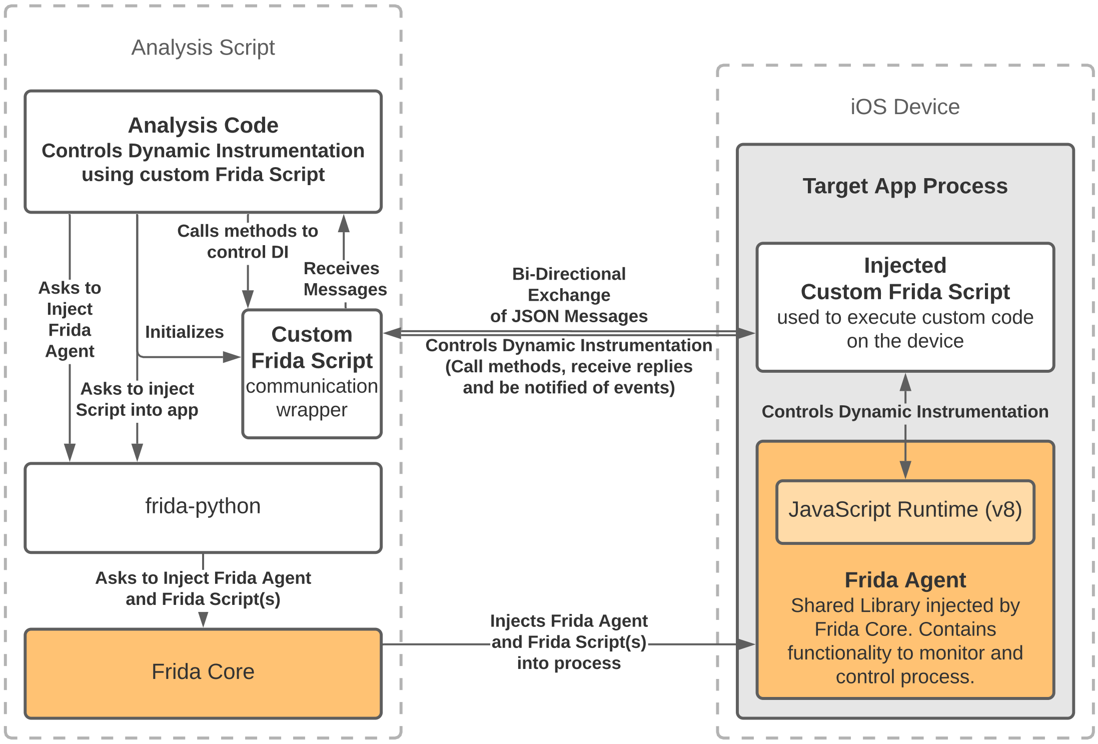

Figure 3.3: Overview of Dynamic Instrumentation on iOS using Frida.

Once communication with the frida-agent is set up, the main interaction happens with Frida Scripts written in JavaScript. Specifically, the agent encapsulates a JavaScript runtime (V8) and offers a JavaScript API that communicates with the frida-agent to enable control of the process. Using Frida Core, developers can submit JavaScript code to be executed in the runtime. These scripts can "export" methods that can then be called from the main analysis script.

Frida Script Workflow Example. In order to make instrumentation with Frida easier to understand, we provide a short example workflow in Listing 3.1.

In this example, we first import the frida-python wrapper to communicate with the Frida Core. Then, we use the convenient methods provided to connect to the device, launch the target app, set up the frida-agent and attach to it. We then provide a few lines of JavaScript to be loaded and evaluated by frida-agent's JavaScript engine. Afterwards, we resume the process, having obtained control over it. Finally, we make a method call to our custom Frida Script. Even though it appears that we are only calling the local method `frida_script.exports.hello()`, the method is actually executed inside the target app and both the execution command and return value are passed via a bi-directional message exchange link, as shown in Figure 3.3.

```
import frida

frida_device = frida.get_usb_device()

# launch app and pause execution
process_id = frida_device.spawn("com.example.app")

# inject frida-agent and take control
frida_session = frida_device.attach(process_id)

# custom JavaScript to be loaded into app
javascript_source = """
rpc.exports = {
      hello() {
            return "Hello, World!"
      }
}
"""

# create and load script
frida_script = frida_session.create_script(javascript_source)
frida_script.load()

# resume app
frida_device.resume(process_id)

# call exported function
print(frida_script.exports.hello())
# prints "Hello, World!"
```

Listing 3.1: Example Python Frida instrumentation setup using a custom Frida Script, which is loaded into the target app and executed on the device.

Asynchronous Frida Script Communication. Even though we have just demonstrated a case where communication with Frida seems synchronous, most of the time we are confronted with situations where we do not expect an immediate response from our Frida Script, e.g., when intercepting API method calls, where we do not know in advance when the respective method will be invoked in the target app. For such purposes, Frida's embedded JavaScript API offers a message-passing system, particularly a `send(payload)` function. In the following Listing 3.2, we have changed the JavaScript source to use the *send* method to deliver a message to the analysis script instead of directly returning it. Additionally, we added a message handler in Python and registered it before loading the script into the target app.

```
javascript_source = """
rpc.exports = {
      async_message() {
            // imagine some long-running task here

            send('Hello, World!')
      }
}
"""

def handle_frida_script_event(message):
      print('recevied message from Frida Script:')
      print(message['payload'])

# create and load script
frida_script = frida_session.create_script(javascript_source)
frida_script.on('message', handle_frida_script_event)
frida_script.load()
```

Listing 3.2: Frida Script example demonstrating asynchronous message-passing.

If we now call `script.exports.async_message()`, our message handler will get called with the target payload. Since the payload can be any valid JSON object, we can use this approach to easily pass extracted data from our target app to our analysis script.

3.6.3 Frida JavaScript API

In order to perform instrumentation, Frida provides a powerful JavaScript API[7] that is automatically available inside Frida Scripts. As a result, all instrumentation code needs to be implemented as part of a Frida Script, loaded into the target app and is then executed directly on-device.

The JavaScript API offers a rich set of features, enabling interception of method calls, tracking code instructions step-by-step ("stalking", like a debugger), memory and file access, executing native Objective-C code (or Java on Android) and even performing CPU instructions. Details on how we intercepted API method calls are elaborated in our Methodology Section 4.1.

3.7 Network Analysis on iOS

Mitmproxy[8] is an open source HTTPS proxy. It can be used to intercept, modify, replay, and inspect network traffic sent out, e.g., by an iOS device. The benefit of using

[7] Frida JavaScript API, https://frida.re/docs/javascript-api/ (accessed Oct. 29, 2021)
[8] https://mitmproxy.org (accessed Oct. 21, 2021)

mitmproxy to analyze HTTPS network traffic is its ease of use and the possibility to intercept traffic without requiring an iOS device with a jailbreak.

Once mitmproxy is running on a device accessible on the network, it can easily be configured as a proxy in the Wi-Fi settings of an iOS device. After that, all traffic will get routed through the relay and can thus theoretically be intercepted.

However, given that HTTPS traffic is encrypted and all iOS apps have been required to use HTTPS since January 1, 2017 [Con16], mitmproxy offers a Certification Authority (CA) certificate, which needs to be trusted by the iOS device after configuring the proxy. Once the CA is trusted, mitmproxy can issue trusted SSL certificates for every accessed domain by the device and therefore successfully act as a machine-in-the-middle device, as long as the respective app does not use certificate-pinning.

Mitmproxy workflows can easily be automated by developing add-ons written in Python, which act as event handlers that can react to network events. As detailed later in Section 4.2.3, we implemented such an add-on for this thesis in order to detect if an iOS app is transmitting the Advertising Identifier over the network.

3.8 User Interaction Simulation on iOS

Appium[9] is a cross-platform user interface ("UI") automation tool. As such, it is mainly intended to perform automated tests on apps. Developers use UI tests to verify that their app's user interface behaves as intended, since manually checking all functionalities every time the app is updated can become infeasible for larger software projects. In essence, Appium allows developers to programmatically inspect and interact with the displayed UI on supported target platforms, such as Android and iOS.

Using the WebDriver protocol, Appium provides a standardized platform-independent API, allowing developers to interact with the UI of apps running on various operating systems without requiring specific knowledge about the used platform - they only have to know how to interact with the UI using Appium's API. In the background, Appium then translates developer's commands into native calls for the respective platform.

In our study, we used Appium to simulate user interactions as part of our dynamic analysis, a process often times referred to as "UI fuzzing" [ROF+19]. This allowed us to automate the app analysis and thereby improve the scalability, since a real user interacting with the inspected app is no longer required. Additionally, it improves the results of our dynamic approach, given that apps usually rely on interactions with their user interface in order to activate or invoke various parts of their code. Details about how we applied user interaction simulation are provided in Section 5.3.7.

[9] https://appium.io (accessed Nov. 18, 2021)

CHAPTER 4

Methodology

In this chapter we will discuss the methodological approach used for this thesis in detail. As briefly introduced in Section 3.3, *static* and *dynamic* analysis are the two commonly distinguished app analysis methods. Overall, this work utilizes a combined approach of a basic static and a more advanced dynamic analysis.

The goal of the methodological approach was to design and implement an analysis workflow that groups together multiple techniques in order to enable a behind-the-scenes privacy analysis of iOS apps. More specifically, dynamic instrumentation was used to identify privacy sensitive method invocations at run time, network analysis was used to determine whether the Advertising Identifier was sent off the device and static analysis enabled us to compare the run time behavior with the declared permissions and App Privacy Details ("privacy labels") on the App Store.

The main contribution of this work is a combination of the methods described in this chapter into an automated iOS dynamic analysis platform, enabling researchers to perform privacy analysis of a large number of iOS apps without manual interaction. This chapter describes the privacy analysis methods that act as the foundation of the developed workflow, while the next chapter provides an in-depth methodological description of the automated analysis platform's design and implementation.

4.1 Privacy Behavior Analysis

Dynamic Instrumentation is the main analysis method used for this thesis. It allows us to inspect and influence an app's behavior while it is running by attaching to the process.

Specifically, we were interested in finding out if, when, and how often an app would access certain privacy sensitive information. Apps can only access such information by calling the system APIs or frameworks responsible for managing the respective information. For example, if an app would like to gain access to the user's photo library, it is required to call the respective functions provided by the PhotoKit iOS framework.

As a result, measuring privacy-sensitive data access involved intercepting the calls to such API methods. In order to achieve this, we used Frida's Interceptor (part of its JavaScript API, as introduced in Section 3.6) to attach a listener to Objective-C API calls of interest.

It is important to note at this point that even though the programming language used to develop iOS apps has changed to Swift in recent years, the former programming language Objective-C was still used behind the scenes for the method calls that we intercepted for our analysis. Indeed, it turned out that even for one of the newest iOS frameworks, AppTrackingTransparency, which manages access to the Advertising Identifier, intercepting Objective-C calls still worked without any problem even for apps developed completely using Swift. We therefore want to highlight that even though Frida only supports intercepting Objective-C calls, this does not pose a limitation to our interception methodology, since Objective-C API methods are still used behind the scenes when apps are implemented using Swift.

4.1.1 API Method Call Interception

Applying method call interception requires certain steps to be repeated for every to-be-analyzed data type [LBAU17, KSB+21]. In order to illustrate these steps and the associated challenges, we use the example of analyzing photo library access.

1. Determine Responsible APIs or Frameworks

First and foremost, it is crucial to answer the question "*Which APIs or frameworks have to be used by developer to gain access to the data type of interest?*". The Apple Developer Documentation[1] provides all information necessary to determine these responsibilities. In the case of photo library access, apps must use the PhotoKit framework for all interactions with the user's photos or videos. Particularly, each such framework usually contains multiple classes and methods to enable CRUD (create, read, update and delete) operations. While this work focuses only on *read* operations, it would be possible to also intercept other interaction types in an analogous manner.

2. Determine Method Calls Of Interest

Once the responsible frameworks or APIs have been determined, classes and methods of interest for interception have to be determined. Again, using the Apple Developer Documentation, we now need to create a summary of all the methods the developer would have to use to perform the CRUD operation of interest. Importantly, it turned out not to be feasible to simply intercept all methods of a certain API to detect usage thereof, since the inspected app crashes if too many methods are intercepted at the same

[1] https://developer.apple.com/documentation/technologies

time. As a result, it was crucial to only select a representable subset for the intended analysis purpose.

For this work, we were interested in three different types of method calls: 1. permission status checks, 2. permission requests 3. read operations.

As detailed in Section 3.2.2, apps must ask the user for access to privacy-sensitive information before they can use this data. On the code-level, this process is similar for most data types. In the case of photo library access, the `PHPhotoLibrary` class is responsible for permission request and status check and offers the following methods that need to be intercepted:

```
+ requestAuthorization:
+ requestAuthorizationForAccessLevel:handler:
+ authorizationStatus
+ authorizationStatusForAccessLevel:
+ checkAuthorizationStatusForAPIAccessLevel:
```

Listing 4.1: Required methods to be intercepted for photo library permission status and permission request calls

Next, we needed to compose a summary of *read* operations of interest. In contrast to permission requests, accessing data is usually implemented quite differently in each framework. For example, the Advertising Identifier can simply be accessed using the `advertisingIdentifier` instance variable of the `ASIdentifierManager` class, whereas the items from the photo library may be accessed using any of 26 static functions of `PHAsset` (excerpt shown in Listing 4.2)

```
+ fetchAssetsWithLocalIdentifiers:options:
+ fetchAssetsWithOptions:
+ fetchAssetsWithUUIDs:options:
+ fetchAssetsInAssetCollection:options:
+ fetchAssetsForPerson:options:
+ fetchAssetsForFaces:options:
... plus 20 more ...
```

Listing 4.2: Required methods to be intercepted for photo library PHAsset read access

In order to generate such a list of candidates, we used Frida's built-in functionalities to enumerate all methods of an Objective-C class. For instance, in the Frida console,[2] the following line prints a list of all Objective-C method declarations of the `PHAsset` class: `ObjC.classes.PHAsset.$ownMethods`. We then manually inspected them and selected the appropriate subset of methods based on their names and descriptions. Additionally, we used a simple Frida Script that intercepted all class methods and

[2] Instead of using Frida using, e.g., a Python script, it is possible to run and attach it to a target app directly via the command line. The interactive console runs every command as a Frida Script and therefore allows easy experimentation.

counted method invocations of a demo app to determine the methods most relevant for interception.

3. Intercept and Analyze

Once a target set of methods had been determined, we used Frida's `Interceptor` to attach a so-called *handler* to each of them.

A handler consists of an `onEnter` and `onLeave` callback. Once attached, the former is called as soon as the target method is invoked and contains an array of arguments passed to the intercepted method, whereas the latter callback is called when a value is about to be returned and also receives a pointer to the return value. Listing 4.3 shows a basic setup of such an invocation handler.

```
Interceptor.attach(
  ObjC.classes[className][methodName].implementation,
  {
    onEnter(methodArguments) {
        // code to be executed when the method is invoked
    },
    onLeave(returnValue) {
        // code to be executed once the intercepted method returns
    }
  }
)
```

Listing 4.3: Basic structure of a method invocation handler

We made use of the intercepted arguments for certain data types to determine additional information for our privacy analysis. As an illustrative example, we intercepted `NSURLSession` network request from methods like `- downloadTaskWithRequest:`, which receives a `NSURLRequest` object with information about the current URL request to be executed. Listing 4.4 demonstrates how the attached `onEnter` method could extract the to-be-called URL and HTTP method so that we can use this information for our analysis.

```
function onEnter(methodArguments) {
    let urlRequest = new ObjC.Object(methodArguments[2])
    let url = urlRequest.URL()
    let method = urlRequest.HTTPMethod().toString()

    console.log(`Intercepted URLRequest (${method} ${url})`)
    // e.g., Intercepted URLRequest (GET https://example.com/)
}
```

Listing 4.4: On enter invocation handler for - downloadTaskWithRequest:

Privacy-Sensitive Method Invocations	**Permission Status Check**	**Permission Request**	**Analysis Metrics**
Advertising Identifier	✓	✓	Access count
Camera	✓	✓	Access count
Microphone	✓	✓	Access count
Photo Library	✓	✓	Access count
User Location	✓	✓	Access count, monitoring requests
Contacts	✓	✓	Access count and type: search, enumeration or personal contact access
Network Requests	Not needed	Not needed	Target URL and HTTP method. Check if URL is known tracking URL.

Table 4.1: Overview of the collected analysis data obtained by intercepting privacy-sensitive method invocations.

Intercepted Privacy-Sensitive Data Access

Overall, for this work we applied the above steps to seven different privacy-sensitive data types so that we could then monitor and detect access to these kinds of data, e.g., photo library or contacts access. Table 4.1 presents an overview of these intercepted method calls and how they were analyzed. Network request URLs are classified as tracking URLs if they appear on the NoTracking hosts blocklist,[3] which contains a large and regularly-updated collection of such addresses.

For each of the data types, we evaluated *Permission Status Check*, *Permission Request*, access count as well as custom metrics. We will now explain why we selected these measurement types for our methodology:

Permission Status Check. As explained in Section 3.2.2, iOS apps must declare any potential permissions they may want to request during run time in advance in their `info.plist` file. However, since apps tend do declare more permissions than they actually use, a declared permission does not directly imply the actual request later on. For every declared permission, apps can check at run time if a certain permission was, e.g., *granted*, *denied* or *not yet requested*. Therefore, we intercepted these method calls as they act as an indicator for an interest in a specific data type, e.g., if an app would like to request the user's contacts, it may check at startup if it *can* already access this data. By intercepting these calls, we gain a better understanding of the kinds of data an app is *actively* interested in, rather than just declaring *potential* interest.

[3]https://github.com/notracking/hosts-blocklists (accessed Nov. 30, 2021)

Permission Request. Naturally, asking the user for access to a certain data type is an important metric to *validate* the app's intent to make use of this data, which is why we have monitored permission requests by intercepting those calls.

Access Count. While having requested access to a certain data type is a strong indicator of true interest, intercepting read access method calls provides further insights into how frequently data is accessed. For instance, we could therefore detect whether an app is enumerating the user's contacts only once (e.g., directly after the user has granted permission), or whether it keeps continuously accessing contact information during run time.

Further Metrics. Finally, there are many additional questions that could be answered by a more detailed analysis of the intercepted method calls and arguments. In this work, we used such an analysis to extract information about URL requests (simplified sample code in Listing 4.4) and then used this information to identify whether an app communicated with known tracking domains. Additionally, we analyzed *how* contacts are being accessed (enumeration, search or personal contact access). Future work could investigate, e.g., what *particular* information of a contact is accessed (name, e-mail address, etc.), as discussed in Section 7.2.

4.2 Verification of App Privacy Details

App Store Privacy Details aim to give users deep insights into how apps use privacy-sensitive data. As described in Section 3.4, one of the main shortcomings is that these declarations are not audited or verified by Apple. In this section, we discuss how dynamic analysis may be used to verify them and describe how we used network analysis to attempt a verification of the "Device ID" privacy label.

4.2.1 Verification Types

First and foremost, it is important to note that there are multiple types of verifications that could be possible:

- **Verifying a label that is present:** If a label has been declared by the developer, this means that potential interest is present and we should thus find evidence of such interest using dynamic or static analysis.
- **Verifying a label that is *not* present:** Even though verifying that the declared labels are correct is important, when it comes to user privacy, verifying that certain kinds of data are *not* accessed in the absence of a declared label might arguably be even more important. For example, if an app does not declare collection of the Advertising Identifier (Privacy Label "Device ID"), then it is important to be able to verify that the ID is indeed *not* collected.

Based on these considerations, we decided for our methodology that it would be most interesting to find contradictions, i.e., cases where the developer has *not* declared a privacy label but *should have* based on the App's data collection behavior. In such cases, the user's privacy is being undermined without the knowledge of the user and the developer may be violating their Apple Developer Agreement.

4.2.2 Detecting Privacy-Sensitive Data "Collection"

One of the main challenges when trying to verify a missing privacy label lies in the definition of the word "collect". Specifically, Apple requires developers to declare a certain privacy label only if the respective data type is *collected*. According to the definition of the term provided by Apple, "collect" refers to "transmitting data off the device in a way that allows you and/or your third-party partners to access it for a period longer than what is necessary to service the transmitted request in real time".

Based on this definition, we conclude that there are only two reasons for a privacy label ***not*** to be declared: 1) the respective privacy-sensitive data is *not* transmitted at all or 2) the data is transmitted in a way that does *prevent* the receiver from accessing it for a period longer than what is necessary to service the transmitted request in real time.

As a result, we argue that if *we* can gain *access* to privacy-sensitive user data transmitted off the device by intercepting the network calls sent by an app, this must mean that the developer and/or a third-party partner would also be able to get access. Consequently, since we could simply store the data and thus access it "longer than what is necessary to service the transmitted request in real time", the same must be true for the actual receiver and thus a privacy label would have to be present for the respective collected data type. Thus, in summary, we applied network traffic analysis to detect privacy-sensitive data transmitted off the device (in our case we focused on the Advertising Identifier). If we can access and read such data on the network, then a privacy label must be declared on the App Store (based on the above reasoning).

4.2.3 Network Traffic Analysis

In order to add network traffic analysis to our methodology, we made use of the open-source tool mitmproxy. As briefly introduced in Section 3.7, it allows researchers to intercept, modify, drop and replay HTTP/HTTPS requests and responses.

Broadly speaking, there are two main ways to analyze the network traffic using a mitm proxy. Analysis can either happen in real-time or as a dedicated analysis step by analyzing the network traffic log after the dynamic analysis is complete. In our case, we opted for the former type.

Processing requests in real-time with mitmproxy is possible by creating an "add-on". Add-ons are Python scripts that can be passed to the mitmproxy instance as a startup parameter (e.g., `mitmweb -s NetworkAnalysis.py`). The injected script is then executed along with the proxy server and acts as an event handler, that is for every network request detected by the proxy, certain methods of the add-on are called, allowing the script to analyze, modify, replay or drop the request/response.

For our goal of discovering the Advertising Identifier in a network request, we created a very basic add-on that simply looks for in an intercepted request's body, headers and URL. In our analysis results (presented in Chapter 6), it turned out that this simple approach was sufficient to discover the identifier. Future work could easily extend this analysis approach to accommodate obfuscation attempts. Finally, our implemented add-on is only meant to serve as an example implementation for future network analysis efforts that could focus specifically on discovering more kinds of privacy-sensitive data to verify additional privacy labels.

In summary, our add-on (simplified implementation presented in Listing 4.5), proceeds in the following steps:

1. Register as an mitmproxy add-on (at the bottom of the script).
2. Upon a received network request, mitmproxy automatically calls the `request` method of the `NetworkAnalysisAddOn`.
3. The request content, url and headers are decoded and then searched for the occurrence of the Advertising Identifier.
4. If the Advertising Identifier was found, we notify the AppAnalyzer (Section 5.3.3), who can then incorporate the event in it's analysis report. The automated analysis platform, including the AppAnalyzer, is discussed in detail in Chapter 5.

```
import requests; import traceback

# add-on configuration, required before analysis
APP_ANALYZER_ADDRESS = "localhost"
APP_ANALYZER_PORT = "8042"
APP_ANALYZER_URL = f"http://{APP_ANALYZER_ADDRESS}:{APP_ANALYZER_PORT}"
ADVERTISING_IDENTIFIER = "XXXX-XXXX-XXXX-XXXX-XXXX"

class NetworkAnalysisAddOn:
  def request(self, flow):
    """
    Request handler, called by mitmproxy
    """
    request = flow.request

    decoded_content = request.content.decode('utf-8', 'ignore')

    if ADVERTISING_IDENTIFIER in decoded_content:
      print('Advertising Identifer discovered in transmitted content')
      self.notify_app_analyzer(request)

  def notify_app_analyzer(self, request):
    """
    Notify the App Analyzer that the Advertising Identifier
    was discovered in a request
    """
    event = {
      "event_name": "tracking",
      "host": request.host,
      "method": request.method
    }
    try:
      requests.post(f'{APP_ANALYZER_URL}/event/tracking', json=event)
    except:
      print('failed to notify App Analyzer')
      traceback.print_exc()

# global variable, used by mitmproxy to call all registered add-ons
addons = [ NetworkAnalysisAddOn() ]
```

Listing 4.5: Simplified example implementation of our mitmproxy add-on, looking for the Advertising Identifier in the request body and then notifying the AppAnalyzer about the event. Our actual implementation also searches in the headers and URL.

4.2.4 Attributing a Network Request to a Specific App

Given that the proxy receives all requests sent out by the iOS device, it is important to note that simply detecting certain privacy-sensitive data would not necessarily mean that it came from the analyzed target app.

In our analysis, we ensure that discovered privacy-sensitive data is coming from our analyzed app by combining the network analysis with the dynamic instrumentation. Specifically, in our results we only attribute the intercepted network request containing the Advertising Identifier to our target app if we also detected a network request to the same target URL using API call interception. This way, we have two independent sources informing us about a network request. Only if the target URL retrieved by both analysis methods matches and the request has happened at around the same time, we attribute it to the analyzed app.

4.2.5 Retrieving Privacy Details from the App Store

Now that we could detect cases where a privacy label for the Advertising Identifier *should* be present, we needed a way to compare our findings with the declared labels on the App Store.

In order to integrate this into the automated analysis workflow, we implemented downloading and parsing of the privacy labels for each target app. Since there is no documented API provided by Apple that would enable us to download the required infos directly, we had to use a workaround to access this data.

The retrieval process is based on the method implemented by Marcel Pfefferle for his bachelor thesis at Ruhr-University Bochum:[4] First, we download the App Store page of a target app. The source contains a `<meta>` HTML tag containing an API token. We extract and use this token to fetch the App Store's own API for a JSON representation of the page. The resulting JSON contains various infos about the app itself, as well as a representation of the declared App Privacy Details.

The declarations are grouped into privacy types. As mentioned in Section 3.4, there are currently three types: “Data Used to Track You”, “Data Linked to You”, and “Data Not Linked to You”.

Each privacy type may contain multiple purposes and multiple data categories. Each category can contain one or more data types. E.g., an app may declare Advertising Identifier use by specifying to collect “Data Linked to You” with the purpose “Third-Party Advertising”, containing the category “Identifiers” and in there declaring interest in the “Device ID”. Listing 4.6 shows a short snippet of such a declaration.

Parsing Approach: Our goal was to assign a boolean value for each supported privacy type (e.g., camera or contacts access), indicating whether a privacy label was declared for

[4] Apple App Store Crawler by Marcel Pfefferle, available on gitlab: https://gitlab.com/marzzzello/appstore_crawler (accessed Oct. 26, 2021)

it or not. Since we are only interested in verifying whether a certain privacy label *should be there*, we do not differentiate between the declared *purpose* of a certain label, e.g., if we discover that the Advertising Identifier is collected, we only care if the corresponding privacy label "Device ID" was declared for *any* purpose. Therefore, for each privacy type supported by our dynamic analysis (see Table 4.1), we generate a boolean value, indicating whether or not the respective privacy label was declared for *any* purpose.

```
{ "privacyType": "Data Linked to You",
  "identifier": "DATA_LINKED_TO_YOU",
  "description": "...",
  "dataCategories": [],
  "purposes": [ {
      "purpose": "Third-Party Advertising",
      "identifier": "THIRD_PARTY_ADVERTISING",
      "dataCategories": [
        {
          "dataCategory": "Identifiers",
          "identifier": "IDENTIFIERS",
          "dataTypes": ["User ID", "Device ID"]
        }
        // ... further data categories
      ]
    }, {
      "purpose": "Analytics",
      "identifier": "ANALYTICS",
      "dataCategories": [/* further data categories, as above */]
    },
    // ... further porposes
  ] }
```

Listing 4.6: Simplified representation of the JSON Privacy Details response retrieved by the App Store API.

In this chapter we have presented the analysis methods used as the basis of our automated dynamic analysis platform. Specifically, we have shown how we use dynamic instrumentation with API method call interception to analyze access to privacy-sensitive data behind the scenes. Furthermore, we have described our approach of verifying App Store Privacy Details, highlighting that our analysis focuses on identifying cases where data is used without declaring the corresponding privacy label.

Given that the main contribution of this work is a combination of these and other processes into an automated analysis platform, we will dedicate the next chapter to describe our methodological approach used to design and implement this platform in detail.

CHAPTER 5

Automated Privacy Analysis Platform

In an effort to enable researchers to apply the methods described in the previous chapter to a wide range of apps, we created an automated iOS privacy analysis platform. The platform is able to perform various dynamic analysis methods without requiring manual interaction. After automatically installing a set of to-be-analyzed apps on a physical iOS device, the platform starts launching one app at a time, intercepting API method calls, simulating user interaction, analyzing network traffic, uninstalling the app, printing out a summarized report with additional log files and then continuing with the next app.

In this chapter, we present our approach used to design and implement this platform in detail.

5.1 Analysis Workflow

We will now briefly introduce the general workflow of a typical analysis making use of the developed platform. For the sake of providing an easy to understand overview, technical details are omitted here and will be discussed thoroughly in the subsequent sections.

1. **Gather list of to-be-analyzed apps.** First and foremost, a researcher needs to select a set of apps to be analyzed. This could be, e.g., the top 100 free apps of the App Store in a specific category and region. The method used to collect this set of apps is up to the researcher. However, tools such as the *Apple App Store Crawler*[1] make it easy to download a large number of App Store page data, which can then be filtered to generate a set of target apps.

[1] Apple App Store Crawler by Marcel Pfefferle, available on gitlab: https://gitlab.com/marzzzello/appstore_crawler (accessed Oct. 26, 2021)

2. **Automated installation of apps.** Once a list of to-be-analyzed apps has been obtained, the analysis pipeline can be used to install all of them automatically on the iOS device.

3. **Automated analysis.** Once everything is set up, the analysis can begin. Each target app will be analyzed by the platform sequentially. For each app, the following actions are taken care of automatically:

   a) **Privacy Label Download:** The app's App Store page is downloaded and saved. Declared privacy labels are parsed and incorporated in the analysis report.
   b) **App launch:** The app is started and the dynamic instrumentation is attached.
   c) **Permission Analysis:** Declared permissions are parsed and incorporated in the analysis report.
   d) **API Method Call Interception:** Privacy-sensitive invocations are observed and analyzed.
   e) **Network Analysis:** All network traffic is routed through a proxy and the traffic is analyzed to look for requests containing the Advertising Identifier. Future work can extend this component to detect other kinds of sensitive information as well.
   f) **User Interaction Simulation:** User taps on the screen are simulated up to a configurable number of actions have been completed.

4. **Automated reporting and logging:** During and after the analysis of each app, multiple output files, such as a raw log of events, are generated. Most importantly, for each app, an analysis summary file is created, containing various information, including the following:

   a) **Analysis information**, such as start and end time.
   b) **App information**, such as name, version and build number.
   c) **Declared App Store Privacy Details**, downloaded and parsed from the App Store, indicating interest in a certain data type during run time.
   d) **Declared Permissions** in the app bundle, including a report of all declared Usage Descriptions.
   e) **Discovered Permission Use (Dynamic Analysis Results)**: For each access type (photos, contacts, advertising identifier, etc.), a standardized summary of the dynamic analysis is presented, answering the following questions:
      - Does the app declare interest in the data type in its `info.plist` file (declared permission)?
      - Does the app declare interest in the data type in its App Store Privacy Details?

- Has the permission status been *actively checked* during run time?
- Has the user been *asked to grant permission* to the respective data type?
- Has data of the respective type been *accessed*?
- How *often* has access occurred?
- For some data types, the *type* of access is reported (e.g., enumeration of all contacts vs. access to specific item)

5. **Automated uninstallation of apps.** After successful completion of the analysis, the platform can automatically uninstall all target apps, allowing the analysis to start from the beginning as well as eliminating potential side effects that would occur if the analysis was run again without re-installing the apps first.

6. **Data review and evaluation.** Finally, it is the researcher's task to review and evaluate the generated results using the provided analysis report file, event log, etc.

5.2 Platform Architecture

The designed and implemented analysis platform combines multiple independent analysis components into a single architecture. Combined, they enable a broad range of possible analyses. Figure 5.1 provides an of overview the implemented analysis setup.

The technical setup comprises of the following components, which will be described in the following sections:

- **Analysis Pipeline** (Section 5.3.1) Responsible for orchestration of the individual components and analysis steps. In effect, it is the entry point into the application, parsing arguments and configuration (target apps, steps, etc.), preparing the individual components and then initiating the automated analysis for one app at a time.

- **App Analyzer** (Section 5.3.3) The App Analyzer manages the dynamic analysis. It uses Frida to interact with the physical device, starts the target app, activates the dynamic instrumentation, keeps track of all detected events (method invocations, network calls, etc.) and writes out the analysis results.

- **Network Proxy with Analyzer** (Section 5.3.5) All network traffic sent out by the iOS device is routed through a machine-in-the-middle proxy. Using a custom add-on, network calls are analyzed for privacy-sensitive data (currently only the Advertising Identifier). Once such a request is discovered, the component notifies the App Analyzer.

- **External Event Receiver** (Section 5.3.2) This component allows external analysis tools (in our case the Network Analyzer) to post events into the analysis pipeline. The events are passed on to the App Analyzer, which processes them.

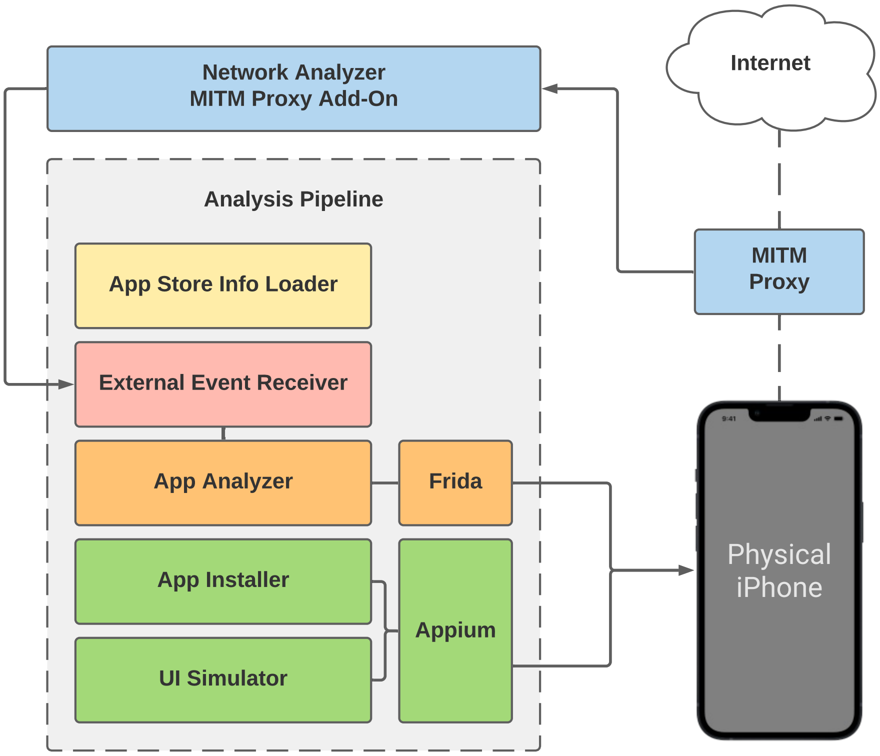

Figure 5.1: Overview of the implemented automated iOS dynamic privacy analysis platform.

- **User Interaction Simulator** (Section 5.3.7) Since dynamic analysis requires user interaction to activate various source code paths in the app, such interactions are simulated with a custom Appium UI simulation script, eliminating the need for a physical user during the analysis.

- **App Store Info Loader** (Section 5.3.4) In order to associate the detected privacy behavior of a target app with its declared Privacy Details von the App Store, this data is downloaded and parsed by this component. The obtained infos are then incorporated into the report of the App Analyzer.

- **App Installer and Uninstaller** (Section 5.3.6) Installing and uninstalling a large number of target apps manually can be tedious and takes time. This component automates this process.

- **Physical iOS Device and Mac Computer** (Section 8.1) Among other requirements, a physical jailbroken iOS device and a Mac computer are strictly necessary to conduct the analysis.

5.3 Platform Components

5.3.1 Pipeline

The Pipeline component is a Python script that orchestrates, as shown in Figure 5.1, the *App Store Info Loader*, *App Analyzer*, *External Event Receiver*, *App Installer* and *User Interaction (UI) Simulator*.

Particularly, the Pipeline is responsible for parsing the configuration, preparing the individual components and starting the analysis for each app.

It has three different modes of operation, which need to be invoked separately. In each of them, the script first parses the provided arguments and reads the App Store IDs of the target apps from a file (see Section 8.2 for usage and configuration details). After that, it continues depending on the selected mode:

- **Installation Mode:** In this mode, the pipeline only uses the *App Installer* to install the target apps, one after another.
- **Analysis Mode:** Performs analysis for each target app using the *App Analyzer*, *UI Simulator* and *External Event Receiver*. Particularly, it executes the following steps:
    1. Start External Event Receiver
    2. Then, for each app:
        a) Create an instance of the *App Analyzer* and ask it to configure the analysis.
        b) Configure the *App Analyzer* to be the event handler of the *External Event Receiver*.
        c) Tell *App Analyzer* to start analysis session.
        d) Create a *UI Simulator* instance and configure it according to the provided arguments (define simulation steps, etc.)
        e) Start simulating user interactions.
        f) Wait for completion of the simulation.
        g) Tell *App Analyzer* to end the session and complete the analysis.
        h) Continue with the next app until all apps have been analyzed.
- **Uninstallation Mode:** This mode can be used to uninstall the target apps after the analysis. It again uses the *App Installer*'s provided functionalities to uninstall the provided target apps.

Unfortunately, it is not possible to perform all three modes of operation by just calling the Pipeline once (e.g., install, analyze and uninstall all target apps in one step). The reason for this is that in-between those individual steps, brief user interaction is required to configure the network proxy on the iOS device (as described later in Section 8.2).

Given that the App Store uses SSL Certificate Pinning, all apps have to be installed before the network proxy is configured on the iOS device, hence the need for multiple distinct modes.

5.3.2 External Event Receiver

The External Event Receiver is a basic HTTP Server exposing a simple REST API implemented as a Python script without using any third-party dependencies. It serves the purposes of making the analysis pipeline easily extendable with other simulation scripts that could otherwise not easily be integrated into the pipeline, as it is the case with the Network Analyzer, which is started as an add-on of mitmproxy and therefore running in a separate application. Since most applications can easily create HTTP requests, we chose this method as a easy-to-integrate extension point for this platform.

Upon startup, it starts listening on a configurable address and port. It then proceeds with the following steps:

1. **Start Accepting Events.** Unless configured differently, by default the server listens for POST requests on `http://localhost:8042/event/:event-name` where `:event-name` is a placeholder for the name of the POSTed event.

2. **Parse JSON Body.** Once such a request is received, the server parses the request body, which must include a valid JSON payload containing further information about the event.

3. **Notify App Analyzer.** Finally, the External Event Receiver notifies the *App Analyzer* using the `analyzer.handle_external_event(evt_name, evt_info)` method, which can then process the event accordingly. The `event_info` parameter is the parsed JSON payload represented as a Python dictionary.

5.3.3 App Analyzer

The App Analyzer is the main entity responsible for managing the analysis session of the currently analyzed app. Figure 5.2 shows an overview of this component and its responsibilities.

Overall, the App Analyzer is based on an event-driven message-passing system to decouple the individual analysis parts and make the analysis easier to extend in the future. Furthermore, it is responsible for the setup of the Frida session, which includes launching the app, attaching to it and loading the Frida Script used for this platform, a concept introduced earlier in Section 3.6.

Analysis Using Various Frida Event Handlers. In an attempt to make the platform easy to extend (i.e., possible to easily add more kinds of analyses), the App Analyzer makes use of event handlers. Currently, for each privacy-sensitive data type a dedicated

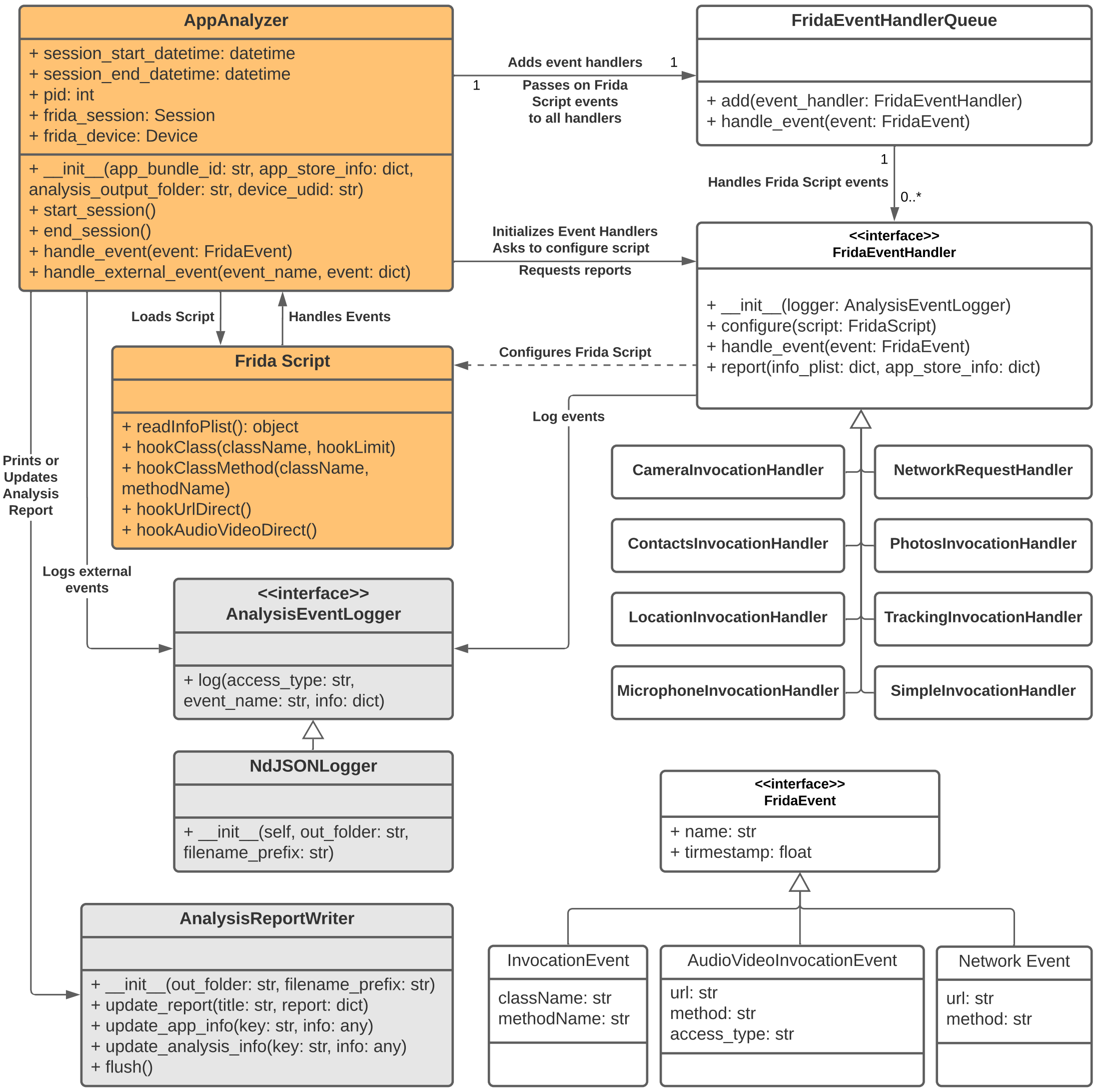

Figure 5.2: App Analyzer Workflow Overview (UML Class Diagram)

event handler is responsible for processing Frida events, e.g., the TrackingInvocationHandler processes Frida Script events related to the Advertising Identifier. After initializing an event handler, it is necessary to invoke its `configure(script)` method, which may call one or more configuration methods provided by the Frida Script, e.g., attach an Interceptor to methods of interest.

Once each event handler has had the chance to configure the script, it is added to an event handler queue. The queue serves the purpose of giving all event handlers an equal chance to respond to *all* events (messages) sent out by the Frida Script, instead of having to explicitly subscribe to *specific* ones. Importantly, this means that it is possible to process the same event in multiple handlers, thereby making the implementation and the possible use cases of the event handlers more diverse.

For example, the `SimpleInvocationHandler` is special in the sense that it processes all `InvocationEvents` it receives and counts classes and method calls. An `InvocationEvent` represents a basic information from the Script that an intercepted class method has been called. While usually only one particular event handler is interested in invocations of a certain class/method, the fact that events are passed to *all* registered handlers in the queue makes it possible that, e.g., the `TrackingInvocationHandler` processes the `InvocationEvent` informing that the Advertising Identifier has been accessed, but the `SimpleInvocationHandler` can also process it to count the number of times *any* class or method has been invoked. This makes the overall analysis system quite flexible.

Analysis Workflow. The general workflow of a typical App Analyzer analysis session involves progressing through the following states (see also Figure 5.2):

1. **Create App Analyzer Instance:** Provide target app bundle id, App Store info (retrieved using *App Store Info Loader* component), an output folder for the analysis report and logs, as well as the UDID (unique device identifier) of the iOS device to be used for the analysis.

2. **Configure and Start Analysis Session:** Using `start_session()`, the app is launched and the Frida Session is configured. Then, the Frida Script is loaded and the various event handlers are initialized. For each event handler, the `configure(script)` method is invoked to give them an opportunity to configure the Frida Script, e.g., start intercepting the respective method calls. Then, they are added to the event handler queue and the app execution is resumed.

3. **Analysis.** In this phase, user-interactions are simulated using the *UI Simulator* (described in Section 5.3.7). During this time, all incoming Frida Scripts events are forwarded using the event handler queue. In addition to that, external events, such as *Network Analyzer* events received by the *External Event Handler* (see Section 5.3.2), are processed. Currently, they are simply logged to the raw event log, but they could also easily be forwarded via the event handler queue in the

future. Finally, occurred events of interest are also logged to the raw log file directly by the event handlers. Details on logging are presented in Section 5.4.

4. **Reporting** Once the analysis session is ended with a call to `end_session()`, the App Analyzer calls the `report(info_plist, app_store_info)` method of each event handler to retrieve a summary of its findings as a dictionary. Since some analysis results are based on static information (e.g., declared permissions and App Store Privacy Details), these infos are passed along to the reporting method as arguments. The retrieved results are then written to the reporting file. Details are provided in Section 5.4. Finally, the Frida Session is ended and the target app is quit. To start the analysis for the next app, a fresh App Analyzer instance is needed.

5.3.4 App Store Info Loader

The AppStoreInfoLoader is responsible for retrieving the analyzed app's published information from the App Store. It then provides this info to the analysis pipeline and also stores it in the app's analysis log folder, as described in Section 5.4. Given that there is no documented API provided by Apple to download these infos, we had to use a workaround to retrieve them, as detailed in our Methodology Section 4.2.5.

5.3.5 Network Analyzer

Network Analysis is implemented as an add-on to mitmproxy (introduced in Section 3.7). The approach we used to detect the Advertising Identifier in the analyzed app's network traffic using our custom add-on is presented in our Methodology Section 4.2.3.

Once the Network Analyzer has discovered a privacy leak (currently, the sent AdId), it informs the AppAnalyzer about the event via the ExternalEventHandler HTTP endpoint, described above in Section 5.3.2. The AppAnalyzer can then incorporate the event into the analysis report and log.

5.3.6 App Installer and Uninstaller

Before an app can be analyzed, it has to be installed on the device. Since manual installation can be a tedious task for larger studies, we automated this process. Likewise, given the limited space on the iOS device and in order to ensure that the experiment can always be repeated with a fresh installation of the to-be-analyzed app, we also implemented automated un-installation.

App Installation: Since we were interested in analyzing apps as they are published on the App Store, we needed to implement a way to download and install them. Given that this is only possible via the App Store directly, we used Appium (introduced in Section 3.8) to simulate the necessary user interactions required to install an app. The implemented process executes the following steps:

1. **Open App Store via Safari.** Using Appium, it is simple to start apps. Since opening the App Store directly and searching for an app is prone to errors (since the App Store search result may yield multiple search results, in an unpredictable order), we used Safari (the default iOS web browser) to directly open our target app in the App Store by opening the app's URL. For example, in order to open the app "GetSorted" in the App Store app, our AppInstaller constructs the App Store URL using the App Store ID provided in the analysis configuration file, in this case `https://apps.apple.com/app/id1475025504`. Using Appium, this URL is inserted into Safari's address bar. Once the URL is loaded by Safari, the App Store is automatically opened and the target app is displayed.

2. **Initiate Download.** Given that the App Store app is designed to be accessible, each button has an accessibility ID assigned to it in the background. Appium can use this ID to find the button to start the download, labeled as "get" or "re-download", using e.g., `self.driver.find_element_by_accessibility_id("get")`.

3. **Confirm Download Dialogues.** If an app is downloaded for the first time, the user is asked to confirm the installation before the download starts. Additionally, the user may be asked for their Apple ID password. In the former case, our AppInstaller simply confirms the install, in the latter case our AppInstaller uses the Apple ID password it received when it was initialized (it can be provided to the analysis toolchain using the environment-variable `APPLE_ID_PASSWORD`) and then initiates the download.

4. **Wait for Completion.** Finally, the AppInstaller waits at most 90 seconds for the download and installation to complete, by periodically looking for the button with the accessibility ID "open" to appear.

App Un-Installation: In contrast to the installation of an app, uninstalling an app is possible on a jailbroken device without having to simulate user interactions by using `ideviceinstaller`.[2] Instead of requiring the App Store ID (as it was the case with the installation), apps are uninstalled by their Bundle ID. Particularly, AppInstaller executes the following system command to uninstall apps: `os.system(f'ideviceinstaller -U {bundle_id}')`.

5.3.7 User Interaction Simulator

Given that apps normally require physical user interaction in order to trigger its various code paths, it is necessary to simulate such interactions in an automated setup like ours.

Based on two previous bachelor theses from Hannes Hauer (TU Wien) and Leon Weiß (Ruhr-University Bochum), we incorporated user interaction simulation into our platform. By communicating with Appium (introduced in Section 3.8), our platform is thus capable

[2] ideviceinstaller github page, which is part of the libmobiledevice toolkit, https://github.com/libimobiledevice/ideviceinstaller (accessed Nov. 16, 2021)

of executing touch events and controlling the analyzed app's user interface in an automated manner.

Each simulated interaction is called a *step*, and there are multiple different *exploration strategies* available. The strategies are used to determine which step is to be executed next, e.g., if there are multiple buttons visible on-screen, the strategy decides which one is tapped next:

- ExplorationStrategy: Abstract strategy only meant to be extended by more concrete strategies (below). It contains a non-implemented method `execute_next_step()` to be implemented by the specific strategy.
- iOSExplorationStrategy: Basic implementation of an iOS exploration strategy, based on `ExplorationStrategy`. Contains utility methods specific for the iOS platform, e.g., for parsing the visible iOS user interface and for assigning scores to the visible buttons on screen to determine which one to tap next. This class also provides a general implementation of the `execute_next_step()` method, which includes code to be used by all concrete exploration strategy implementations, such as login with Google, or our extension that automatically detects permission request dialogues (described below).
- iOSDFSExplorationStrategy: Concrete `iOSExplorationStrategy` implementation. Prioritizes exploring the visible screen in detail before switching to different views.
- iOSBFSExplorationStrategy: Concrete `iOSExplorationStrategy` implementation. Focuses on visiting different views.
- iOSRandomButtonExplorationStrategy: Concrete `iOSExplorationStrategy` implementation. Selects visible buttons at random.

Both the strategy and the number of step to be executed is configurable, and the app analysis session always runs until the defined maximum number of steps have been executed.

In order to be able to use the UI simulator for our privacy analysis, we needed to ensure that permission dialogues, displayed to the user by iOS, would always be accepted. Otherwise, our platform would not have been able to detect access to privacy sensitive data. To achieve this goal, we extended the `execute_next_step()` method of the base `iOSExplorationStrategy` and incorporated code capable of detecting and accepting any permission request dialogue. Since this method is invoked by all more specific strategies (subclasses), the implemented change is thus automatically active for all of them.

The implemented automatic acceptance of permission dialogues works as follows: First, at the beginning of each `execute_next_step` call we check if an alert is visible using `find_element_by_ios_predicate("type == 'XCUIElementTypeAlert'")`. Then, we check for buttons indicating a confirmation. In our tests, we noticed that system permission request dialogues always contain a button with a label that is either "ok"/"okay" or starts with "Allow" (e.g., "Allow Access to All Photos"). As a result, we check the first word of all buttons contained in the detected alert for a match to those requirements. It is important to note that it is not sufficient to, e.g., look for the word "Allow" *anywhere* in the button's label, since the text could be "Don't Allow". Once a suitable button is found, we ask the UI simulator to tap it and continue with the next step. If no alert or suitable button was found, we continue evaluating the UI based on the configured strategy.

5.4 Analysis Output – Reporting and Logging

Throughout the analysis, multiple kinds of information are tracked and logged in an effort to make the results and executed steps as transparent and back-traceable as possible. In this section, we summarize the collected data points and detail how they are reported or stored for further evaluations after the analysis.

5.4.1 Output Folder Structure and Files

Analysis results are stored in a well-structured output folder that can be specified when launching the analysis (see Section 8.2). This folder will be populated with report files for the various analyses running as part of the platform. A sample structure is provided in Listing 8.1. It contains the following files and folders:

```
output/
|- 20211103_054901_log.log
|- 20211103_054510_appium.log
|- 20211103_054516_mitmproxy.log
|- 20211103_054516_mitmproxy_ssl_keys.txt
|- 20211103_054622_traffic.pcapng
|- com.facebook.Facebook/
|  |- 20211103_062306_analysis.json
|  |- 20211103_062306_log.ndjson
|  |- app_store_info.json
|- ... further app analysis folders ...
```

Listing 5.1: Sample analysis output folder structure. All analysis files are prefixed with their creation date and time (UTC timezone). Given that apps are evaluated sequentially, this means that each app's analysis folder and the contained files are generated once their analysis starts.

- **Analysis Pipeline Log** (log.log) This file contains the log output of the pipeline itself. As such, it contains log information about the current state of the analysis, e.g., “Starting analysis of <app bundle id>”, etc.

- **Appium Log** (appium.log) Log file generated by Appium. It contains information such as executed commands (e.g., which button was tapped).
- **mitmproxy Log** (mitmproxy.log) Network traffic captured in mitmproxy’s own format. It can be analyzed/evaluated, e.g., using the command `mitmweb -rfile LOG_FILE`.
- **Captured mitmproxy TLS/SSL Secrets** (mitmproxy_ssl_keys.txt) TLS secrets captured by mitmproxy as part of the TLS handshake. Using this file, `wireshark` (or other network analysis tools) can decrypt the traffic captured by `tshark`.[3]
- **Captured tshark Network Traffic** (traffic.pcapng) Network traffic of the iOS device, captured using `tshark`.
- **App Analysis Output Folder** (for each app, named after the app’s Bundle ID), containing the following files:
  - **Analysis Report** (analysis.json) Summary of the findings during the analysis, as described below.
  - **Event Log** (log.ndjson) Event log, populated by the various EventHandlers as well as by the AppAnalyzer for events received by the ExternalEventReceiver, as described below.
  - **App Store Info** (app_store_info.json) Retrieved App Store information of the analyzed app in JSON format, retrieved by the AppStoreInfoLoader.

5.4.2 Analysis Report

The analysis report (analysis.json, example provided in the Appendix Listing 8.14) contains a summary of the findings during the analysis of each app, as well as information about the analyzed app and the analysis itself. It is populated by the AppAnalyzer and also contains the reports generated by the EventHandlers. The file contains a JSON object, which is structured as follows:

- **analysisInfo** (object) contains the start and end date of the analysis, as well as the duration in seconds. All timestamps issued by the platform use the UTC timezone.
- **appInfo** (object) contains infos about the analyzed app, such as its name, bundle ID, version and build number. Additionally, it contains all usage descriptions provided by the app, which are required to ask for permissions, as well as the declared App Store privacy labels (discussed in Section 4.2.5).

[3] https://docs.mitmproxy.org/stable/howto-wireshark-tls/ (accessed Nov. 16, 2021)

- **accessTypes** (array of objects) contains the reports of the individual EventHandlers responsible for the analyzed data types. Each report is named after the respective data type ("tracking", "photos", etc.) and contains the following fields:
  - **privacyLabel** (object) informs whether a privacy label for the respective data type was declared, for what purposes and also contains the search term that was used to determine whether it was declared or not (for transparency).
  - **accessStatusChecked** (boolean) true if the app has checked during run time if it can access the respective data type, otherwise false.
  - **accessRequested** (boolean) true if app has asked the user for access during the analysis, otherwise false.
  - **requirements** (object) informs whether the app meets all requirements necessary to be able to request permission and access the respective data type, i.e., whether it has declared the permission in the `info.plist` file and also included the relevant libraries/frameworks. This object also includes the respective usage description, if it was provided. For example, in order to access the photo library, an app is required to include the `NSPhotoLibraryUsageDescription` and include the PhotoKit framework (which means the `PHPhotoLibrary` and `PHAsset` must be present). Only if all of these requirements are met, the app is able to request access to the photo library.
  - **metrics** (object) is an object containing all additional data points collected by the respective EventHandler for the respective access type, e.g., access count, as summarized in Table 4.1.
- **logVersion** (float) clarifies the current version of the log file format (currently 1.0). Should there be any changes to the log format in the future, this value ensures that the respective parser can interpret the format correctly.

5.4.3 Event Log

Based the event-based nature of our platform, the event log (log.ndjson[4]) contains the individual events (along with their associated event data) that contributed to the results of the report. It can be used to analyze, e.g., in which order permissions were requested or (in combination with the other log files) which user interaction led to the request of a certain privacy-sensitive data point. Each log object contains an *access_type* field (which corresponds to the EventHandler that issued it), a name, timestamp (UTC), and the raw event that was sent by the Frida Script, leading to this log output. An example of such a log is presented in Listing 5.2. Additionally, the walkthrough in the Appendix Section 8.2 provides a walkthrough with details on how to evaluate the generated log files after the analysis.

[4].ndjson files contain "new-line delimited JSON", meaning that each line in the file is a JSON object, which enables a structured reporting of logged events.

```
{
   "access_type": "tracking",
   "event_name": "advertising_identifier_requested",
   "timestamp": "2021-11-03T13:47:24.778206+00:00",
   "event": {
      "name": "invocation",
      "timestamp": 1635947244.76,
      "className": "ASIdentifierManager",
      "methodName": "- advertisingIdentifier",
      "backtrace": [...]
   }
}
```


Listing 5.2: Example of an Event Log entry.

5.5 Requirements and Setup

An overview of the platform requirements is presented in the Appendix Section 8.1. Details are provided in the `README.md` file contained in the project folder.

5.6 Platform Usage

In the Appendix Section 8.2, we present a walkthrough of a typical analysis using our platform. Specifically, we demonstrate how our platform can be used to analyze an app and demonstrate how the resulting Analysis Report and Event Log can be used to show that an app has sent the Advertising Identifier without declaring the respective privacy label for it on the App Store. Further details on the usage and configuration options for the platform are provided in the `README.md` file contained in the project folder.

CHAPTER 6

Results

In this chapter, we present the results of three case studies performed by us to test the ability of our platform to verify App Store Privacy Details ("privacy labels"). Particularly, we wanted to determine whether the platform could reliably analyze multiple apps autonomously in sequence and gather all relevant information to evaluate whether or not the app's use of certain privacy-sensitive data was in line with its declared App Store privacy labels.

Overall, in accordance to our goals and methodology, we focused our work on verifying the use of the Advertising Identifier ("AdId"). Future work can then extend our approach to other privacy-sensitive data types. Specifically, we tried to achieve the following two goals with the performed case studies:

1. **Assess the accuracy and reliability of the platform:** We wanted to find out whether the platform was capable of executing the analysis of a larger number of apps in a reliable manner. This included running all target apps without interruption, reliably analyzing and populating the desired log files as well as continuously applying the dynamic analysis methods throughout the analysis.

2. **Verify the correct use or detect misuse of the Advertising Identifier in accordance with an app's declared privacy labels:** Once we were certain that the platform could reliably gather results of multiple apps in sequence, we focused our analysis on verifying the use and collection of the Advertising Identifier in real-world scenarios using our implemented dynamic analysis methods. Specifically, we wanted to compare the collection of the AdId with the developer's declared privacy labels in order to find out if our platform was capable of detecting cases where the identifier was used without declaring it beforehand on the App Store.

Given that our goal was to analyze a larger number of apps using our platform, we needed an efficient way of analyzing the various result files created by our platform for each app (see Section 5.4), rather than analyzing them manually.

To achieve this task, we created an interactive utility script capable of extracting the information of interest out of the various platform log files, either as a summary view or in CSV format. After extracting relevant data to a CSV-file, we could open it in a spreadsheet application to gain an at-a-glance overview of the results. We then manually inspected them to determine our findings, which will be presented in the following sections.

6.1 Setup and Configuration

All results presented in this chapter were gathered with the following setup:

- The analysis platform was set up as described in Section 8.1.
- Analysis Device: iPhone 6s
- iOS Version: iOS 14.3, jailbroken using checkra1n.[1]
- User Interaction Simulation: 15 interaction steps were simulated using the "iOS depth first" strategy.
- iOS App Store location: Austria

6.2 Case Study 1: Top 50 Free Apps

For our first case study, we wanted to investigate the privacy behavior of the 50 most commonly used apps. We therefore used the platform to analyze the overall top 50 free apps on the Austrian iOS App Store.

App Selection. In order to obtain the list of candidate apps, we created a utility script that used the node.js package *app-store-scraper*[2] to download the App Store info of the top 50 free apps of the Austrian iOS App Store. Using our script, we then extracted the App Store IDs along with the app name and Bundle ID and saved them, ordered by their App Store ranking, as an input file for our platform. The ranking used for the analysis was captured on Nov. 02, 2021. An excerpt of the first 10 lines in the resulting configuration file for the platform is shown in Figure 6.1.

[1] We want to stress that since the iOS 14.5 release and the introduction of the AppTrackingTransparency Framework (described in Section 3.5), apps need to explicitly ask permission before being able to access the Advertising Identifier. On the iOS version that we used, it was still *optional* to ask and instead developers could simply access it as long as the user had not disabled the identifier manually for all apps in the System Settings.

[2] https://github.com/facundoolano/app-store-scraper (accessed Nov. 6, 2021)

Top50 > top50.txt

```
1574155774 # top free 1 Grüner Pass at.gv.brz.wallet
310633997 # top free 2 WhatsApp Messenger net.whatsapp.WhatsApp
1576692816 # top free 3 GreenCheck at.itsv.mobile.cochap
389801252 # top free 4 Instagram com.burbn.instagram
544007664 # top free 5 YouTube com.google.ios.youtube
585027354 # top free 6 Google Maps - Transit & Essen com.google.Maps
835599320 # top free 7 TikTok: Videos, Lives & Musik com.zhiliaoapp.musically
447188370 # top free 8 Snapchat com.toyopagroup.picaboo
422689480 # top free 9 Gmail – E-Mail von Google com.google.Gmail
324684580 # top free 10 Spotify - Musik und Playlists com.spotify.client
```

Figure 6.1: Excerpt of the first 10 lines of our case study 1 candidate app configuration file used as the input for our platform.

After obtaining the list of apps, we run our analysis platform with the setup described above. We then gathered and extracted relevant information using our analysis utility script. An overview of the results is shown in Table 6.1. Overall, our analysis yielded the following results:

- **App Store Label:** Out of the top 50 free apps, 41 (= 82%) declared a privacy label for the Advertising Identifier on the App Store.
- **Access:** Even though a large majority of apps declared a label on the App Store, overall we only detected 23/50 apps accessing the identifier (= 46%). Out of the 41 that declared the App Store label, 23 (= 56%) also accessed it. Given that a large number of apps in this category asked for a login at startup and to accept their Terms of Service, we speculate that the number of apps accessing and potentially sending the AdId would have been higher after that. Unfortunately, handling logins and registrations is a known challenge for methodologies applying user interaction simulation.
- **Collection:** For those 23 apps who accessed the identifier, we could detect that 19 (= 82%) sent them over the network.
- **Asking for Access:** Before accessing the identifier, only 5/41 (= 12.2%) apps asked the user for explicit permission. (As mentioned above, this was still optional prior to iOS 14.5)
- **Privacy Label Misuse:** Overall, we could not detect any app out of the top 50 free apps on the Austrian iOS App Store that had accessed the Advertising Identifier without declaring it first.
- **Use of the new AppTrackingTransparency:** Interestingly, out of the 41 apps declaring the use of the AdId on the App Store, only 16 (= 39%) had also declared the new AppTrackingTransparency permission in the app, meaning that 61% of the

apps that declared use on the App Store would not have been able to access the identifier for users of iOS 14.5 or newer. However, out of those 23 who we detected accessing it, 15 (= 65%) had declared the ATT permission and could thus have accessed it on iOS 14.5 or newer after asking the user.

- **Advertising Identifier Sent Endpoints:**
  - On average, the 19 detected apps sent the identifier to 3 different endpoints (median = 3, min = 1, max = 8).
  - The most used endpoints were Google Firebase (12/19 = 63.1%), Adjust (10/19 = 52.6%) and Facebook (7/19 = 36.8%).
  - Overall, 15/19 apps (= 79%) sent the identifier to either Google or Facebook. Only 4/19 (= 21%) apps did not send it to either of them.
- **Contacted Tracking Endpoints**
  - We detected 22/50 apps (= 44%) accessing tracking domains.[3].
  - On average, these apps accessed 1.4 distinct tracking domains (median = 1, min = 1, max = 2) and made 3.2 requests (median = 3, min = 1, max = 10)
  - By far the most widely used tracking domain was "app-measurement.com" (Google Firebase), accessed by 18/22 apps (≙ 82%, or 18/50 = 36%)
  - If we also consider domains containing facebook.com as tracking,[4] then we find 24 apps accessing tracking domains overall (= 48%), where 9 of them communicated with Facebook (= 36%, or 9/50 = 18%). An overview of these findings is presented in Table 6.5.
- **Overall Network Access:**
  - In general, we detected network calls for 43/50 apps (= 86%).
  - On average, they accessed 6 distinct domains[5] (median = 6, min = 0, max = 22).

[3] Based on the NoTracking hosts block list last updated on Sept 28, 06:00:11, 2021 CEST, https://github.com/notracking/hosts-blocklists (accessed Nov. 29, 2021)

[4] We noticed during our analysis that particularly graph.facebook.com is not included in the NoTracking block list. However, since we discovered apps sending the AdId to this URL, we decided to calculate a second tracking count where we also classified communicated domains that included the "facebook.com" string as tracking.

[5] We count different subdomains on the same top-level domain as separate domains, as they may serve different purposes (e.g., ad-tracking vs. developer analytics).

#	App Name	AdId Store Label	Adid Perm. Declared	AdId Perm. Checked	Asked User	AdId Accessed	AdId Sent	Sent to Domains
1	Grüner Pass	no	no	no	no	no	no	0
2	WhatsApp	yes	no	no	no	no	no	0
3	GreenCheck	no	no	no	no	no	no	0
4	Instagram	yes	yes	yes	no	yes	yes	1
5	YouTube	yes	no	yes	no	no	no	0
6	Google Maps	yes	no	no	no	no	no	0
7	TikTok	yes	yes	yes	no	yes	yes	1
8	Snapchat	yes	yes	yes	no	yes	no	0
9	Gmail	yes	no	no	no	no	no	0
10	Spotify	yes	yes	yes	no	yes	yes	2
11	Facebook	yes	yes	yes	no	yes	no	0
12	GreenPass EU	no	no	no	no	no	no	0
13	willhaben	yes	yes	yes	yes	yes	yes	3
14	Microsoft Teams	yes	no	no	no	no	no	0
15	Netflix	yes	no	no	no	no	no	0
16	Amazon	yes	no	yes	no	yes	yes	4
17	Telegram	no	no	no	no	no	no	0
18	Google	yes	no	no	no	no	no	0
19	Joom	yes	yes	yes	yes	yes	yes	6
20	Handy-Signatur	yes	no	no	no	no	no	0
21	Vestiaire Collective	yes	yes	yes	no	yes	yes	3
22	PayPal	yes	no	no	no	yes	yes	3
23	Zalando	yes	no	no	no	yes	yes	5
24	Knowunity	yes	no	yes	no	yes	yes	2
25	Messenger	yes	no	no	no	yes	no	0
26	Amazon Prime Video	yes	no	no	no	no	no	0
27	Lieferando.at	yes	yes	yes	yes	yes	yes	4
28	Microsoft Outlook	yes	yes	yes	no	yes	yes	1
29	Klarna	yes	yes	no	no	no	no	0
30	Too Good To Go	yes	yes	yes	yes	yes	yes	2
31	George	yes	no	no	no	no	no	0
32	Google Chrome	yes	no	no	no	no	no	0
33	s Identity	yes	no	no	no	no	no	0
34	Pinterest	yes	yes	yes	no	yes	yes	1
35	Bitpanda	yes	no	yes	yes	yes	yes	2
36	Mein ELBA-App	yes	no	no	no	no	no	0
37	Signal	no	no	no	no	no	no	0
38	ÖBB	yes	no	no	no	no	no	0
39	mjam	yes	yes	yes	no	yes	yes	5
40	BestSecret	no	no	no	no	no	no	0
41	ÖBB Scotty	no	no	no	no	no	no	0
42	Microsoft Auth.	yes	no	no	no	no	no	0
43	Discord	yes	no	no	no	yes	yes	2
44	Twitter	yes	yes	no	no	yes	no	0
45	Shazam	no	no	no	no	no	no	0
46	Microsoft Word	yes	no	no	no	no	no	0
47	Untis Mobile	no	no	no	no	no	no	0
48	Gaiali	yes	no	no	no	yes	yes	3
49	Raiffeisen ELBA-pay	yes	no	no	no	no	no	0
50	CapCut	yes	yes	yes	no	yes	yes	8

Table 6.1: Case Study 1 analysis results: Declaration and use of the Advertising Identifier in the top 50 free apps overall on the Austrian iOS App Store. (Ranking as of Nov. 2, 2021; Analysis performed on Nov. 3, 2021; Advertising Identifier is represented as the "Device ID" in the App Store Privacy Details)

#	App Name	AdId Store Label	Adid Perm. Declared	AdId Perm. Checked	Asked User	AdId Accessed	AdId Sent	Sent to Domains
1	TooGoodToGo	yes	yes	yes	yes	yes	yes	2
2	Lieferando.at	yes	yes	yes	yes	yes	yes	4
3	McDonald's AT	yes	yes	no	no	yes	yes	1
4	mjam	yes	yes	yes	no	yes	yes	5
5	Burger King AT	no	yes	yes	yes	yes	yes	2
6	TheFork	yes	yes	yes	no	yes	yes	1
7	JOKR	no	yes	yes	yes	yes	yes	1
8	Thermomix	no	no	no	no	no	no	0
9	NORDSEE	no	no	no	no	yes	yes	2
10	Flink	yes	yes	yes	yes	yes	yes	3
11	HNGRY	yes	yes	no	no	no	no	0
12	Domino's AT	yes	yes	no	yes	yes	yes	1
13	Pam	no	no	no	no	no	no	0
14	KptnCook	yes	yes	yes	yes	yes	yes	4
15	HelloFresh	yes	yes	yes	yes	yes	yes	4
16	McDonald's DE	yes	yes	yes	no	yes	yes	3
17	Nespresso	yes	yes	yes	no	yes	yes	4
18	Uber Eats	yes	yes	yes	no	yes	yes	1
19	Starbucks AT	yes	yes	yes	no	yes	yes	2
20	Chefkoch	yes	yes	yes	yes	yes	yes	1

Table 6.2: Case Study 2 analysis results: Declaration and use of the Advertising Identifier in the top 20 free apps in the category "Food and Drink" of the Austrian iOS App Store. (Ranking as of Nov. 2, 2021; Analysis performed on Nov. 3, 2021; Advertising Identifier is represented as the "Device ID" in the App Store Privacy Details)

6.3 Case Study 2: Top 20 Free Apps in Food and Drink

The apps selected for our first case study were very heterogenous — they served many different use cases (shopping, reading, social media, etc.) and came from multiple categories. For our second case study, we thus wanted to investigate a set of *similar* apps. As a result, we selected the top 20 free apps out of a single category, namely "Food and Drink". These apps were all similar in the sense that they either offered users means to order food online or present cooking recipes.

App Selection. Similarly to our first case study, we again used *app-store-scraper*. This time, we used it to query the top 20 apps out of the category "Food and Drink" in the Austrian iOS App Store. The ranking used for this study was effective on Nov. 02, 2021.

An overview of the results can be found in Table 6.2; the most interesting findings of this case study are the following:

- **App Store Label:** Out of the top 20 free apps in "Food and Drink", 15 (= 75%)

declared a privacy label for the Advertising Identifier on the App Store.

- **Access:** We could detect 17/20 apps (= 85%) accessing the identifier. 14 of them had declared a label. One app declared a label but did not access the identifier.
- **Collection:** Out of those who accessed it, our tool could successfully detect that 100% of them also sent it over the network.
- **Privacy Label Misuse:** We identified 3/20 apps (= 15%) that accessed and sent the Advertising Identifier without declaring a privacy label for it on the App Store:
  1. **Nordsee** (com.nordsee.de.app): This app neither declared a privacy label on the app store nor did it already integrate the new AppTrackingTransparency permission. Thus, it accessed the Advertising Identifier without asking the user and sent it to Google Firebase.
  2. **Burger King** (at.burgerking.BurgerKingApp): This app did not declare the privacy label on the App Store. Interestingly, though, the app already adapted the AppTrackingTransparency permission in the app and also asked the user for access during run time. We then detected the app sending the identifier to both Google Firebase and Facebook Graph.
  3. **Jokr Delivery** (com.jokr.delivery): Similarly to the Burger King app, this app did not declare the privacy label but asked for access during run time. It then sent the identifier to Adjust.
- **Asking for Access:** 9/17 apps (= 52.9%) actively asked the user for access to the Advertising Identifier.
- **Use of the new AppTrackingTransparency:** Out of those 17 apps who sent the identifier, all except for the Nordsee app (16/17 = 94%) had already declared the new AppTrackingTransparency permission for using the Advertising Identifier. Overall, 17/20 (=85%) declared the new permission.
- **Advertising Identifier Sent Endpoints:**
  - On average, the 17 detected apps sent the identifier to 2.4 different endpoints (median = 2, min = 1, max = 5).
  - The most used endpoints were Google Firebase (13/17 = 76.4%), Facebook (10/17 = 58.8%) and Adjust (4/17 = 23.5%).
  - 16/17 apps (= 94%) sent the identifier to either Google or Facebook.
- **Contacted Tracking Endpoints**
  - We detected 17/20 apps (= 85%) accessing tracking domains.
  - On average, these apps accessed 1.3 distinct tracking domains (median = 1, min = 1, max = 4) and made 3.2 requests (median = 3, min = 1, max = 10)

- – As with the top 50 apps in our first case study, by far the most widely used tracking domain was "app-measurement.com" (Google Firebase), accessed by 15/17 apps ($\hat{=}$ 88%, or 15/20 = 75%).
  – If we also consider domains containing facebook.com as tracking, then we find 18 apps accessing tracking domains overall (= 90%), where 12 of them communicated with Facebook (= 67%, or 12/20 = 60%). An overview of these findings is presented in Table 6.5.

- **Overall Network Access:**
  – All 20 apps made network calls.
  – On average, they accessed 10 distinct domains (median = 10, min = 3, max = 19).
  – All 20 apps (= 100%) communicated with either Facebook or Google: 19 (= 95%) communicated with Google, 12 (= 60%) with Facebook.

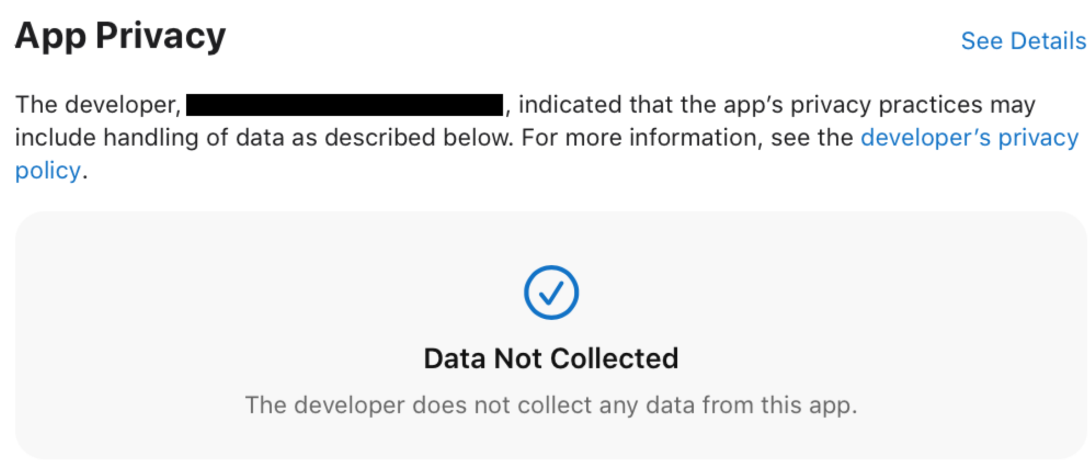

Figure 6.2: Example App Store Privacy Details declaration of an app that claims to not collect any data.

6.4 Case Study 3: 50 Popular Apps Declaring “No Data Collected”

For our final case study, we wanted to investigate the privacy behavior of a very specific group of apps, namely those who *explicitly* declared on the App Store not to collect *any* privacy-sensitive data. Figure 6.2 shows an example of such a declaration on the App Store. Specifically, we wanted to evaluate the trustworthiness of such declarations by analyzing the use of the Advertising Identifier of 50 “popular”[6] free apps featuring such a declaration.

App Selection. Qualifying apps for this analysis were determined in the following way: First, we used *appstore_crawler*[7] to download the metadata of popular apps from the US iOS App Store. We then filtered these apps for being free and for declaring not to collect any data. From the resulting list, we used the first 50 apps that were available on the Austrian App Store.

An overview of the results is presented in Table 6.3. We summarize our findings as follows:

- **App Store Label:** None of the apps in the target group declared to collect the Advertising Identifier. In fact, all of them declared “Data Not Collected: The developer does not collect any data from this app”.

[6] When browsing the various app categories (Games, Navigation, Books, etc.) on the App Store website, each such category page shows a list of “popular” apps, e.g.: https://apps.apple.com/us/genre/ios-photo-video/id6008 (accessed Nov. 6, 2021)

[7] https://gitlab.com/marzzzello/appstore_crawler (accessed Nov. 7, 2021)

- **Access:** We detected that 15/50 apps (= 30%) accessed the Advertising Identifier.
- **Collection:** Furthermore, we could identify that 12 of the 15 apps (=80%) sent the identifier over the network after accessing it.
- **Privacy Label Misuse:** We identified 12/15 apps (= 24%) that accessed and sent the Advertising Identifier without declaring a privacy label for it on the App Store.
- **Asking for Access:** None of the apps asked users for permission to use the identifier.
- **Use of the new AppTrackingTransparency:** Only 1 of the 50 apps had declared the AppTrackingTransparency permission in the app. Thus, none of the apps except for one would have been able to access the identifiers for users running iOS 14.5 or newer.
- **Advertising Identifier Sent Endpoints:**
  - On average, the 12 detected apps sent the identifier to 1.5 different endpoints (median = 1, min = 1, max = 3)
  - The most used endpoints were Google Firebase (7/12 = 58.3%) and Facebook (5/12 = 41.6%)
  - If the Advertising Identifier was sent, 11 of the 12 apps (= 91.6%) sent it to either Google or Facebook.
- **Contacted Tracking Endpoints**
  - We detected 17/50 apps (= 34%) accessing tracking domains.
  - On average, these apps accessed 1.7 distinct tracking domains (median = 2, min = 1, max = 4) and made 6.1 requests (median = 3, min = 2, max = 51)
  - Again, the most widely used tracking domain was “app-measurement.com” (Google Firebase), accessed by 13/17 apps ($\triangleq$ 76%, or 13/50 = 26%)
  - If we also consider domains containing facebook.com as tracking, then we find 21 apps accessing tracking domains overall (= 42%), where 5 of them communicated with Facebook (= 24%, or 5/50 = 10%). An overview of these findings is presented in Table 6.5.
- **Overall Network Access:**
  - 36/50 apps (= 72%) made network requests.
  - On average, they accessed 2.5 distinct domains (median = 1, min = 0, max = 18).

App Name	AdId Store Label	Adid Perm. Declared	AdId Perm. Checked	Asked User	AdId Accessed	AdId Sent / # Domains
WorkJam	no	no	no	no	no	no
MathPapa Algebra Calc.	no	no	no	no	no	no
Swimsuits & Sports Mag.	no	no	no	no	yes	yes / 3
Wyoming 511	no	no	no	no	no	no
Filmmaker Pro Video E.	no	no	no	no	no	no
Cymath Math Prob. S.	no	no	no	no	yes	yes / 1
Northern Light Aurora F.	no	no	no	no	no	no
Hair Color Changer	no	no	no	no	yes	yes / 2
Samsung Galaxy Watch	no	no	no	no	no	no
Flymaps	no	no	no	no	no	no
Windscribe VPN	no	no	no	no	no	no
Weather Widget & R. F.	no	yes	yes	no	yes	yes / 2
Model Society	no	no	no	no	yes	yes / 3
HappyLighting	no	no	no	no	no	no
Love Tester & Calc.	no	no	no	no	yes	no
Java Recipes	no	no	no	no	no	no
Python Recipes	no	no	no	no	no	no
Logo Maker	no	no	no	no	no	no
HS Team App	no	no	no	no	yes	yes / 1
MYmta	no	no	no	no	no	no
Tide Alert (NOAA)	no	no	no	no	no	no
Official Navy PFA	no	no	no	no	no	no
Python3IDE	no	no	no	no	no	no
SHIB, NFT	no	no	no	no	no	yes / 1
com.gov.mfa.wjb12308	no	no	no	no	yes	no
Recipes for Swift	no	no	no	no	no	no
Hello Neighbor	no	no	yes	no	yes	yes / 2
App Icon Maker	no	no	no	no	no	no
Nautide	no	no	no	no	yes	no
Thumbnail Erst. F. YT	no	no	no	no	yes	yes / 1
HryFine	no	no	no	no	yes	yes / 1
Clockology	no	no	no	no	no	no
Skullcandy	no	no	no	no	no	no
Find IP • Public & WiFi	no	no	no	no	yes	no
a-Shell	no	no	no	no	no	no
Slender: The Arrival	no	no	no	no	no	no
ColorMax	no	no	no	no	no	no
Retro Bowl	no	no	no	no	no	no
Text Editor.	no	no	no	no	no	no
Secret Neighbor	no	no	no	no	no	no
Talon	no	no	no	no	no	no
Randonautica	no	no	no	no	no	no
Gruppe für Telegram	no	no	no	no	yes	yes / 1
Silhouette Go	no	no	no	no	no	no
myAdMob	no	no	no	no	yes	yes / 1
Doodle Drawing Pad	no	no	no	no	no	no
Bluetooth Inspector	no	no	no	no	no	no
Fonts for Cricut	no	no	no	no	no	no
A Guide To Crystals	no	no	no	no	no	no
Pregnant Mom & Baby	no	no	no	no	no	no

Table 6.3: Case Study 3 analysis results: Declaration and use of the Advertising Identifier in 50 popular free apps declaring “No Data Collected” on the Austrian iOS App Store. (Ranking as of Nov. 2, 2021; Analysis performed on Nov. 3, 2021; Advertising Identifier is represented as the "Device ID" in the App Store Privacy Details)

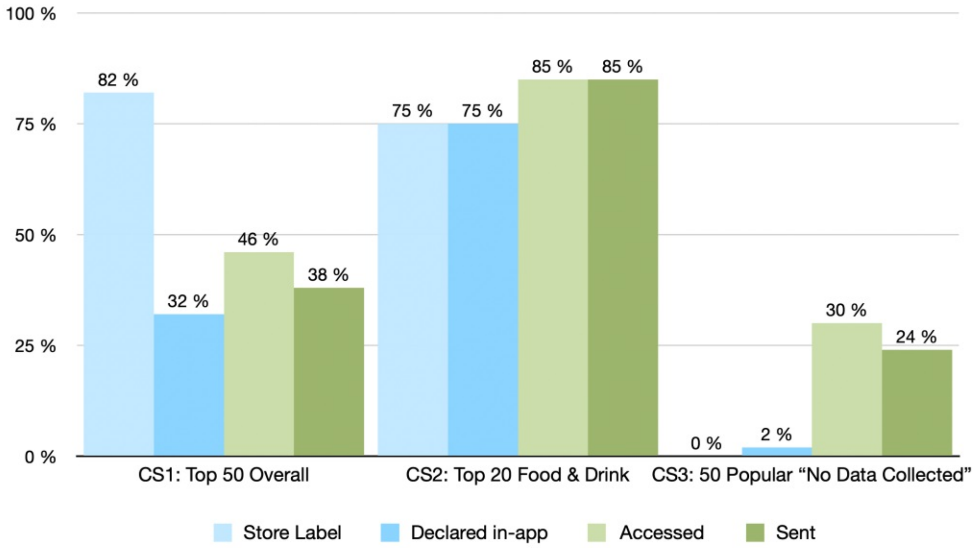

Figure 6.3: Overview of the Advertising Identifier use across our three case studies. Additional details are presented in Table 6.4.

6.5 Discussion

Overall, we could see that the use of the Advertising Identifier was widespread among the various apps investigated by us. Our platform performed well when analyzing multiple apps in sequence and reliably conducted the analysis in an automated way. A summary of our evaluations across the three case studies is presented in Table 6.4 and Figure 6.3.

In total, based on the 120 apps analyzed in our case studies, we come to the following conclusions:

- Throughout our study, we could see that a *present* privacy label for the Advertising Identifier on the App Store could indeed be seen as a good indicator of what the user can expect from the run time behavior of that app. On average, 70% of apps accessed and collected the identifier at run time if they had declared it on the App Store.
- Additionally, we could see that *accessing* the identifier also resulted in *sending* it in the majority of cases. Specifically, after accessing the AdId, our platform was able to detect that 85% of apps sent it afterwards, on average.
- What's more, we could see that if an app has even declared the new App Tracking Transparency permission, chances were very high for the identifier to get accessed (average of 96%) and our platform has shown that it can detect *collection* of it in 90% of those cases, on average.

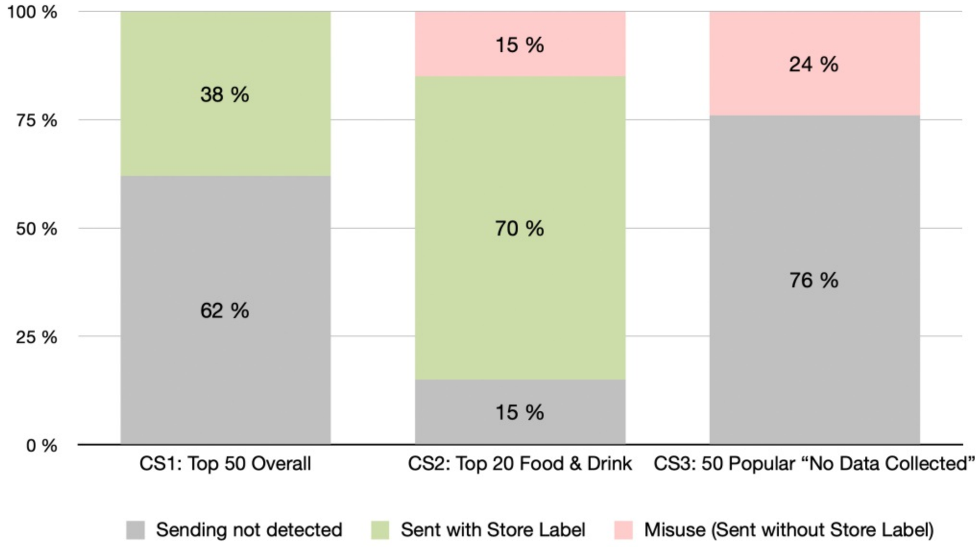

Figure 6.4: Sending behavior of the Advertising Identifier discovered with our platform.

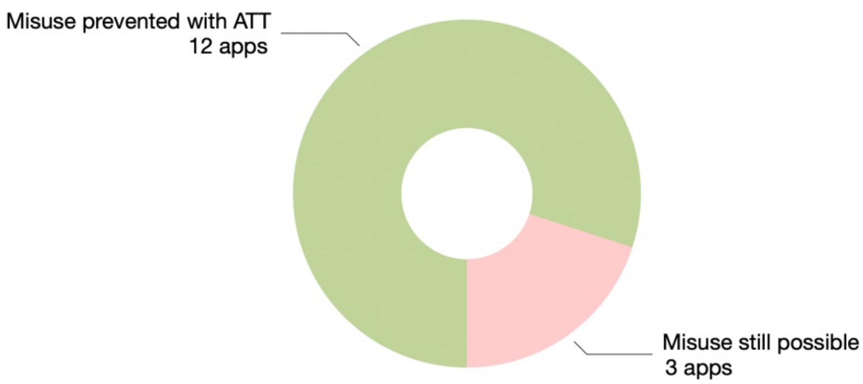

Figure 6.5: Accessing and sending the Advertising Identifier would have been prevented by the new App Tracking Transparency policy in 80% of our discovered cases, since 12 out of the 15 apps had not declared the new permission in the app and would thus not have been able to ask the user for permission in recent versions of iOS ($\geq$ 14.5).

- Across our three case studies, we discovered 15/120 apps (= 12.5%) accessing and sending the Advertising Identifier without the "Device ID" privacy label on the App Store (Figure 6.4). Importantly, in the group that explicitly declared not to collect any data, at least 24% of them still did.

- However, we could also see that in 12 out of the 15 cases of misuse (= 80%), apps did not declare the App Tracking Transparency permission in the app, thus they would not have been able to access and send the AdId in recent versions of iOS ($\geq$ 14.5). This shows that the new system, requiring developers to explicitly ask for permission before being able to use the identifier, is very effective in preventing access in cases where the user (or even the developer implementing third-party libraries) may not expect it — a major leap for user privacy (Figure 6.5). Specifically, misuse would have been prevented in 80% of our discovered cases in total.

- Overall, we discovered that more than half of all evaluated apps contacted tracking domains (54% on average, or 60% if we also count Facebook), as summarized in Table 6.5. Given that in the homogenous set of case study 2 85% of apps communicated with tracking domains (90% incl. Facebook), we speculate that the amount of tracking may be highly dependent on the *kind* or category of an app. Importantly, we saw that *if* a tracking domain was accessed, it was Google Firebase in 82% of cases, yielding an average dominance of 46% across our studied apps. Likewise, Facebook was accessed in 43% of such cases, giving it a total presence of 29%.

- Finally, it is worth noting that Google and Facebook have shown a strong dominance when it comes to collection of the Advertising Identifier, too. If it was sent, in 88% of cases it was sent to either Facebook or Google. This shows the unparalleled insights these companies have about the user interactions *across* the most used or popular apps on the App Store, since they can effectively link and cross-reference user interactions by receiving the same identifier from multiple apps.

We are responsibly disclosing our findings with the developers and the App Store.

	CS1: Top 50 Free Overall	CS2: Top 20 Food & Drink	CS3: 50 Popular “No Data Collected”	Avg.
Store Label Total	82 %	75 %	0 %	52 %
Declared Total	32 %	75 %	2 %	36 %
Access Total	46 %	85 %	30 %	54 %
Sent Total	38 %	85 %	24 %	49 %
Label → Declared	39 %	100 %	N/A	70 %
Label → Access	56 %	93 %	N/A	75 %
Label → Sent	46 %	93 %	N/A	70 %
¬Label → Sent	0 %	15 %	24 %	13 %
Declaration → Access	94 %	94 %	100 %	96 %
Declaration → Sent	75 %	94 %	100 %	90 %
User Asked for Access	12 %	53 %	0 %	22 %
Misuse prevented (Fig. 6.5)	N/A	33 %	92 %	63 %
Access → Sent	83 %	100 %	73 %	85 %
Sent to Google	63 %	76 %	58 %	66 %
Sent to Facebook	37 %	59 %	42 %	46 %
Sent to Google or FB	79 %	94 %	92 %	88 %

Table 6.4: Summary of the Advertising Identifier use across three performed case studies. “Store Label” means that an app has declared the “Device ID” privacy label on the App Store. “Declared” refers to the fact that the app has declared the AppTrackingTransparency permission in the `info.plist` file. Note that from iOS 14.5 onwards, the identifier can only be accessed iff ATT permission has been declared and the user has been asked. The implication symbols (“→”) can be read as “resulted in”, e.g., “Label → Access” can be read as “Label resulted in Access” or more specifically “Given that a Store Label was present, for what percentage did it also result in access?”. The “¬“ symbol means “not”.

	CS1: Top 50 Free Overall	CS2: Top 20 Food & Drink	CS3: 50 Popular “No Data Collected”	Avg.
Tracking Domains	44 %	85 %	34 %	54 %
thereof Google Firebase	82 %	88 %	76 %	82 %
Tracking Domains w/ Facebook	48 %	90 %	42 %	60 %
thereof Facebook	38 %	67 %	24 %	43 %
Total Google Firebase	36 %	75 %	26 %	46 %
Total Facebook	18 %	60 %	10 %	29 %

Table 6.5: Overview of accessed tracking domains, based on the NoTracking Block List as of Sept 28, 06:00:11, 2021 CEST. Google Firebase refers to the app-measurement.com domain. Since we found some Facebook domains (particularly graph.facebook.com) that were not included in the block list, even though we discovered multiple cases where the AdId was sent to graph.facebook.com, we also provide a second tracking count where we also classified domains containing “facebook.com” as tracking.

CHAPTER 7

Conclusion

7.1 Summary

In this thesis, we combined multiple state-of-the art methodologies into an analysis platform capable of autonomously investigating the run time privacy behavior of iOS apps. Specifically, the platform handles the installation, analysis, logging, reporting and uninstallation of a configurable target set of apps with minimal interaction required by the researcher. During the analysis, it uses API Method Call Interception to detect privacy-sensitive data access and use, network traffic analysis using a machine-in-the-middle proxy, and user interaction simulation to make the analysis scalable by eliminating the need for a physical user interacting with the apps during the process. In summary, our platform is capable of analyzing the use of the following data types: contacts, camera, user location, microphone, photos, network requests and tracking.

Additionally, we investigated how such dynamic analysis methods may be used to verify the recently introduced App Privacy Details ("privacy labels") on the App Store, which aim to offer users unprecedented insights into *how* developers use collected privacy-sensitive user data behind the scenes. After presenting and discussing Apple's guidelines regarding the declaration of those labels, we argued that a verification process needs to be able to detect *collection* of the sensitive data, which includes transmitting the respective data type off the device.

Given the large number of available labels, we focused our attention on verifying the "Device ID" label, which developers need to declare if they collect the Advertising Identifier ("AdId"). In order to achieve this goal, we added the capability to our platform to download an app's declared labels, parse them and incorporate them in the privacy report. Then, we extended our proxy with an add-on to intercept network requests and search for the AdId.

We evaluated the real-world performance of our platform by conducting three case studies with the following target app sets: 1) Top 50 free apps of the Austrian App Store, 2) Top 20 free apps in the category "Food & Drink" and 3) 50 popular apps that claim not to collect *any* data. The platform managed to reliably analyze these 120 apps in an automated way. Overall, we could see that the use of the Advertising Identifier was widespread and that our platform could verify a *declared* privacy label for it in 70% of cases. Most importantly, we also discovered that 13% of our tested apps collected it *without* declaring a label. For those apps that declared not to collect *any* data, we discovered that at least 24% of them still did.

Finally, we conclude that our platform has delivered promising results indicating that a verification process for privacy labels using dynamic analysis methods is indeed possible. Given that we could already discover a non-negligible number of apps who *should have* declared a privacy label, we feel like such a verification process is highly necessary to ensure that users do not lose their trust in this system in the future. In order to gain a more complete picture, future work extending the number of verified labels is needed.

7.2 Limitations and Future Work

The current implementation of the platform is mostly limited in two ways: First, the network analysis is limited in the sense that it does not detect the Advertising Identifier if it is obfuscated or encrypted. Second, the analysis results are somewhat limited by the intelligence of the user interaction simulator, given that, e.g., many apps require logins at startup or have web links that, if opened, result in the automator interacting with the web page instead of the app. While these two points are limitations, they are inherent issues of dynamic analysis methods in general and thus to be expected. Their impact can be reduced by iteratively improving these parts to make them more resilient over time.

Apart from that, the created analysis platform can act as the foundation for various future research areas. Future work could focus on improving the platform itself or extending its analysis capabilities. Improving the platform could involve making it more configurable, easier to use, improving/streamlining the data output, or developing tools to process and display the analysis results.

Extending the platform could include improvements to the privacy label verification process, i.e., extend the network analysis in order to detect the collection of more privacy-sensitive data types (e.g., detecting sent contact information or user photos)

Finally, another option would be to add more breadth or depth to the dynamic analysis. Broadening the analysis could include adding event handlers for additional data types (e.g., for health data access), while adding more depth could involve creating more advanced interceptors, e.g., detecting which specific contact infos are accessed or possibly using Frida Stalker[1] to evaluate data flow in the app (e.g., investigate *how* the accessed data is used).

[1] https://frida.re/docs/stalker/ (accessed Nov. 23, 2021)

CHAPTER 8

Appendix

8.1 Requirements and Setup

Below we provide a broad overview of the requirements for running an analysis with our platform. **Detailed setup instructions including references and commands can be found in the README.md file in the project folder.**

- Hardware
  - Mac running macOS (tested on Big Sur up until v11.6)
  - Jailbroken iPhone (tested with checkra1n jailbreak until v0.12.4 beta on an iPhone 6s running iOS v14.3)
- macOS software
  - Local copy of the analysis platform project folder
  - Python 3.5 or newer
  - Xcode
  - Apple Developer account
  - Appium (+ set up for iOS)
  - mitmproxy (custom install, see README.md)
  - wireshark and tshark
  - libmobiledevice
  - Python virtual environment set up, with Python dependencies installed
- iOS software

  - Jailbroken
  - Cydia (usually automatically installed by jailbreaking tool)
  - Frida
  - Signed in to iCloud account capable of downloading apps

8.2 Platform Walkthrough

In this section, we provide a walkthrough of a typical analysis workflow when using our platform. First, we briefly discuss the platform's file structure and configuration options. Then, we go through the required commands, the configuration and the resulting outputs of a sample analysis of the Burger King Österreich app.

8.2.1 File Structure

The platform is structured in the following way:

```
platform/
|- README.md
|- requirements.txt
|- start_appium.sh
|- start_mitmproxy.sh
|- start_tshark.sh
|- main.py
|- pipeline.py
|- analysis/
|  |- AppAnalyzer.py
|  |- AppStoreInfoLoader.py
|  |- constants.py
|  |- model.py
|  |- loggers.py
|  |- events/
|  |  |- FridaEventHandler.py
|  |  |- ExternalEventReceiver.py
|  |  |- handlers/
|  |  |  |- CameraInvocationHandler.py
|  |  |  |- ContactsInvocationHandler.py
|  |  |  |- LocationInvocationHandler.py
|  |  |  |- MicrophoneInvocationHandler.py
|  |  |  |- NetworkInvocationHandler.py
|  |  |  |- PhotosInvocationHandler.py
|  |  |  |- TrackingInvocationHandler.py
|  |  |  |- SimpleInvocationHandler.py
|  |- frida_scripts/
```

```
| | | |- main.js
| | | |- util.js
| |- assets/
| | | |- dnsmasq.blacklist.txt
|- proxy/
| |- NetworkAnalyzer.py
| |- get_advertising_identifier.py
|- ui_automation/
| |- README.md
| |- requirements.txt
| |- AppSimulator.py
| |- AppInstaller.py
| |- exploration_session.py
| |- exploration_strategies.py
| |- exploration_benchmarks.py
```

Listing 8.1: Overview of the platform's folder structure.

The most important files or folders are the following:

- `README.md` contains a detailed list of the platform's requirements, setup instructions and usage infos.
- `start_appium.sh`, `start_mitmproxy.sh` and `start_tshark.sh` can be used to conveniently start the external platform components.
- `main.py` is the main entry point into the platform. Details are provided in the `README.md` file and a usage example is presented below in Section 8.2.3.
- The `analysis` folder consists of all code related the dynamic analysis with Frida, i.e., the `AppAnalyzer`, `AppStoreInfoLoader`, various Frida event handlers, and the `ExternalEventReceiver`.
- The `proxy` folder contains the `NetworkAnalyzer` and a helper Python script that retrieves the device's Advertising Identifier using Frida, since the NetworkAnalyzer requires this information to be able to find it on the traffic.
- The `ui_automation` folder bundles all Appium code, i.e., the `AppInstaller` for automated installation and the `AppSimulator`, which manages the UI exploration.

8.2.2 Platform Configuration File

The list of to-be-analyzed apps needs to be provided to the platform when launching it as a text configuration file. It needs to contain a list of newline-separated App Store IDs.

Lines that are blank or start with a hash symbol (#) will be ignored and can be used for comments or to add structure to the file. Additionally, each line can contain a comment after the App Store ID, which may be used to add more info about the app, e.g., its bundle ID. An example of a valid configuration file is shown in Listing 8.2.

Once run, the platform will process the provided apps in sequence, one by one. The `README.md` contains further information on the configuration of the platform.

```
# --- Top 2 Free - FOOD_AND_DRINK:
362764499 # FOOD_AND_DRINK, 1, Lieferando.at, lieferservice.at
1060683933 # FOOD_AND_DRINK, 2, Too Good To Go, com.moonsted.TGTG

# --- Top 2 Free - NEWS:
333903271 # NEWS, 1, Twitter, com.atebits.Tweetie2
1064216828 # NEWS, 2, Reddit, com.reddit.Reddit
```

Listing 8.2: An example of a platform configuration file, specifying the apps to be analyzed.

8.2.3 Platform Usage Example

In this section, we provide a specific example of a typical workflow when using the platform, including a discussion of commands and outputs. Detailed information about the platform usage can be found in the `README.md` file.

Goal. In our sample analysis, we want to investigate the privacy behavior of the Burger King Österreich app. For this purpose, we retrieve the App Store ID from its App Store URL (https://apps.apple.com/at/app/burger-king-österreich/id**1574333332**) and create the following example configuration file:

```
1574333332 # Burger King AT
```

Listing 8.3: Simple configuration file for the Burger King Österreich App

Steps Outlook. In total, the analysis consists of three dedicated steps: 1) automated installation of target apps, 2) automated analysis and 3) inspection of the results (output files). Interaction by the researcher is only required before the start of each analysis part and for the inspection of the results at the end. The app installation and analysis itself - which takes the majority of time - is executed autonomously.

Before we start with each step, we make sure that our Python virtual environment (created during the platform setup) has been activated using

```
source <PATH_TO_VENV>/venv-python3.9/bin/activate
```

1. Automated Installation of All Target Apps:

In order to install the target app(s) for our analysis using our platform, we make sure our jailbroken analysis iPhone is connected via USB and unlocked.

Then, start the Appium server using the following line, receiving the following output:

```
(venv-python3.9) -> ./start_appium.sh
Starting appium
Settting log output file to out/20211122_231354_appium.log
[Appium] Welcome to Appium v1.22.0
[Appium] Non-default server args:
[Appium] logFile: out/20211122_231354_appium.log
[Appium] Appium REST http interface listener started on 0.0.0.0:4723
```

Listing 8.4: Sample Appium start output.

In a separate terminal, we run the platform in install-only mode using the following command:

```
APPLE_ID_PASSWORD="pass" python main.py -xcode-org-id 9B00000024
-install-only burgerking_config.txt
```

As shown, we need to provide the password for the device's Apple ID as an environment variable so that our installation script can provide it if asked by iOS. Additionally, we need to provide the *Xcode Organization ID*, identifying which Apple Developer account Appium should use to compile and install its own helper app used to control the user interface.

We receive the following output:

```
(venv-python3.9) -> APPLE_ID_PASSWORD="password here" python main.py
      --xcode-org-id 9B00000024
      --install-only
      burgerking_config.txt
[INFO] Starting up, config:
            install? True, analyze? False, uninstall? False
[INFO] starting analysis of 1 apps
[INFO] starting to serve on :8042...
[INFO] -- Installing app with App Store id 1574333332
      on device with udid=auto:
[INFO] connecting to appium at http://localhost:4723/wd/hub
      (the first time may take a while and "socket hung up"
      messages in appium log are normal)...
[INFO] opening safari...
[INFO] waiting to type url...
[INFO] typing app url...
[INFO] looking for download button...
[INFO] tapping download button...
[INFO] no additional dialogue
[INFO] waiting for download to complete...
[INFO] download in progress...
```

```
[INFO] download in progress...
[INFO] download in progress...
[INFO] download complete.
[INFO] installed app bundle id: at.burgerking.BurgerKingApp
[INFO] tasks succeeded for app 1574333332
[INFO] shuttung down server
[INFO] No more tasks to execute.
[INFO] 1 apps succeeded:
[INFO] - 1574333332
[INFO] 0 apps failed:
```

Listing 8.5: Sample platform installation output. Each log line also contains a UTC timestamp (e.g., "2021-11-23 00:01:06,794"), which has been omitted in all sample log outputs to conserve space.

2. Automated Analysis:

Once the installation of all target apps is complete, the actual automated analysis using our platform can begin.

In addition to Appium, we now also start our proxy, start capturing network traffic with tshark and then start our platform.

First, we make sure the iPhone is unlocked and then **start mitmproxy**:

```
(venv-python3.9) -> ./start_mitmproxy.sh auto
Retrieving Advertising Identifier
to configure NetworkAnalyzer (device_udid=auto)...
Retrieved Advertising Identifier: 7D6C4C70-DEMO-DEMO-DEMO-C56A
Starting mitmproxy (mitmweb)
Using add-on to parse network traffic: proxy/NetworkAnalyzer.py
Settting output file to out/20211123_001627_mitmproxy.log
Logging SSL Keys to out/20211123_001627_mitmproxy_ssl_keys.txt
Hint: Afterwards, you could use the following command
to browse and analyze the intercepted
network traffic: mitmweb --rfile out/20211123_001627_mitmproxy.log

Web server listening at http://127.0.0.1:8081/
Loading script proxy/NetworkAnalyzer.py
NetworkAnalyzer is starting, configured
to look for Advertising Identifier "7D6C4C70-DEMO-DEMO-DEMO-C56A"
Proxy server listening at http://*:8080
```

Listing 8.6: Sample mitmproxy start output.

As shown, during startup, our script launches a simple Frida session to retrieve the AdId from the connected iPhone. Once received, it is used to configure the NetworkAnalyzer with it. Given that only one USB-device was connected, we could simply pass ’auto’ instead of the device UDID to the `start_mitmproxy.sh` script.

Once the proxy is running, we configure it in the Wi-Fi settings of the currently connected iPhone. There, we set the proxy setting to the IP address of the computer running mitmproxy and the port to 8080. To verify that the proxy is configured properly, we open any URL in Safari (e.g., https://mitm.it) and check for corresponding output in the terminal.

Next, in a separate terminal, **we additionally start tshark** after looking up the UDID of the connected iPhone and creating a dedicated virtual network interface for it:

```
(venv-python3.9) -> idevice_id
0000DEMO-NETWORKID (Network)
7007123d4cDEMODEMODEMODEMO41681fe7931bc (USB)

(venv-python3.9) -> rvictl -s 7007123d4cDEMODEMODEMO41681fe7931bc
Starting device 7007123d4cDEMODEMODEMO41681fe7931bc [SUCCEEDED]
with interface rvi0

(venv-python3.9) -> ./start_tshark.sh rvi0
Starting tshark with network interface rvi0
Settting output file to out/20211123_002730_traffic.pcapng
Capturing on ’rvi0’
```

Listing 8.7: Sample tshark start output.

As a quick recap, we currently have Appium, mitmproxy and tshark running at the same time. Additionally, the target apps are installed on the device (in this case, only the Burger King app).

Now we can finally **start our analysis with out platform**, which will analyze each app in the provided configuration file one by one.

As the output during the analysis for the one app had 1922 lines, we omit large sections of it and only discuss the most interesting parts.

In the first section (see below), we start the platform in “analysis” mode: After connecting to the device, spawning the app and attaching to it, it starts attaching the interceptors (“hooks”). Then, it loads a local copy of the NoTracking blocklist[1] file from the assets folder, which contains a large number of known tracking URLs, which the platform later uses to classify URL calls as tracking. In the last line, the platform informs that it will now wait for 5 seconds before starting the UI simulation, giving the app enough time to initialize its state and user interface.

```
venv-python3.9) -> python main.py
```

[1] https://github.com/notracking/hosts-blocklists (accessed Nov. 22, 2021)

```
2        --xcode-org-id 9B00000024 burgerking_config.txt
3  [INFO] Starting up, config:
4        install? False, analyze? True, uninstall? False
5  [INFO] starting analysis of 1 apps
6  [INFO] starting to serve on :8042...
7  [INFO] successfully wrote app store info to
8        out/at.burgerking.BurgerKingApp/app_store_info.json.
9  [INFO] analysis log file opened:
10       out/at.burgerking.BurgerKingApp/20211123_013049_log.ndjson
11 [INFO] found usb device with UUID: 7007123d4cDEMO41681fe7931bc
12 [INFO] spawning app with bundle id: at.burgerking.BurgerKingApp...
13 [INFO] attaching to app (pid: 1573)...
14 [INFO] successfully attached
15 [INFO] listening for ATT...
16 [INFO] listening for photos access...
17 [INFO] listening for contacts access...
18 [INFO] listening for camera access...
19 [INFO] listening for microphone access...
20 done hooking method:
21 ObjC.classes["ASIdentifierManager"]["- advertisingIdentifier"]
22 done hooking method:
23 ObjC.classes["ATTrackingManager"]["+ trackingAuthorizationStatus"]
24 done hooking method:
25 ObjC.classes["ATTrackingManager"]
26       ["+ requestTrackingAuthorizationWithCompletionHandler:"]
27 // ... many further hooking calls omitted (interception setup)
28 hooking audio/video methods...
29 [INFO] Resuming App Execution
30 reading info.plist
31 [INFO] load known tracking urls from /assets/dnsmasq.blacklist.txt
32 [INFO] 250073 known tracking urls loaded
33 [INFO] dynamic analysis started, waiting 5s
34       before starting UI simulation to ensure app init
```

Listing 8.8: First part of the analysis output: Initialization of the platform.

Later on, below, our platform logs that it has discovered a new NSURLSession dataTask (= a network request) to the Facebook Graph API. Additionally, line 41 shows that ExternalEventReceiver has received a call at its endpoint `/event/network_ad_id`, indicating that the NetworkAnalyzer has discovered the AdId in the network traffic. The individual events with details about them are logged in the Event Log, which we will investigate below in the analysis step.

```
35 NSURLSession - dataTaskWithRequest:completionHandler:url:
36 URL: https://graph.facebook.com/v8.0/196055407227250/activities
```

```
37 METHOD: POST
38 [INFO] non-tracking domain accessed - graph.facebook.com
39 [INFO] analysis report updated at
40 out/at.burgerking.BurgerKingApp/20211123_013049_analysis.json
41 127.0.0.1 "POST /event/network_ad_id"
```

Listing 8.9: Second part of the analysis output: The platform discovered the Advertising Identifier in the network traffic.

After 5 seconds, the UI simulation begins. In step 4, the platform discovers a request to a known tracking URL (line 55), followed by a notice from the NetworkAnalyzer that the AdId was found (line 58). Furthermore, the platform discovers an AppTrackingTransparency system dialogue asking the user for tracking permission (line 59). It then looks for the allow button and confirms the dialogue:

```
42 [INFO] simulation starting
43 (INFO) ExplorationSession: Starting session
44 [INFO] Starting session
45 [INFO] Starting exploration for 10 steps
46 [INFO] Starting exploration step 1
47 (INFO) StepTimerExplorationBenchmark: Step 1 took 2.043351158s
48 [INFO] Step 1 took 2.043351158s
49 // ... step 2 and 3 omitted
50 [INFO] Starting exploration step 4
51 TASK WITH REQUEST:
52 NSURLSession - uploadTaskWithRequest:fromData:url:
53 URL: https://app-measurement.com/a
54 METHOD: POST
55 [INFO] tracking domain accessed - app-measurement.com
56 [INFO] analysis report updated at
57       out/at.burgerking.BurgerKingApp/20211123_013049_analysis.json
58 127.0.0.1"POST /event/network_ad_id HTTP/1.1"
59 [INFO] found alert: Allow "BurgerKing" to track
60       your activity across other companies' apps and websites?
61 [INFO] inspecting alert buttons...
62 [INFO] found accept button with label 'Allow'
63 [INFO] Tapping alert accept button
```

Listing 8.10: Third part of the analysis output: The user interaction simulation begins. In step 4, the platform discovers a request to a tracking domain, a sent AdId, and an AppTrackingTransparency tracking request dialogue.

After all UI simulation steps have been completed, the platform ends the analysis session for the current app. If there would have been another app in the config file, the platform would have continued with the same workflow until all apps would have been processed.

Finally, the output presents a summary of the succeeded and failed apps. Given that the platform continues with the analysis even if some apps fail, such an overview is important so that the researchers can investigate the failed apps and re-attempt their analysis.

```
[INFO] Step 10 took 1.1419819380000007s
(INFO) ExplorationSession: Quitting Appium session
[INFO] Quitting Appium session
[INFO] simulation ended
[INFO] ending analysis session...
[INFO] analysis report updated at
   out/at.burgerking.BurgerKingApp/20211123_013049_analysis.json
[INFO] analysis complete.
[INFO] tasks succeeded for app 1574333332
[INFO] shutting down server
[INFO] No more tasks to execute.
[INFO] 1 apps succeeded:
[INFO] - 1574333332
[INFO] 0 apps failed:
```

Listing 8.11: Final part of the analysis output: all steps completed, platform shutdown.

3. Output File Inspection

In this final subsection, we want to provide a very brief look into the files generated during the analysis. Specifically, we want to demonstrate how to identify that the Advertising Identifier was accessed and sent. Thus, we show that a privacy label *should have been* declared by the app. For information about the output folder structure and the report files, see Section 5.4.

Analysis Report. The Analysis Report shows that use of the Advertising Identifier was not declared as a label (line 3), but shows that ATT permission has been declared (line 14 and 11) and the user has been asked for permission (line 9). Finally, it shows that the AdId was accessed 16 times (line 26):

```
1  "accessTypes": {
2    "tracking": {
3      "privacyLabel": {
4        "hasPrivacyLabel": false,
5        "search_terms": [... omitted ...],
6        "declared_in_types": []
7      },
8      "accessStatusChecked": true,
9      "accessRequested": true,
10     "requirements": {
11       "allRequirementsMet": true,
12       "usageDescriptions": [{
13           "usageDescriptionKey": "NSUserTrackingUsageDescription",
14           "usageDescriptionPresent": true,
15           "usageDescriptionValue": "This app collects data
16                 to provide personalized offers to you",
17           "required": true
18         }],
19       "libsIncluded": {
20         "ATTrackingManager": true,
21         "ASIdentifierManager": true
22       }
23     },
24     "metrics": {
25       "advertisingIdentifierAccessed": true,
26       "advertisingIdentifierAccessCount": 16
27     }
28   }, // ...
29 }
```

Listing 8.12: Excerpt of the Burger King Österreich app Analysis Report, showing the infos regarding the tracking access type.

Event Log. The Event Log can now be used to identify that the Advertising Identifier has not only been accessed, but also sent. This is done by looking for events with the name `network_request` (created using our Frida interceptor) followed by an event named `network_ad_id` (sent from the NetworkAnalyzer). If both events contain the same target URL, we can be sure that the AdId has indeed been sent by this app.

Indeed, we find two pairs of such events:

```
{
   "url": "https://graph.facebook.com/v8.0/196055407227250/activities",
   "method": "POST",
   "is_tracking_url": false,
   "access_type": "network",
   "event_name": "network_request",
   "timestamp": "2021-11-23T01:30:52.561472+00:00"
}
{
   "event_name": "network_ad_id",
   "found_locations": ["body"],
   "host": "graph.facebook.com",
   "url": "https://graph.facebook.com/v8.0/196055407227250/activities",
   "method": "POST",
   "access_type": "external",
   "timestamp": "2021-11-23T01:30:52.637561+00:00"
}
{
   "url": "https://app-measurement.com/a",
   "method": "POST",
   "is_tracking_url": true,
   "access_type": "network",
   "event_name": "network_request",
   "timestamp": "2021-11-23T01:31:02.889778+00:00"
}
{
   "event_name": "network_ad_id",
   "found_locations": ["body"],
   "host": "app-measurement.com",
   "url": "https://app-measurement.com/a",
   "method": "POST",
   "access_type": "external",
   "timestamp": "2021-11-23T01:31:03.004088+00:00"
}
```

Listing 8.13: Excerpt of the Burger King Österreich app analysis Event Log, showing events indicating that the AdId has been sent.

8.3 Analysis Report Example

The purpose and structure of the file is discussed in Section 5.4.2.

Listing 8.14: Sample privacy analysis report of the Microsoft Outlook app, generated by our platform. In case of arrays, only the first few elements are displayed (as an example), while the remaining similar entries were omitted to to save space.

```
{
  "analysisInfo": {
    "startTime": "2021-11-03T13:47:21.632264+00:00",
    "endTime": "2021-11-03T13:48:10.619499+00:00",
    "duration_seconds": 48
  },
  "appInfo": {
    "CFBundleDisplayName": "Outlook",
    "CFBundleIdentifier": "com.microsoft.Office.Outlook",
    "CFBundleShortVersionString": "4.2143.0",
    "CFBundleVersion": "10680788",
    "usageDescriptions": {
      "NSBluetoothAlwaysUsageDescription": "Allow Outlook to discover
            and connect to Bluetooth devices and accessories.",
      "NSCameraUsageDescription": "This lets you insert photos
            into your emails and more within the app.",
      "NSUserTrackingUsageDescription": "This will be used to show
            you more personalized ads.",
      // ... further usage descriptions omitted
    },
    "privacyLabels": [
      {
        "privacyType": "Data Used to Track You",
        "identifier": "DATA_USED_TO_TRACK_YOU",
        "description": "The following data may be used to track you
          across apps and websites owned by other companies:",
        "dataCategories": [
          {
            "dataCategory": "Usage Data",
            "identifier": "USAGE_DATA",
            "dataTypes": [
              "Advertising Data"
            ]
          }
        ],
        "purposes": []
      },
```

```
    {
      "privacyType": "Data Linked to You",
      "identifier": "DATA_LINKED_TO_YOU",
      "description": "The following data, which may be collected
         and linked to your identity, may be used
         for the following purposes:",
      "dataCategories": [],
      "purposes": [
        {
          "purpose": "Third-Party Advertising",
          "identifier": "THIRD_PARTY_ADVERTISING",
          "dataCategories": [
            {
              "dataCategory": "Usage Data",
              "identifier": "USAGE_DATA",
              "dataTypes": [
                "Advertising Data"
              ]
            }
          ]
        },
        // ... further purposes omitted
      ]
    },
    {
      "privacyType": "Data Not Linked to You",
      "identifier": "DATA_NOT_LINKED_TO_YOU",
      "description": "The following data, which may be collected
         but is not linked to your identity, may be used
         for the following purposes:",
      "dataCategories": [],
      "purposes": [
        {
          "purpose": "App Functionality",
          "identifier": "APP_FUNCTIONALITY",
          "dataCategories": [
            {
              "dataCategory": "Location",
              "identifier": "LOCATION",
              "dataTypes": [
                "Precise Location"
              ]
            }
```

```
          ]
        }
      ]
    }
  ]
},
"accessTypes": {
  "tracking": {
    "privacyLabel": {
      "hasPrivacyLabel": true,
      "search_terms": [
        {
          "category": "IDENTIFIERS",
          "data_types": [
            "Device ID",
            "Ger\u00e4te-ID"
          ]
        }
      ],
      "declared_in_types": [
        {
          "type": "DATA_LINKED_TO_YOU",
          "purposes": [
            "ANALYTICS",
            "APP_FUNCTIONALITY"
          ]
        }
      ]
    },
    "accessStatusChecked": true,
    "accessRequested": false,
    "requirements": {
      "allRequirementsMet": true,
      "usageDescriptions": [
        {
          "usageDescriptionKey": "NSUserTrackingUsageDescription",
          "usageDescriptionPresent": true,
          "usageDescriptionValue": "This will be used to show you more personalized ads.",
          "required": true
        }
      ],
      "libsIncluded": {
```

```
          "ATTrackingManager": true,
          "ASIdentifierManager": true
        }
      },
      "metrics": {
        "advertisingIdentifierAccessed": true,
        "advertisingIdentifierAccessCount": 5
      }
    },
    "network": {
      "trackingDomainsDetected": false,
      "trackingDomains": {},
      "domains": {
        "app.adjust.com": 5,
        "outlookmobile-office365-tas.msedge.net": 2,
        "prod-autodetect.outlookmobile.com": 1,
        "in.appcenter.ms": 1,
        "self.events.data.microsoft.com": 17,
        "outlook-1.cdn.office.net": 2,
        "prod-collecticon.outlookmobile.com": 2,
        "olmprodcollecticonblob.blob.core.windows.net": 164
      }
    },
    "photos": {
      "privacyLabel": {
        "hasPrivacyLabel": true,
        "search_terms": [
          {
            "category": "USER_CONTENT",
            "data_types": [
              "Photos or Videos",
              "Fotos oder Videos"
            ]
          }
        ],
        "declared_in_types": [
          {
            "type": "DATA_LINKED_TO_YOU",
            "purposes": [
              "APP_FUNCTIONALITY"
            ]
          }
        ]
```

```
      },
      "accessStatusChecked": false,
      "accessRequested": false,
      "requirements": {
        "allRequirementsMet": true,
        "usageDescriptions": [
          {
            "usageDescriptionKey": "NSPhotoLibraryUsageDescription",
            "usageDescriptionPresent": true,
            "usageDescriptionValue": "Allow Outlook access to your photos to attach photos to your emails.",
            "required": true
          }
        ],
        "libsIncluded": {
          "PHPhotoLibrary": true,
          "PHAsset": true
        }
      },
      "metrics": {
        "photoLibraryAccessed": false,
        "photoLibraryAccessCount": 0
      }
    },
    "contacts": {
      // ...
    },
    "location": {
      // ...
    },
    "camera": {
      // ...
    },
    "microphone": {
      // ...
    }
  },
  "logVersion": 1.0
}
```

List of Figures

List of Tables

Bibliography

[AH13] Yuvraj Agarwal and Malcolm Hall. ProtectMyPrivacy: Detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *MobiSys 2013 - Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, volume 6, pages 97–109, 2013.

[App18] Apple. Unauthorized modification of iOS can cause security vulnerabilities, instability, shortened battery life, and other issues. *Apple Support, Online, https://support.apple.com/en-gb/HT201954*, (accessed Nov. 18, 2021), 2018.

[BLV+18] Reuben Binns, Ulrik Lyngs, Max Van Kleek, Jun Zhao, Timothy Libert, and Nigel Shadbolt. Third party tracking in the mobile ecosystem. *WebSci 2018 - Proceedings of the 10th ACM Conference on Web Science*, pages 23–31, 2018.

[Bur18] Dave Burke. Previewing Android P. *Android Developers Blog, Online, https://android-developers.googleblog.com/2018/03/previewing-android-p.html*, (accessed Nov. 12, 2021), 2018.

[Cam20] Ian Carlos Campbell. Apple will require apps to add privacy 'nutrition labels' starting December 8th. *The Verge, Online, https://www.theverge.com/2020/11/5/21551926/apple-privacy-developers-nutrition-labels-app-store-ios-14*, (accessed Nov. 17, 2021), 2020.

[CFL+17] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings 2017 Network and Distributed System Security Symposium*, Reston, VA, 2017. Internet Society.

[Con16] Kate Conger. Apple will require HTTPS connections for iOS apps by the end of 2016. *TechCrunch, Online, https://techcrunch.com/2016/06/14/apple-will-require-https-connections-for-ios-apps-by-the-end-of-2016/*, (accessed Nov. 12, 2021), 2016.

[CWC+16] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, Xiaofeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following Devil's

Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 357–376. IEEE, 2016.

[Dor21] Luke Dormehl. Apple's App Privacy Labels May Not Be Totally Accurate. *MakeUseOf, Online, https://www.makeuseof.com/app-privacy-labels-not-totally-accurate/*, (accessed Nov. 30, 2021), 2021.

[DSZX15] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. IRiS: Vetting private API abuse in iOS applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, volume 2015-Octob, pages 44–56, 2015.

[EKKV11] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[FHE+12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS 2012 - Proceedings of the 8th Symposium on Usable Privacy and Security*, 2012.

[Fow21] Geoffrey A Fowler. I checked Apple's new privacy 'nutrition labels.' Many were false. *Washington Post, Online, https://www.washingtonpost.com/technology/2021/01/29/apple-privacy-nutrition-label/*, (accessed Nov. 30, 2021), 2021.

[GC21] Gennie Gebhart and Bennett Cyphers. Apple's AppTrackingTransparency is Upending Mobile Phone Tracking. *Electronic Frontier Foundation, Online, https://www.eff.org/deeplinks/2021/04/apples-apptrackingtransparency-upending-mobile-phone-tracking*, (accessed Nov. 17, 2021), 2021.

[HRE+19] Catherine Han, Irwin Reyes, Amit Elazari, Bar On, Joel Reardon, Álvaro Feal, Kenneth A Bamberger, Serge Egelman, and Narseo Vallina-rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. *Workshop on Technology and Consumer Protection (ConPro '19)*, pages 1–7, 2019.

[HYG+13] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, and Robert Huijie Deng. Comparing Mobile Privacy Protection through Cross-Platform Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*, 2013.

[KBCR09] Patrick Gage Kelley, Joanna Bresee, Lorrie Faith Cranor, and Robert W. Reeder. A "nutrition label" for privacy. In *SOUPS 2009 - Proceedings of the 5th Symposium On Usable Privacy and Security*, volume 1990, 2009.

[KHRW19] Ansgar Kellner, Micha Horlboge, Konrad Rieck, and Christian Wressnegger. False sense of security: A study on the effectivity of jailbreak detection in banking apps. In *Proceedings - 4th IEEE European Symposium on Security and Privacy, EURO S and P 2019*, pages 1–14, 2019.

[KSB+21] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. Are iPhones Really Better for Privacy? Comparative Study of iOS and Android Apps. *arXiv*, 2109.13722, 2021.

[LBAU17] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A. Selcuk Uluagac. A Survey on Function and System Call Hooking Approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, 2017.

[MM17] Kelly D. Martin and Patrick E. Murphy. The role of data privacy in marketing. *Journal of the Academy of Marketing Science*, 45(2):135–155, 2017.

[MP15] Ibtisam Mohamed and Dhiren Patel. Android vs iOS security: A comparative study. In *Proceedings of the 12th International Conference on Information Technology: New Generations, ITNG 2015*, pages 725–730. IEEE, 2015.

[OBE16] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. CRiOS: Toward Large-Scale iOS Application Analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 33–42, New York, NY, USA, oct 2016. ACM.

[RNVR+18] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA, 2018. Internet Society.

[ROF+19] Joel Reardon, Amit Elazari Bar On, Álvaro Feal, Narseo Vallina-Rodriguez, Primal Wijesekera, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *Proceedings of the 28th USENIX Security Symposium*, pages 603–620, 2019.

[RRL+16] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic Jingjing. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374, New York, NY, USA, jun 2016. ACM.

[RT15] Ole André Vadla Ravnås and Karl Trygve Kalleberg. Frida: The engineering behind the reverse engineering. *Conference Talk Slides, Online, https://frida.re/slides/osdc-2015-the-engineering-behind-the-reverse-engineering.pdf*, (accessed Dec. 7, 2021), 2015.

[Sch11] Erick Schonfeld. Apple Sneaks A Big Change Into iOS 5: Phasing Out Developer Access To The UDID. *TechCrunch, Online, https://techcrunch.com/2011/08/19/apple-ios-5-phasing-out-udid/*, (accessed Nov. 11, 2021), 2011.

[SLA+16] Anastasia Shuba, Anh Le, Emmanouil Alimpertis, Minas Gjoka, and Athina Markopoulou. AntMonitor: A System for On-Device Mobile Network Monitoring and its Applications. *arXiv*, 1611.04268, 2016.

[TTX+20] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. iOS, your OS, everybody's OS: Vetting and analyzing network services of iOS applications. In *Proceedings of the 29th USENIX Security Symposium*, pages 2415–2432, 2020.

[War20] Tom Warren. Facebook criticizes Apple's iOS privacy changes with full-page newspaper ads. *The Verge, Online, https://www.theverge.com/2020/12/16/22178068/facebook-apple-newspaper-ads-ios-privacy-changes*, (accessed Nov. 17, 2021), 2020.