Premio de Investigación en Protección de Datos Personales Emilio Aced

Resumen ejecutivo:

50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System

Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodríguez, Serge Egelman

1. Objetivo

El sistema de control de permisos de Android se encarga de proteger recursos restringidos del sistema y datos personales del usuario. Sin embargo, las **aplicaciones pueden explotar vulnerabilidades de seguridad desconocidas para acceder a recursos y datos protegidos sin el conocimiento, permiso y consentimiento del usuario como requiere la RGPD**. En este estudio, utilizamos la infraestructura de análisis automático de apps, AppCensus para detectar la utilización de mecanismos de *covert channels* (canales encubiertos) y *side channels* (canales laterales) por aplicaciones Android publicadas en Google Play. Nuestro estudio reporta evidencias del uso fraudulento y engañoso de varias vulnerabilidades del sistema operativo por parte de aplicaciones y librerías de terceros—principalmente de servicios de publiciad y analíticas de usuario—para acceder a datos privilegiados sin permiso del usuario con fines de monitorización.

2. Motivación

Muchos estudios previos han teorizando sobre el uso de covert y side channels para acceder a información restringida con el fin de rastrear a los usuarios pero sin requerir el permiso pertinente. Esto implica directamente, que apps podrían acceder a datos sensibles sin el consentimiento del usuario como requiere la RGPD. Sin embargo, los ataques propuestos son altamente complejos e impracticos con poca aplicabilidad en el mundo real. Además, ningún estudio ha demostrado el uso de dichos mecanismos en la práctica por apps reales. Debido a esta falta de conocimiento, nuestro estudio buscaba detectar el uso de éstas técnicas por apps reales y que van en detrimento de la privacidad de millones de usuarios. En nuestros esfuerzos analizamos más de 88.000 apps Android presentes en Google Play, y empleamos métodos de análisis estático y dinámico para descubrir comportamientos intrusivos que habían permanecido sin detectar por Google Play protect.

3. Metodología

Para este estudio, hemos desarrollado un framework basado en la capacidad de AppCensus para analizar automáticamente el tráfico de las aplicaciones—incluso cuando está encriptado y codificado—y su comportamiento en ejecución gracias a la mezcla de una versión customizada del sistema operativo Android. En nuestro análisis, analizamos el conjunto de permisos requeridos por cada app y restringimos el acceso a los mismos. Posteriormente, auditamos cada app a través de nuestro sistemas, monitorizando el tráfico de red para observar si datos protegidos y asociados a esos permisos (a los que la app no tiene acceso) han sido transmitidos a la nube. Si esto ocurre, las apps están explotando vulnerabilidades del sistema para acceder a dichos datos. Luego, estudiamos el código fuente de la aplicación para identificar el side channel o covert channel utilizado.

4. Resultados

Probamos más de 88.000 aplicaciones y encontramos los siguientes casos de covert y side channels:

- Encontramos 5 aplicaciones accediendo a la MAC del punto de acceso WiFi para inferir la localización del usuario sin usar el GPS, además de otras 5 con el código pertinente

para hacerlo sin requerir ningún tipo de permiso gracias al uso de las tablas ARP (que antes de este artículo estaban desprotegidas).

- Encontramos que la librería de la popular plataforma Unity usaba llamadas IOCTL para descubrir la dirección MAC de la tarjeta de red del dispositivo, dato que puede ser usado como un identificador único del dispositivo (como el IMEI). Encontramos evidencia de 42 apps utilizando este método además de 12.408 con el código que habilita dicho canal lateral. Este ataque permite monitorizar de forma única a los dispositivos sin ningún permiso.

- Dos librerías populares de terceros, Salmon Ads y Baidu, utilizan un covert channel por el que una app con permiso para acceder al IMEI, un identificador único del dispositivo, guarda su valor en un archivo del sistema de ficheros para que cualquier otra app que incluya la misma librería pueda leer este identificador directamente del archivo donde se ha escrito su valor, sin necesidad de que el usuario conceda el permiso relevante a la aplicación. Encontramos 159 apps con el potencial de realizar este ataque y evidencia de 13 apps haciéndolo.

- Por último, encontramos una app que utilizaba los metadatos EXIF presentes en las imágenes tomadas por la cámara y almacenadas en la tarjeta de memoria para acceder a la localización del usuario sin permiso, usando para ello el permiso de STORAGE.

5. Novedad y aplicabilidad

Este artículo presenta el primer método para detectar y estudiar el uso de covert y side channels en el ecosistema Android por apps y librerías usadas por millones de usuarios. Este artículo fue presentado en el prestigioso congreso internacional (revisado por pares) USENIX Security Symposium y ha recibido el reconocimiento de la comunidad científica al otorgarsele el **premio al artículo distinguido**. [1] Multiples medios internacionales de gran difusión como “El País”, “Le Monde”, “Le Figaro” o “Business Insider” se hicieron eco de esta investigación. **El impacto industrial y social del trabajo es masivo, ya que ha permitido detectar y solventar vulnerabilidades desconocidas en el sistema operativo Android, y por tanto, mejorar la privacidad de miles de millones de usuarios en todo el mundo.** Tras revelar estos descubrimientos a Google, estos han implementado varios cambios y mejoras de privacidad incluidas en Android 10. [2] Por nuestros esfuerzos, los autores recibieron una recompensa por parte de Google y varios CVEs asociados con nuestros descubrimientos.

6. Conclusiones

Este estudio descubre por vez primera el uso de covert y side channels en más de 88.000 aplicaciones de Android para acceder a datos sensibles de usuario sin su conocimiento y consentimiento. El artículo revela 4 ataques presentes en más de 12.000 apps con la capacidad de esquivar el sistema de permisos de Android para acceder a datos sensibles como geolocalización e identificadores únicos sin el consentimiento del usuario. Reportamos nuestros resultados a Google lo que impulsó diferentes cambios y mejoras en Android 10.

[1] https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[2] https://developer.android.com/about/versions/10/privacy/

1. Objective

Android's permission model is in charge of protecting access to restricted system resources and personal data from users. Nevertheless, **apps can exploit unknown vulnerabilities to gain access to certain resources and protected data without users knowledge, permission and consent, as GDPR requires**. In this study, we user our automatic data analysis infrastructure, AppCensus, to study the presence and usage of *covert* and *side channels* by Android applications publicly available on Google Play. Our study reports evidence of a fraudulent usage of several vulnerabilities of the operative system by apps and third-party libraries—mainly those used for user tracking and advertisement purposes—to access personal data from users without the appropriate permission in an attempt to monitor them.

2. Motivation

Previous work has looked at theoretical ways to use of covert and side channels to access private information in an attemtp to track users without the pertinent permissions. This implies that apps could access sensitive data without user's consent, contrary to what GDPR mandates. Nevertheless, the proposed attacks are highly complex and impractical to be exploited in real life settings. Furthermore, none of these studies have shown these mechanisms being used on the wild by real apps. Due to this lack of knowledge, we have focused on the analysis of the use of these techniques by real apps and potentially affecting the privacy of millions of users. We analyzed over 88k Android apps from google Play, using static and dynamic analysis techniques to discover intrusive behaviors that had not been previously detected by Google Play protect.

3. Methodology

For this study, we have developed a framework baed on AppCensus's ability to automatically analyze—without human interaction—apps' traffic—even when encypted—and their runtime behavior thanks to the use of a custom version of Android's operative system. In our analysis, we analyze the set of permissions requested by each app and reject the access to different resources to apps. Then, we audit each app using our system, monitoring the network traffic and observing whether the app is transmitting any protected resource for which we have not granted the appropriate permission to the cloud. If this happens, these apps are exploiting vulnerabilities on the system to access said data. Then, we analyze the source code of the app to identify what is the covert or side channel used.

4. Results

We test over 88k app and find the following covert and side channels:

- We find 5 apps accessing the WiFi's Access Point MAC address to infer user location without using the GPS, and other 5 apps with the pertinent code to do so without requiring any permission by virtue of the ARP tables (which were unprotected prior to this study).
- We find that the popular library Unity was using IOCTL calls to discover the MAC address of the device's network card. This piece of data can be used as a unique identifier (like the IMEI). We find evidence of 42 apps using this method, as well as 12.408 apps with the

code that enables such side-channel. This attack allows to uniquely identify users without the need for any permission.

- Two third-party libraries, Salmon Ads and Baidu, were using a covert channel in which an app with permission to access the IMEI, a unique identifier for the device, saved its value in a file on the folder system so that any other app that includes the same library could read this value directly from the file where it was written without the need for the relevant permission to obtain this unique identifier. We find 159 apps with the code to exploit this kind of attack, as well as evidence form 13 apps doing so.
- Finally, we find that one app was using EXIF metadata present in the pictures taken with the camera and saved on the memory card to access location data from the user using only the STORAGE permission.

5. Novelty and applicability

This article presents the first method to detect and study the use of covert and side channels in Android's ecosystem of apps and third-party libraries, used by millions of users. This article was presented in the prestigious USENIX Security conference and receive the **Distinguished paper award**. [3] Multiple highly read international media outlets such as "El País", "Le Monde", "Le Figaro" or "Business Insider" highlighted the results of our investigation. **The industrial and societal impact of this work is massive, as it allowed to detect and solve previously unknown vulnerabilities in Android's operative system, and thus, improve the privacy of millions of users worldwide.** After reporting our findings to Google, they have implemented several privacy changes and improvements included in Android 10. [4] As part of our efforts, the authors received a bug bounty from Google and several CVEs associated with our findings.

6. Conclusions

In this study, we analyze the use of covert and side channels in over 88.000 Android apps. We discover 4 different attacks in over 12.000 apps that had the pertinent code to bypass ANdroid's permission model and access restricted information and resources such as location info or unique identifiers without user consent. We report our result to Google, obtaining a bug bounty for our findings and impulsing several changes in the following version of Android's operative system.

[3] https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[4] https://developer.android.com/about/versions/10/privacy/

50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System

Joel Reardon
University of Calgary
AppCensus, Inc.

Álvaro Feal
IMDEA Networks Institute
Universidad Carlos III de Madrid

Primal Wijesekera
U.C. Berkeley / ICSI

Amit Elazari Bar On
U.C. Berkeley

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

Serge Egelman
U.C. Berkeley / ICSI
AppCensus, Inc.

Abstract

Modern smartphone platforms implement permission-based models to protect access to sensitive data and system resources. However, apps can circumvent the permission model and gain access to protected data without user consent by using both covert and side channels. Side channels present in the implementation of the permission system allow apps to access protected data and system resources without permission; whereas covert channels enable communication between two colluding apps so that one app can share its permission-protected data with another app lacking those permissions. Both pose threats to user privacy.

In this work, we make use of our infrastructure that runs hundreds of thousands of apps in an instrumented environment. This testing environment includes mechanisms to monitor apps' runtime behaviour and network traffic. We look for evidence of side and covert channels being used in practice by searching for sensitive data being sent over the network for which the sending app did not have permissions to access it. We then reverse engineer the apps and third-party libraries responsible for this behaviour to determine how the unauthorized access occurred. We also use software fingerprinting methods to measure the static prevalence of the technique that we discover among other apps in our corpus.

Using this testing environment and method, we uncovered a number of side and covert channels in active use by hundreds of popular apps and third-party SDKs to obtain unauthorized access to both unique identifiers as well as geolocation data. We have responsibly disclosed our findings to Google and have received a bug bounty for our work.

1 Introduction

Smartphones are used as general-purpose computers and therefore have access to a great deal of sensitive system resources (*e.g.*, sensors such as the camera, microphone, or GPS), private data from the end user (*e.g.*, user email or contacts list), and various persistent identifiers (*e.g.*, IMEI). It is crucial to protect this information from unauthorized access. Android, the most-popular mobile phone operating system [74], implements a permission-based system to regulate access to these sensitive resources by third-party applications. In this model, app developers must explicitly request *permission* to access sensitive resources in their Android Manifest file [5]. This model is supposed to give users control in deciding which apps can access which resources and information; in practice it does not address the issue completely [30, 85].

The Android operating system sandboxes user-space apps to prevent them from interacting arbitrarily with other running apps. Android implements isolation by assigning each app a separate *user ID* and further mandatory access controls are implemented using SELinux. Each running process of an app can be either code from the app itself or from SDK libraries embedded within the app; these SDKs can come from Android (*e.g.*, official Android support libraries) or from third-party providers. App developers integrate third-party libraries in their software for things like crash reporting, development support, analytics services, social-network integration, and advertising [62, 16]. By design, any third-party service bundled in an Android app inherits access to all permission-protected resources that the user grants to the app. In other words, if an app can access the user's location, then all third-party services embedded in that app can as well.

In practice, security mechanisms can often be circumvented; *side channels* and *covert channels* are two common techniques to circumvent a security mechanism. These channels occur when there is an alternate means to access the protected resource that is not audited by the security mechanism, thus leaving the resource unprotected. A *side channel* exposes a path to a resource that is outside the security mechanism; this can be because of a flaw in the design of the security mechanism or a flaw in the implementation of the design. A classic example of a side channel is that power usage of hardware when performing cryptographic operations can leak the particulars of a secret key [42]. As an example in the physical world, the frequency of pizza deliveries to government buildings may leak information about political crises [68].

A *covert channel* is a more deliberate and intentional effort between two cooperating entities so that one with access to some data provides it to the other entity without access to the data in violation of the security mechanism [43]. As an example, someone could execute an algorithm that alternates between high and low CPU load to pass a binary message to another party observing the CPU load.

The research community has previously explored the potential for covert channels in Android using local sockets and shared storage [49], as well as other unorthodox means, such as vibrations and accelerometer data to send and receive data between two coordinated apps [3]. Examples of side channels include using device sensors to infer the gender of the user [51] or uniquely identify the user [71]. More recently, researchers demonstrated a new permission-less device fingerprinting technique

that allows tracking Android and iOS devices across the Internet by using factory-set sensor calibration details [89]. However, there has been little research in detecting and measuring at scale the prevalence of covert and side channels in apps that are available in the Google Play Store. Only isolated instances of malicious apps or libraries inferring users' locations from WiFi access points were reported, a side channel that was abused in practice and resulted in about a million dollar fine by regulators [81].

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

To this end, we extend the state of the art by developing methods to detect actual circumvention of the Android permission system, at scale in real apps by using a combination of dynamic and static analysis. We automatically executed over 88,000 Android apps in a heavily instrumented environment with capabilities to monitor apps' behaviours at the system and network level, including a TLS man-in-the-middle proxy. In short, we ran apps to see when permission-protected data was transmitted by the device, and scanned the apps to see which ones *should not* have been able to access the transmitted data due to a lack of granted permissions. We grouped our findings by *where* on the Internet *what* data type was sent, as this allows us to attribute the observations to the actual app developer or embedded third-party libraries. We then reverse engineered the responsible component to determine exactly how the data was accessed. Finally, we statically analyzed our entire dataset to measure the prevalence of the channel. We focus on a subset of the *dangerous* permissions that prevent apps from accessing location data and identifiers. Instead of imagining new channels, our work focuses on tracing evidence that suggests that side- and covert-channel abuse is occurring in practice.

We studied more than 88,000 apps across each category from the U.S. Google Play Store. We found a number of side and covert channels in active use, responsibly disclosed our findings to Google and the U.S. Federal Trade Commission (FTC), and received a bug bounty for our efforts.

In summary, the contributions of this work include:

- We designed a pipeline for automatically discovering vulnerabilities in the Android permissions system through a combination of dynamic and static analysis, in effect creating a scalable honeypot environment.
- We tested our pipeline on more than 88,000 apps and discovered a number of vulnerabilities, which we responsibly disclosed. These apps were downloaded from the U.S. Google Play Store and include popular apps from all categories. We further describe the vulnerabilities in detail, and measure the degree to which they are in active use, and thus pose a threat to users. We discovered covert and side channels used in the wild that compromise both users' location data and persistent identifiers.
- We discovered companies getting the MAC addresses of the connected WiFi base sta-

tions from the ARP cache. This can be used as a surrogate for location data. We found 5 apps exploiting this vulnerability and 5 with the pertinent code to do so.
- We discovered Unity obtaining the device MAC address using `ioctl` system calls. The MAC address can be used to uniquely identify the device. We found 42 apps exploiting this vulnerability and 12,408 apps with the pertinent code to do so.
- We also discovered that third-party libraries provided by two Chinese companies—Baidu and Salmonads—independently make use of the SD card as a covert channel, so that when an app can read the phone's IMEI, it stores it for other apps that cannot. We found 159 apps with the potential to exploit this covert channel and empirically found 13 apps doing so.
- We found one app that used picture metadata as a side channel to access precise location information despite not holding location permissions.

These deceptive practices allow developers to access users' private data without consent, undermining user privacy and giving rise to both legal and ethical concerns. Data protection legislation around the world—including the General Data Protection Regulation (GDPR) in Europe, the California Consumer Privacy Act (CCPA) and consumer protection laws, such as the Federal Trade Commission Act—enforce transparency on the data collection, processing, and sharing practices of mobile applications.

This paper is organized as follows: Section 2 gives more background information on the concepts discussed in the introduction. Section 3 describes our system to discover vulnerabilities in detail. Section 4 provides the results from our study, including the side and covert channels we discovered and their prevalence in practice. Section 5 describes related work. Section 6 discusses their potential legal implications. Section 7 discusses limitations to our approach and concludes with future work.

2 Background

The Android permissions system has evolved over the years from an ask-on-install approach to an ask-on-first-use approach. While this change impacts when permissions are granted and how users can use contextual information to reason about the appropriateness of a permission request, the backend enforcement mechanisms have remained largely unchanged. We look at how the design and implementation of the permission model has been exploited by apps to bypass these protections.

2.1 Android Permissions

Android's permissions system is based on the security principle of *least privilege*. That is, an entity should only have the minimum capabilities it needs to perform its task. This standard design principle for security implies that if an app acts maliciously, the damage will be limited. Developers must declare the permissions that their apps need beforehand, and the user is given an opportunity to review them and decide whether to install the app. The Android platform, however, does not judge whether the set of requested permissions are all strictly necessary for the app to function. Developers are free to request more permissions than they actually need and users are expected to judge if they are reasonable.

The Android permission model has two important aspects: obtaining user consent before an app is able to access any of its requested permission-protected resources, and then ensuring that the app cannot access resources for which the user has not granted consent. There is a long line of work uncovering issues on how the permission model interacts with the user: users are inadequately informed about why apps need permissions at installation time, users misunderstand exactly what the purpose of different permissions are, and users lack context and transparency into how apps will ultimately use their granted permissions [30, 77, 85, 24]. While all of these are critical issues that need attention, the focus of our work is to understand how apps are circumventing system checks to verify that apps have been granted various permissions.

When an app requests a permission-protected resource, the resource manager (*e.g.*, `LocationManager`, `WiFiManager`, etc.) contacts the `ActivityServiceManager`, which is the *reference monitor* in Android. The resource request originates from the sandboxed app, and the final verification happens inside the Android platform code. The platform is a Java operating system that runs in system space and acts as an interface for a customized Linux kernel, though apps can interact with the kernel directly as well. For some permission-protected resources, such as network sockets, the reference monitor is the kernel, and the request for such resources bypasses the platform framework and directly contacts the kernel. Our work discusses how real-world apps circumvent these system checks placed in the kernel and the platform layers.

The Android permissions system serves an important purpose: to protect users' privacy and sensitive system resources from deceptive, malicious, and abusive actors. At the very least, if a user denies an app a permission, then that app should not be able to access data protected by that permission [80, 24]. In practice, this is not always the case.

2.2 Circumvention

Apps can circumvent the Android permission model in different ways [49, 69, 3, 17, 54, 73, 71, 51, 52]. The use of covert and side channels, however, is particularly troublesome as their usage indicates deceptive practices that might mislead even diligent users, while underscoring a security vulnerability in the operating system. In fact, the United State's Federal Trade Commission (FTC) has fined mobile developers and third-party libraries for exploiting side channels: using the MAC address of the WiFi access point to infer the user's location [81]. Figure 1 illustrates the difference between covert and side channels and shows how an app that is denied permission by a security mechanism is able to still access that information.

Covert Channel A covert channel is a communication path between two parties (*e.g.*, two mobile apps) that allows them to transfer information that the relevant security enforcement mechanism deems the recipient unauthorized to receive [18]. For example, imagine that AliceApp has been granted permission through the Android API to access the phone's IMEI (a persistent identifier), but BobApp has been denied access to that same data. A covert channel is created when AliceApp legitimately reads the IMEI and then gives it to BobApp, even though BobApp has already been denied access to

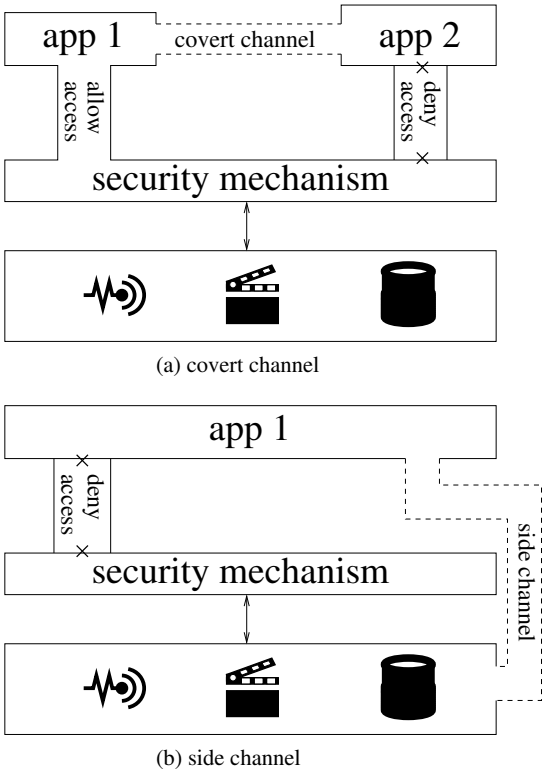


Figure 1: Covert and side channels. (a) A security mechanism allows `app1` access to resources but denies `app2` access; this is circumvented by `app2` using `app1` as a facade to obtain access over a communication channel not monitored by the security mechanism. (b) A security mechanism denies `app1` access to resources; this is circumvented by accessing the resources through a side channel that bypasses the security mechanism.

this same data when requesting it through the proper permission-protected Android APIs.

In the case of Android, different covert channels have been proposed to enable communication between apps. This includes exotic mediums such as ultrasonic audio beacons and vibrations [26, 17]. Apps can also communicate using an external network server to exchange information when no other opportunity exists. Our work, however, exposes that rudimentary covert channels, such as shared storage, are being used in practice at scale.

Side Channel A side channel is a communication path that allows a party to obtain privileged information without relevant permission checks occurring. This can be due to non-conventional unprivileged functions or features, as well as ersatz versions of the same information being available without being protected by the same permission. A classical example of a side channel attack is the timing attack to exfiltrate an encryption key from secure storage [42]. The system under attack is an algorithm that performs computation with the key and unintentionally leaks timing information—*i.e.*, how long it runs—that reveals critical information about the key.

Side channels are typically an unintentional consequence of a complicated system. ("Backdoors" are intentionally-created side channels that are meant to be obscure.) In Android, a large and complicated API results in the same data appearing in different locations, each governed by different access control mechanisms. When one API is protected with permissions, another unprotected method may be used to obtain the same data or an ersatz version of it.

2.3 App Analysis Methods

Researchers use two primary techniques to analyze app behaviour: static and dynamic analysis. In short, static analysis studies *software as data* by reading it; dynamic analysis studies *software as code* by running it. Both ap-

proaches have the goal of understanding the software's ultimate behaviour, but they offer insights with different certainty and granularity: static analysis reports instances of hypothetical behaviour; dynamic analysis gives reports of observed behaviour.

Static Analysis Static analysis involves scanning the code for all possible combinations of execution flows to understand potential execution behaviours—the behaviours of interest may include various privacy violations (*e.g.*, access to sensitive user data). Several studies have used static analysis to analyze different types of software in search of malicious behaviours and privacy leaks [21, 10, 39, 20, 22, 37, 45, 9, 11, 19, 32, 41, 4, 91]. However, static analysis does not produce actual observations of privacy violations; it can only suggest that a violation may happen if a given part of the code gets executed at runtime. This means that static analysis provides an upper bound on hypothetical behaviours (*i.e.*, yielding false positives).

The biggest advantage of static analysis is that it is easy to perform automatically and at scale. Developers, however, have options to evade detection by static analysis because a program's runtime behaviour can differ enormously from its superficial appearance. For example, they can use code obfuscation [23, 29, 48] or alter the flow of the program to hide the way that the software operates in reality [23, 29, 48]. Native code in unmanaged languages allow pointer arithmetic that can skip over parts of functions that guarantee pre-conditions. Java's reflection feature allows the execution of dynamically created instructions and dynamically loaded code that similarly evades static analysis. Recent studies have shown that around 30% of apps render code dynamically [46], so static analysis may be insufficient in those cases.

From an app analysis perspective, static analysis lacks the contextual aspect, *i.e.*, it fails to observe the circumstances surrounding each observation of sensitive resource access and sharing, which is important in understanding when a given privacy violation is likely to happen. For these reasons, static analysis is useful, but is well complemented by dynamic analysis to augment or confirm findings.

Dynamic analysis Dynamic analysis studies an executable by running it and auditing its runtime behaviour. Typically, dynamic analysis benefits from running the executable in a controlled environment, such as an instrumented mobile OS [27, 84], to gain observations of an app's behaviour [16, 32, 46, 47, 50, 65, 72, **?**, 84, 86, 87, 88].

There are several methods that can be used in dynamic analysis, one example is taint analysis [27, 32] which can be inefficient and prone to control flow attacks [67, 70]. A challenge to performing dynamic analysis is the logistical burden of performing it at scale. Analyzing a single Android app in isolation is straightforward, but scaling it to run automatically for tens of thousands of apps is not. Scaling dynamic analysis is facilitated with automated execution and creation of behavioural reports. This means that effective dynamic analysis requires building an instrumentation framework for possible behaviours of interest *a priori* and then engineering a system to manage the endeavor.

Nevertheless, some apps are resistant to being audited when run in virtual or privileged environments [12, 67]. This has led to new

auditing techniques that involve app execution on real phones, such as by forwarding traffic through a VPN in order to inspect network communications [63, 44, 60]. The limitations of this approach are the use of techniques robust to man-in-the-middle attacks [28, 31, 61] and scalability due to the need to actually run apps with user input.

A tool to automatically execute apps on the Android platform is the UI/Application Exerciser Monkey [6]. The Monkey is a UI fuzzer that generates synthetic user input, ensuring that some interaction occurs with the app being automatically tested. The Monkey has no context for its actions with the UI, however, so some important code paths may not be executed due to the random nature of its interactions with the app. As a result, this gives a lower bound for possible app behaviours, but unlike static analysis, it does not yield false positives.

Hybrid Analysis Static and dynamic analysis methods complement each other. In fact, some types of analysis benefit from a hybrid approach, in which combining both methods can increase the coverage, scalability, or visibility of the analyses. This is the case for malicious or deceptive apps that actively try to defeat one individual method (*e.g.*, by using obfuscation or techniques to detect virtualized environments or TLS interception). One approach would be to first carry out dynamic analysis to triage potential suspicious cases, based on collected observations, to be later examined thoroughly using static analysis. Another approach is to first carry out static analysis to identify interesting code branches that can then be instrumented for dynamic analysis to confirm the findings.

3 Testing Environment and Analysis Pipeline

Our instrumentation and processing pipeline, depicted and described in Figure 2, combines the advantages of both static and dynamic analysis techniques to triage suspicious apps and analyze their behaviours in depth. We used this testing environment to find evidence of covert- and side-channel usage in 252,864 versions of 88,113 different Android apps, all of them downloaded from the U.S. Google Play Store using a purpose-built Google Play scraper. We executed each app version individually on a physical mobile phone equipped with a customized operating system and network monitor. This testbed allows us to observe apps' runtime behaviours both at the OS and network levels. We can observe how apps request and access sensitive resources and their data sharing practices. We also have a comprehensive data analysis tool to de-obfuscate collected network data to uncover potential deceptive practices.

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

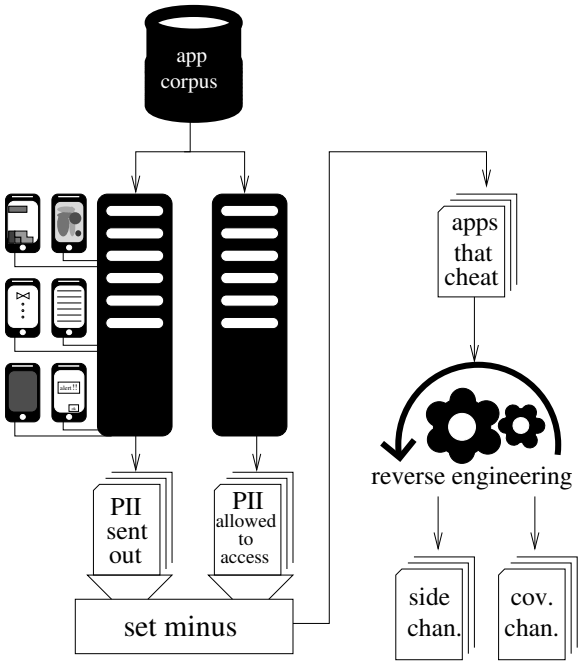


Figure 2: Overview of our analysis pipeline. Apps are automatically run and the transmissions of sensitive data are compared to what would be allowed. Those suspected of using a side or covert channel are manually reverse engineered.

that the same data-destination pair reflects the same underlying side or covert channel. We take one example per group and manually reverse engineer it to determine how the app gained permission-protected information without the corresponding permission.

Finally, we fingerprint the apps and libraries found using covert- and side-channels to identify the static presence of the same code in other apps in our corpus. A fingerprint is any string constant, such as specific filename or error message, that can be used to statically analyze our corpus to determine if the same technique exists in other apps that did not get triggered during our dynamic analysis phase.

3.1 App Collection

We wrote a Google Play Store scraper to download the most-popular apps under each category. Because the popularity distribution of apps is long tailed, our analysis of the 88,113 most-popular apps is likely to cover most of the apps that people currently use. This includes 1,505 non-free apps we purchased for another study [38]. We instrumented the scraper to inspect the Google Play Store to obtain application executables (APK files) and their associated metadata (*e.g.*, number of installs, category, developer information, etc.).

As developers tend to update their Android software to add new functionality or to patch bugs [64], these updates can also be used to introduce new side and covert channels. Therefore, it is important to examine different versions of the same app, because they may exhibit different behaviours. In order to do so, our scraper periodically checks if a new version of an already downloaded app is available and downloads it. This process allowed us to create a dataset consisting of 252,864 different versions of 88,113 Android apps.

3.2 Dynamic Analysis Environment

We implemented the dynamic testing environment described in Figure 2, which consists of about a dozen Nexus 5X Android phones running an instrumented version of the Android Marshmallow platform.[1] This purpose-built en-

[1]While as of this writing Android Pie is the current release [35], Marshmallow and older versions were used by a majority of users at the time that we began data collection.

vironment allows us to comprehensively monitor the behaviour of each of 88,113 Android apps at the kernel, Android-framework, and network traffic levels. We execute each app automatically using the Android Automator Monkey [6] to achieve scale by eliminating any human intervention. We store the resulting OS-execution logs and network traffic in a database for offline analysis, which we discuss in Section 3.3. The dynamic analysis is done by extending a platform that we have used in previous work [**?**].

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

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

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

Network-Level Monitoring We monitored all network traffic, including TLS-secured flows, using a network monitoring tool developed for our previous research activities [63]. This network monitoring module leverages Android's VPN API to redirect all the device's network traffic through a localhost service that inspects all network traffic, regardless of the protocol used, through deep-packet inspection and in user-space. It reconstructs the network streams and ascribes them to the originating app by mapping the app owning the socket to the UID as reported by the `proc` filesystem. Furthermore, it also performs TLS interception by installing a root certificate in the system trusted certificate store. This technique allows it to decrypt TLS traffic unless the app performs advanced techniques, such as certificate pinning, which can be identified by monitoring TLS records and proxy exceptions [61].

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

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

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

3.3 Personal Information in Network Flows

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

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

We focus our study on detecting apps using covert and side channels to access specific types of highly sensitive data, including persistent identifiers and geolocation information. Notably, the unauthorized collection of geolocation information in Android has been the subject of prior regulatory action [81]. Table 1 shows the different types of personal information that we look for in network transmissions,

Table 1: The types of personal information that we search for, the permissions protecting access to them, and the purpose for which they are generally collected. We also report the subsection in this paper where we report side and covert channels for accessing each type of data, if found, and the number of apps exploiting each. The dynamic column depicts the number of apps that we directly observed inappropriately accessing personal information, whereas the static column depicts the number of apps containing code that exploits the vulnerability (though we did not observe being executed during test runs).

Data Type	Permission	Purpose/Use	Subsection	Nº of Apps		Nº of SDKs		Chann
				Dynamic	Static	Dynamic	Static	Covert
IMEI	READ_PHONE_STATE	Persistent ID	4.1	13	159	2	2	2
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.2	42	12,408	1	1	0
Email	GET_ACCOUNTS	Persistent ID	Not Found					
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found					
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found					
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3	5	355	2	10	0
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found					
GPS	ACCESS_FINE_LOCATION	Location Data	4.4	1	1	0	0	0

what each can be used for, the Android permission that protects it, and the subsection in this paper where we discuss findings that concern side and covert channels for accessing that type of data.

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

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

A few libraries followed best practices by generating random AES session keys to encrypt the data and then encrypt the session key with a hard-coded RSA public key, sending both the encrypted data and encrypted session key together. To de-cipher their network transmissions, we instrumented the relevant Java libraries. We found two examples of third-party SDKs "encrypting" their data by XOR-ing a keyword over the data in a Viginère-style cipher. In one case, this was *in addition* to both using

standard encryption for the data *and* using TLS in transmission. Other interesting approaches included reversing the string after encoding it in base64 (which we refer to as "46esab"), using base64 multiple times (basebase6464), and using a permuted-alphabet version of base64 (sa4b6e). Each new discovery is added to our suite of decodings and our entire dataset is then re-analyzed.

3.4 Finding Side and Covert Channels

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

Reverse Engineering After finding a set of apps exhibiting behaviour consistent with the existence of side and covert channels, we manually reverse engineered them. While the reverse engineering process is time consuming and not easily automated, it is necessary to determine how the app actually obtained information outside of the permission system. Because many of the transmissions are caused by the same SDK code, we only needed to reverse engineer each unique *circumvention technique*: not every app, but instead for a much smaller number of unique SDKs. The destination endpoint for the network traffic typically identifies the SDK responsible.

During the reverse engineering process, our first step was to use apktool[7] to decompile and extract the smali bytecode for each suspicious app. This allowed us to analyse and identify where any strings containing PII were created and from which data sources. For some particular apps and libraries, our work also necessitated reverse engineering C++ code; we used IdaPro[1] for that purpose.

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

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

We then decompiled all of the apps in our corpus and searched for the string in the resulting files. Within smali bytecode, we searched for the string in its entirety as a `const-string` instruction. For shared objects libraries like Unity, we use the `strings` command to output its printable strings. We include the path and name of the file as matching criteria to protect against

false positives. The result is a set of all apps that may also exploit the side or covert channel in practice but for which our instrumentation did not flag for manual investigation, *e.g.*, because the app had been granted the required permission, the Monkey did not explore that particular code branch, etc.

4 Results

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

Our testing environment allowed us to identify five different types of side and covert channels in use among the 88,113 different Android apps in our dataset. Table 1 summarizes our findings and reports the number of apps and third-party SDKs that we find exploiting these vulnerabilities in our dynamic analysis and those in which our static analysis reveals code that *can* exploit these channels. Note that this latter category—those that *can* exploit these channels—were not seen as doing so by our instrumentation; this may be due to the Automator Monkey not triggering the code to exploit it or because the app had the required permission and therefore the transmission was not deemed suspicious.

4.1 IMEI

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

The IMEI is also useful to online services as a persistent device identifier for tracking individual phones. The IMEI is a powerful identifier as it takes extraordinary efforts to change its value or even spoof it. In some jurisdictions, it is illegal to change the IMEI [56]. Collection of the IMEI by third parties facilitates tracking in cases where the owner tries to protect their privacy by resetting other identifiers, such as the advertising ID.

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

Salmonads and External Storage Salmonads is a "third party developers' assistant platform in Greater China" that offers analytics and monetization services to app developers [66]. We identified network flows to `salmonads.com` coming from five mobile apps that contained the device's IMEI, despite the fact that the apps did not have permission to access it.

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

app with the Salmonads SDK embedded and with legitimate access to the IMEI, the SDK—through the host app—reads and stores the IMEI in this file.

The covert channel is the apps' shared access to the SD card. Once the file is written, all other apps with the same SDK can simply read the file instead of obtaining access through the Android API, which is regulated by the permission system. Beyond the IMEI, Salmonads also stores the advertising ID—a resettable ID for advertising and analytics purposes that allows opting out of interest-based advertising and personalization—and the phone's MAC address, which is protected with the ACCESS_NETWORK_STATE permission. We modified the file to store new random values and observed that the SDK faithfully sent them onwards to Salmonads' domains. The collection of the advertising ID alongside other non-resettable persistent identifiers and data, such as the IMEI, undermines the privacy-preserving feature of the advertising ID, which is that it can be *reset*. It also may be a violation of Google's Terms of Service [36],

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

Baidu and External Storage Baidu is a large Chinese corporation whose services include, among many others, an online search engine, advertising, mapping services [14], and geocoding APIs [13]. We observed network flows containing the device IMEI from Disney's Hong Kong Disneyland park app (com.disney.hongkongdisneyland_goo) to Baidu domains. This app helps tourists to navigate through the Disney-themed park, and the app makes use of Baidu's Maps SDK. While Baidu Maps initially only offered maps of mainland China, Hong Kong, Macau and Taiwan, as of 2019, it now provides global services.

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

Baidu uses AES in CBC mode with a static key and the same static value for the initialization vector (IV). These values are, in hexadecimal, `33303231323130326469637564696162`. The reason why this value is not superficially representative of a random hexadecimal string is because Baidu's key is computed from

the binary representation of the ASCII string `30212102dicudiab`—observe that when reversed, it reads as `baidu cid 2012 12 03`. As with Salmonads, we confirmed that we can change the (encrypted) contents of this file and the resulting identifiers were faithfully sent onwards to Baidu's servers.

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

4.2 Network MAC Addresses

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

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

Unity and IOCTLs Unity is a cross-platform game engine developed by Unity Technologies and heavily used by Android mobile games [76]. Our traffic analysis identified several Unity-based games sending the MD5 hash of the MAC address to Unity's servers and referring to it as a `uuid` in the transmission (*e.g.*, as an HTTP GET parameter key name). In this case, the access was happening inside of Unity's C++ native library. We reverse engineered `libunity.so` to determine how it was obtaining the MAC address.

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

Unity opens a network socket and uses an `ioctl` (UNIX "input-output control") to obtain the MAC address of the WiFi network interface. In effect, `ioctl`s create a large suite of "numbered" API calls that are technically no different than well-named system calls like `bind` or `close` but used for infrequently used features. The behaviour of an `ioctl` depends on the spe-

cific "request" number. Specifically, Unity uses the SIOCGIFCONF[2] `ioctl` to get the network interfaces, and then uses the SIOCGIFHWADDR[3] `ioctl` to get the corresponding MAC address.

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

4.3 Router MAC Address

Access to the WiFi router MAC address (BSSID) is protected by the ACCESS_WIFI_STATE permission. In Section 2, we exemplified side channels with router MAC addresses being ersatz location data, and discussed the FTC enacting millions of dollars in fines for those engaged in the practice of using this data to deceptively infer users' locations. Android Nougat added a requirement that apps hold an additional location permission to scan for nearby WiFi networks [34]; Android Oreo further required a location permission to get the SSID and MAC address of the connected WiFi network. Additionally, knowing the MAC address of a router allows one to link different devices that share Internet access, which may reveal personal relations by their respective owners, or enable cross-device tracking.

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

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

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

We discovered this technique during dynamic

[2]Socket ioctl get interface configuration

[3]Socket ioctl get interface hardware address

Table 2: SDKs seen sending router MAC addresses and also containing code to access the ARP cache. For reference, we report the number of apps and a lower bound of the total number of installations of those apps. We do this for all apps containing the SDK; those apps that *do not* have ACCESS_WIFI_STATE, which means that the side channel circumvents the permissions system; and those apps which *do* have a location permission, which means that the side channel circumvents location revocation.

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

analysis, when we observed one library using this method in practice: OpenX [57], a company that according to their website "creates programmatic marketplaces where premium publishers and app developers can best monetize their content by connecting with leading advertisers that value their audiences." OpenX's SDK code was not obfuscated and so we observed that they had named the responsible function `getDeviceMacAddressFromArp`. Furthermore, a close analysis of the code indicated that it would first *try* to get the data legitimately using the permission-protected Android API; this vulnerability is only used after the app has been explicitly denied access to this data.

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

While OpenX was the only SDK that we observed exploiting this side channel, we searched our entire app corpus for the string `/proc/net/arp`, and found multiple third-party libraries that included it. In the case of one of them, `igexin`, there are existing reports of their predatory behaviour [15]. In our case, log files indicated that after `igexin` was denied permission to scan for WiFi, it read `/system/xbin/ip`, ran `/system/bin/ifconfig`, and then ran `cat /proc/net/arp`. Table 2 shows the prevalence

[4] Using commodity hardware, the MD5 for every possible MAC address can be calculated in a matter of minutes [40].

of third-party libraries with code to access the ARP cache.

Router UPnP One SDK in Table 2 includes another technique to get the MAC address of the WiFi access point: it uses UPnP/SSDP discovery protocols. Three of Peel's smart remote control apps (tv.peel.samsung.app, tv.peel.smartremote, and tv.peel.mobile.app) connected to `192.168.0.1`, the IP address of the router that was their gateway to the Internet. The router in this configuration was a commodity home router that supports universal plug-and-play; the app requested the `igd.xml` (Internet gateway device configuration) file through port 1900 on the router. The router replied with, among other manufacturing details, its MAC address as part of its UUID. These apps also sent WiFi MAC addresses to their own servers and a domain hosted by Amazon Web Services.

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

4.4 Geolocation

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

We identified 70 different apps sending location data to 45 different domains without having any of the location permissions. Most of these location transmissions were not caused by circumvention of the permissions system, however, but rather the location data was provided within incoming packets: ad mediation services provided the location data embedded within the ad link. When we retested the apps in a different location, however, the returned location was no longer as precise, and so we suspect that these ad mediators were using IP-based geolocation, though with a much higher degree of precision than is normally expected. One app explicitly used `www.googleapis.com`'s IP-based geolocation and we found that the returned location was accurate to within a few meters; again, however, this accuracy did not replicate when we retested elsewhere [59]. We did, however, discover one genuine side channel through photo EXIF data.

Shutterfly and EXIF Metadata We observed that the Shutterfly app (com.shutterfly) sends precise geolocation data to its own server (`apcmobile.thislife.com`) without holding a location permission. Instead, it sent photo metadata from the photo library, which included the phone's precise location in its exchangeable image file format (EXIF) data. The app actually processed the image file: it parsed the EXIF metadata—including location—into a JSON object with labelled `latitude` and `longitude` fields and transmitted it to their server.

While this app may not be intending to circumvent the permission system, this technique can be exploited by a malicious actor to gain access to the user's location. Whenever a new picture is taken by the user with geolocation enabled, any app with read access to the photo library (*i.e.*, READ_EXTERNAL_STORAGE) can learn the user's precise location when said picture was taken. Furthermore, it also allows obtaining his-

torical geolocation fixes with timestamps from the user, which could later be used to infer sensitive information about that user.

5 Related Work

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

Covert Channels Marforio *et al.* [49] proposed several scenarios to transmit data between two Android apps, including the use of UNIX sockets and external storage as a shared buffer. In our work we see that the shared storage is indeed used in the wild. Other studies have focused on using mobile noises [26, 69] and vibrations generated by the phone (which could be inaudible to users) as covert channels [3, 17]. Such attacks typically involve two physical devices communicating between themselves. This is outside of the scope of our work, as we focus on device vulnerabilities that are being exploited by apps and third parties running in user space.

Side Channels Spreitzer *et al.* provided a good classification of mobile-specific side-channels present in the literature [73]. Previous work has demonstrated how unprivileged Android resources could be to used to infer personal information about mobile users, including unique identifiers [71] or gender [51]. Researchers also demonstrated that it may be possible to identify users' locations by monitoring the power consumption of their phones [52] and by sensing publicly available Android resources [90]. More recently, Zhang *et al.* demonstrated a sensor calibration fingerprinting attack that uses unprotected calibration data gathered from sensors like the accelerometer, gyroscope, and magnetometer [89]. Others have shown that unprotected system-wide information is enough to infer input text in gesture-based keyboards [71]. Research papers have also reported techniques that leverage lowly protected network information to geolocate users at the network level [54, 81, 2]. We extend previous work by reporting third-party libraries and mobile applications that gain access to unique identifiers and location information in the wild by exploiting side and covert channels.

6 Discussion

Our work shows a number of side and covert channels that are being used by apps to circumvent the Android permissions system. The number of potential users impacted by these findings is in the hundreds of millions. In this section, we discuss how these issues are likely to defy users' reasonable expectations, and how these behaviours may constitute violations of various laws.

We note that these exploits may not necessarily be malicious and intentional. The Shutterfly app that extracts geolocation information from EXIF metadata may not be doing this to learn location information about the user or may not be using this data later for any purpose. On the other hand, cases where an app contains

both code to access the data through the permission system and code that implements an evasion do not easily admit an innocent explanation. Even less so for those containing code to legitimately access the data and then store it for others to access. This is particularly bad because covert channels can be exploited by *any app* that knows the protocol, not just ones sharing the same SDK. The fact that Baidu writes user's IMEI to publicly accessible storage allows any app to access it without permission—not just other Baidu-containing apps.

6.1 Privacy Expectations

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

Nissenbaum's "Privacy as Contextual Integrity" framework defines privacy violations as data flows that defy contextual information norms [55]. In Nissenbaum's framework, data flows are modeled by senders, recipients, data subjects, data types, and transmission principles in specific contexts (*e.g.*, providing app functionality, advertising, etc.). By circumventing the permissions system, apps are able to exfiltrate data to their own servers and even third parties in ways that are likely to defy users' expectations (and societal norms), particularly if it occurs after having just denied an app's explicit permission request. That is, regardless of context, were a user to explicitly be asked about granting an app access to personal information and then explicitly declining, it would be reasonable to expect that the data then would not be accessible to the app. Thus, the behaviours that we document in this paper constitute clear privacy violations. From a legal and policy perspective, these practices are likely to be considered deceptive or otherwise unlawful.

Both a recent CNIL decision (France's data protection authority), with respect to GDPR's notice and consent requirements, and various FTC cases, with respect to unfair and deceptive practices under U.S. federal law—both described in the next section—emphasize the notice function of the Android permissions system from a consumer expectations perspective. Moreover, these issues are also at the heart of a recent complaint brought by the Los Angeles County Attorney (LACA) under the California State Unfair Competition Law. The LACA complaint was brought against a popular mobile weather app on related grounds. The case further focuses on the permissions system's notice function, while noting that, "users have no reason to seek [geolocation data collection] information by combing through the app's lengthy [privacy policy], buried within which are opaque discussions of [the developer's] potential transmission of geolocation data to third parties and

use for additional commercial purposes. Indeed, on information and belief, the vast majority of users do not read those sections at all" [75].

6.2 Legal and Policy Issues

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

The Federal Trade Commission (FTC), which is charged with protecting consumer interests, has brought a number of cases under Section 5 of the Federal Trade Commission (FTC) Act [78] in this context. The underlying complaints have stated that circumvention of Android permissions and collection of information absent users' consent or in a manner that is misleading is an unfair and deceptive act [83]. One case suggested that apps requesting permissions beyond what users expect or what are needed to operate the service were found to be "unreasonable" under the FTC Act. In another case, the FTC pursued a complaint under Section 5 alleging that a mobile device manufacturer, HTC, allowed developers to collect information without obtaining users' permission via the Android permission system, and failed to protect users from potential third-party exploitation of a related security flaw [80]. Finally, the FTC has pursued cases involving consumer misrepresentations with respect to opt-out mechanisms from tailored advertising in mobile apps more generally [82].

Also in the United States, state-level Unfair and Deceptive Acts and Practices (UDAP) statutes may also apply. These typically reflect and complement the corresponding federal law. Finally, with growing regulatory and public attention to issues pertaining to data privacy and security, data collection that undermines users' expectations and their informed consent may also be in violation of various general privacy regulations, such as the Children's Online Privacy Protection Act (COPPA) [79], the recent California Privacy Protection Act (CCPA), and potentially data breach notification laws that focus on unauthorized collection, depending on the type of personal information collected.

In Europe, these practices may be in violation of GDPR. In a recent landmark ruling, the French data regulator, CNIL, levied a 50 million Euro fine for a breach of GDPR's transparency requirements, underscoring informed consent requirements concerning data collection for personalized ads [25]. This ruling also suggests that—in the context of GDPR's consent and transparency provisions—permission requests serve a key function of both informing users of data collection practices and as a mechanism for providing informed consent [80].

Our analysis brings to light novel permission circumvention methods in actual use by otherwise legitimate Android apps. These circumventions enable the collection of information either without asking for consent or after the user has explicitly refused to provide consent, likely undermining users' expectations and potentially violating key privacy and data protection requirements on a state, federal, and even global level. By uncovering these practices and making our data public,[5] we hope to provide suffi-

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

cient data and tools for regulators to bring enforcement actions, industry to identify and fix problems before releasing apps, and allow consumers to make informed decisions about the apps that they use.

7 Limitations and Future Work

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

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

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

Many popular apps also use certificate pinning [61, 28], which results in them rejecting the custom certificate used by our man-in-the-middle proxy; our system then allows apps to continue without interference. Certificate pinning is reasonable behaviour from a security standpoint; it is possible, however, that it is being used to thwart attempts to analyse and study the network traffic of a user's mobile phone.

Our dynamic analysis uses the Android Exerciser Monkey as a UI fuzzer to generate random UI events to interact with the apps. While in our prior work we found that the Monkey explored similar code branches as a human for 60% of the apps tested [**?**], it is likely that it still fails to explore some code branches that may exploit covert and side channels. For example, the Monkey fails to interact with apps that require users to interact with login screens or, more generally, require specific inputs to proceed. Such apps are consequently not as comprehensively tested as apps amenable to automated exploration. Future work should compare our approaches to more sophisticated tools for automated exploration, such as Moran et al.'s Crashscope [53], which generates inputs to an app designed to trigger crash events.

Ultimately, these limitations only result in the possibility that there are side and covert channels that we have not yet discovered (*i.e.*, false negatives). It has no impact on the validity of

the channels that we did uncover (*i.e.*, there are no false positives) and improvements on our methodology can only result in the discovery of more of these channels.

Moving forward, there has to be a collective effort coming from all stakeholders to prevent apps from circumventing the permissions system. Google, to their credit, have announced that they are addressing many of the issues that we reported to them [33]. However, these fixes will only be available to users able to upgrade to Android Q—those with the means to own a newer smartphone. This, of course, positions privacy as a luxury good, which is in conflict with Google's public pronouncements [58]. Instead, they should treat privacy vulnerabilities with the same seriousness that they treat security vulnerabilities and issue hotfixes to all supported Android versions.

Regulators and platform providers need better tools to monitor app behaviour and hold app developers accountable by ensuring apps comply with applicable laws, namely by protecting users' privacy and respecting their data collection choices. Society should support more mechanisms, technical and other, that empower users' informed decision-making with greater transparency into what apps are doing on their devices. To this end, we have made the list of all apps that exploit or contain code to exploit the side and covert channels we discovered available online [8].

Acknowledgments

This work was supported by the U.S. National Security Agency's Science of Security program (contract H98230-18-D-0006), the Department of Homeland Security (contract FA8750-18-2-0096), the National Science Foundation (grants CNS-1817248 and grant CNS-1564329), the Rose Foundation, the European Union's Horizon 2020 Innovation Action program (grant Agreement No. 786741, SMOOTH Project), the Data Transparency Lab, and the Center for Long-Term Cybersecurity at U.C. Berkeley. The authors would like to thank John Aycock, Irwin Reyes, Greg Hagen, René Mayrhofer, Giles Hogben, and Refjohürs Lykkewe.

References

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

[2] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. WifiLeaks: Underestimated privacy implications of the access wifi state Android permission. Technical Report EURECOM+4302, Eurecom, 05 2014.

[3] A. Al-Haiqi, M. Ismail, and R. Nordin. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[5] Android Documentation. App Manifest Overview. `https://developer.android.com/guide/topics/manifest/manifest-intro`, 2019. Accessed: February 12, 2019.

[6] Android Studio. UI/Application Exerciser Monkey. `https://developer.android.`

com/studio/test/monkey.html, 2017. Accessed: October 12, 2017.

[7] Apktool. Apktool: A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

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

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI*, pages 259–269, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[12] G. S. Babil, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–8, July 2013.

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

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

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

[16] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, San Diego, CA, 2014. USENIX Association.

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

[18] S. Cabuk, C. E. Brodley, and C. Shields. IP covert channel detection. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):22, 2009.

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and

Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of NDSS*, 2015.

[20] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[21] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[22] M. Christodorescu, S. Jha, S. A Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[23] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2017.

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

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

[26] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *USENIX WOOT*, 2014.

[27] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[29] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 414–421. IEEE, 2014.

[30] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, page 3, New York, NY, USA, 2012. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov.

The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[32] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

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

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

[35] Google, Inc. Distribution dashboard. `https://developer.android.com/about/dashboards`, May 7 2019. Accessed: June 1, 2019.

[36] Google Play. Usage of Google Advertising ID. `https://play.google.com/about/monetization-ads/ads/ad-id/`, 2019. Accessed: February 12, 2019.

[37] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[38] C. Han, I. Reyes, A. Elazari Bar On, J. Reardon, Á. Feal, S. Egelman, and N. Vallina-Rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. In *Workshop on Technology and Consumer Protection*, ConPro '19, 2019.

[39] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

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

[41] J. Kim, Y. Yoon, K. Yi, J. Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[42] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[43] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[44] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Workshop on*

Crowdsourcing and Crowdsharing of Big (Internet) Data, pages 15–20, 2015.

[45] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of ACM HotMobile*, page 2, 2012.

[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *badgers*, pages 3–17, 2014.

[47] M. Liu, H. Wang, Y. Guo, and J. Hong. Identifying and Analyzing the Privacy of Apps for Kids. In *Proc. of ACM HotMobile*, 2016.

[48] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[49] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[50] E. McReynolds, S. Hubbard, T. Lau, A. Saraf, M. Cakmak, and F. Roesner. Toys That Listen: A Study of Parents, Children, and Internet-Connected Toys. In *Proc. of ACM CHI*, 2017.

[51] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, pages 1053–1067, 2014.

[52] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, pages 785–800, 2015.

[53] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.

[54] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang. Unlocin: Unauthorized location inference on smartphones without being caught. In *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8. IEEE, 2013.

[55] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79:119, February 2004.

[56] United Kingdom of Great Britain and Northern Ireland. Mobile telephones (re-programming) act. `http://www.legislation.gov.uk/ukpga/2002/31/introduction`, 2002.

[57] OpenX. Why we exist. https://www.openx.com/company/, 2019.

[58] Sundar Pichai. Privacy Should Not Be a Luxury Good. The New York Times, May 7 2019. `https://www.nytimes.com/2019/`

05/07/opinion/google-sundar-pichai-privacy.html.

[59] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[60] A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 65–66. ACM, 2012.

[61] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362. ACM, 2017.

[62] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[63] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, 2015.

[64] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug fixes, improvements,... and privacy leaks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[65] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In *Proceedings of the ACM SIGMOBILE MobiSys*, pages 361–374, 2016.

[66] Salmonads. About us. http://publisher.salmonads.com, 2016.

[67] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[68] Sarah Schafer. With capital in panic, pizza deliveries soar. The Washington Post, December 19 1998. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm.

[69] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[70] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010*

IEEE Symposium on Security and Privacy, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[71] L. Simon, W. Xu, and R. Anderson. Don't interrupt me while i type: Inferring text entered through gesture typing on android keyboards. *Proceedings on Privacy Enhancing Technologies*, 2016(3):136–154, 2016.

[72] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of ACM SPSM*, 2015.

[73] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[74] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. `https://www.statista.com/statistics/266136`, 2019. Accessed: February 11, 2019.

[75] COUNTY OF LOS ANGELES SUPERIOR COURT OF THE STATE OF CALIFORNIA. Complaint for injunctive relief and civil penalties for violations of the unfair competition law. `http://src.bna.com/EqH`, 2019.

[76] Unity Technologies. Unity 3d. `https://unity3d.com`, 2019.

[77] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.W. Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 145–162. USENIX Association, 2017.

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

[79] U.S. Federal Trade Commission. Children's online privacy protection rule ("coppa"). `https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule`, November 1999.

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

[81] U.S. Federal Trade Commission. Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. `https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked`, June 22 2016.

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

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

[84] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[85] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, May 2017.

[86] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[87] B. Yankson, F. Iqbal, and P.C.K. Hung. Privacy preservation framework for smart connected toys. In *Computing in Smart Toys*, pages 149–164. Springer, 2017.

[88] S. Yong, D. Lindskog, R. Ruhl, and P. Zavarsky. Risk Mitigation Strategies for Mobile Wi-Fi Robot Toys from Online Pedophiles. In *Proc. of IEEE SocialCom*, pages 1220–1223. IEEE, 2011.

[89] J. Zhang, A. R. Beresford, and I. Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[90] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM, 2013.

[91] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. Automated analysis of privacy requirements for mobile apps. In *24th Network & Distributed System Security Symposium*, 2017.

50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System

Joel Reardon
University of Calgary
AppCensus, Inc.

Álvaro Feal
IMDEA Networks Institute
Universidad Carlos III de Madrid

Primal Wijesekera
U.C. Berkeley / ICSI

Amit Elazari Bar On
U.C. Berkeley

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

Serge Egelman
U.C. Berkeley / ICSI
AppCensus, Inc.

Resumen

Las plataformas modernas de teléfonos inteligentes implementan modelos basados en permisos para proteger el acceso a datos confidenciales y recursos del sistema.Sin embargo, las aplicaciones pueden eludir el modelo de permisos y obtener acceso a datos protegidos sin el consentimiento del usuario mediante el uso de canales encubiertos y laterales. Los canales laterales, presentes en la implementación del sistema de permisos, permiten a las aplicaciones acceder a datos protegidos y recursos del sistema sin permiso; mientras que los canales encubiertos permiten la comunicación entre dos aplicaciones en conflicto para que una aplicación puede compartir sus datos protegidos por permisos con otra aplicación que carece de esos permisos. Ambos plantean amenazas a la privacidad del usuario. En este trabajo, hacemos uso de nuestra infraestructura que ejecuta cientos de miles de aplicaciones en un entorno instrumentado. Este entorno de prueba incluye mecanismos para monitorizar el comportamiento de las aplicaciones y las comunicaciones a Internet durante el tiempo de ejecución. Buscamos evidencia de canales laterales y encubiertos usados en la práctica buscando datos confidenciales que se envían a través de la red por aplicaciones que no tenían permisos para acceder a ella. Después usamos ingeniería inversa en las aplicaciones y bibliotecas de terceros responsables de estos comportamientos para determinar cómo ocurrió el acceso no autorizado. También usamos métodos de huellas digitales de software para medir la prevalencia de la técnica que descubrimos en otras aplicaciones de nuestro corpus. Usando este entorno y método de prueba, descubrimos un número de canales laterales y encubiertos en uso activo por cientos de aplicaciones populares y SDK de terceros para obtener acceso no autorizado a identificadores únicos y datos de geolocalización. Hemos revelado responsablemente nues-

tros hallazgos a Google y hemos recibido una recompensa por nuestro trabajo.

1. Introducción

Los teléfonos inteligentes se usan como computadoras de uso general y, por lo tanto, tienen acceso a una gran cantidad de recursos sensibles del sistema (p. ej., sensores como la cámara, micrófono o GPS), datos privados del usuario (por ejemplo, correo electrónico o contactos del usuario lista) y varios identificadores persistentes (p. ej., IMEI). Es crucial proteger esta información de acceso no autorizado. Android, el sistema operativo más popular en teléfonos inteligentes [74], implementa un sistema basado en permisos para regular el acceso a estos recursos sensibles por aplicaciones de terceros. En este modelo, los desarrolladores deben solicitar permiso explícitamente para acceder a recursos confidenciales en su archivo de manifiesto de Android [5]. Se supone que este modelo da a los usuarios control para decidir qué aplicaciones pueden acceder a qué recursos e información; en la práctica no aborda el problema por completo [30, 85].

El sistema operativo Android protege las aplicaciones de espacio de usuario para evitar que interactúen arbitrariamente con otras aplicaciones en ejecución. Android implementa aislamiento al asignar a cada aplicación una ID de usuario separada además de otros controles de acceso. Los controles se implementan utilizando SELinux. Cada proceso en ejecución de una aplicación puede ser código de la aplicación en sí o de las bibliotecas SDK incrustadas en la aplicación; estos SDK pueden provenir de Android (por ejemplo, bibliotecas oficiales de soporte de Android) o de proveedores externos. Los desarrolladores de aplicaciones integran bibliotecas de terceros en su software para cosas como informes de fallos, soporte de desarrollo, servicios de analítica, integración de redes sociales y publicidad [62, 16]. Por diseño, cualquier servicio de terceros incluido en una aplicación de Android hereda el acceso a todos los recursos protegidos por permisos que el usuario otorga a la aplicación. En otras palabras, si una aplicación puede acceder a la ubicación del usuario, luego todos los servicios de terceros integrados en esa aplicación también pueden.

En la práctica, los mecanismos de seguridad a menudo se pueden eludir; los canales laterales y los canales encubiertos son dos técnicas comunes para evadir un mecanismo de seguridad. Estos canales se producen cuando hay un medio alternativo para acceder a un recurso protegido que no es auditado por el mecanismo de seguridad. Un canal lateral expone una ruta a un recurso que está fuera del mecanismo de seguridad; esto puede deberse a un fallo en el diseño de el mecanismo de seguridad o una fallo en la implementación del diseño. Un buen ejemplo de un canal lateral es analizar el uso de energía del hardware cuando se realiza las operaciones criptográficas para descubrir los detalles de una clave secreta [42]. Como un ejemplo en el mundo físico, la frecuencia de entregas de pizza a los edificios del gobierno pueden filtrar información sobre crisis políticas [68].

Un canal encubierto es un esfuerzo más deliberado e intencional entre dos entidades cooperantes para que una con acceso a algunos datos se los proporcione a la otra entidad sin acceso a los datos en violación del mecanismo de seguridad [43]. Como ejemplo, alguien podría ejecutar un algoritmo que alterna entre carga de CPU alta y baja para pasar un mensaje binario a otra parte observando dicha carga

de la CPU.

La comunidad de investigación ha explorado previamente el potencial de los canales encubiertos en Android con sockets locales y almacenamiento compartido [49], así como otros medios poco ortodoxos, como vibraciones y datos del acelerómetro para enviar y recibir datos entre dos aplicaciones coordinadas [3]. Los ejemplos de canales laterales incluyen la utilización de los sensores del dispositivo para inferir el género del usuario [51] o identificar de forma única a un usuario [71]. Más recientemente, los investigadores demostraron una nueva técnica para generar huellas digitales sin permisos que permite rastrear dispositivos Android e iOS a través de Internet utilizando los detalles de calibración del sensor configurados de fábrica [89]. Sin embargo, ha habido poca investigación en la detección y medición a escala de la prevalencia de canales encubiertos y laterales en aplicaciones que están disponibles en Google Play Store. Solo ha habido instancias aisladas de aplicaciones o bibliotecas maliciosas que infieren las ubicaciones de los usuarios a través de los puntos de acceso WiFi, un canal lateral que fue abusado en la práctica y que resultó en una multa dun millón de dólares por parte de los reguladores [81].

De hecho, la mayor parte de la literatura existente se centra en comprender como las aplicaciones recopilan datos personales utilizando los mecanismos de control de acceso compatibles con el sistema (es decir, los permisos de Android). Con una mayor atención reguladora a la privacidad de datos y problemas relacionados con el consentimiento del usuario, creemos que es imprescindible comprender la efectividad (y limitaciones) del sistema de permisos y si está siendo evitado como un paso preliminar hacia la implementación de defensas efectivas.

Con este fin, ampliamos el estado del arte desarrollando métodos para detectar la elusión real del sistema de permisos de Android, a escala en aplicaciones reales utilizando una combinación de análisis dinámico y estático. Ejecutamos automáticamente más de 88,000 aplicaciones de Android en un entorno altamente instrumentado con capacidad para supervisar el comportamiento de las aplicaciones a nivel de sistema y de red, incluido un Proxy TLS man-in-the-middle. En resumen, ejecutamos aplicaciones para ver cuándo el dispositivo transmitió datos protegidos sin permiso y escaneamos las aplicaciones para ver cómo se accedió a los datos protegidos sin permiso. Agrupamos nuestros hallazgos por a dónde en Internet se envió el tipo de datos, ya que esto nos permite atribuir las observaciones al desarrollador de aplicaciones o bibliotecas integradas de terceros. Luego realizamos ingeniería inversa para determinar exactamente cómo se accedió a los datos. Finalmente, analizamos estáticamente todo nuestro conjunto de datos para medir la prevalencia del canal. Nos centramos en un subconjunto de los permisos peligrosos que impiden a las aplicaciones el acceso a los datos de ubicación e identificadores. En lugar de imaginar nuevos canales, nuestro trabajo se enfoca en rastrear evidencia que sugiera que el canal lateral y encubierto está ocurriendo en la práctica. Estudiamos más de 88,000 aplicaciones en cada categoría de la Google Play Store de EE. UU. Encontramos una serie de canales laterales y encubiertos en uso activo, divulgando nuestros hallazgos a Google y a la Comisión Federal de Comercio de EE. UU. (FTC), recibiendo una recompensa por nuestros esfuerzos encontrando errores.

En resumen, las contribuciones de este trabajo incluyen:

- Diseñamos una metodología para descubrir automáticamente vulnerabilidades en el Sistema de permisos de Android a través de una combinación de análisis dinámico y estático, creando un entorno escalable.
- Probamos nuestra metodología en más de 88,000 aplicaciones y descubrimos una cantidad de vulnerabilidades, que revelamos de manera responsable. Estas aplicaciones fueron descargadas desde la Google Play Store de EE. UU. e incluyen aplicaciones populares de todas las categorías. Describimos las vulnerabilidades en detalle y medimos el grado en que están en uso activo y, por lo tanto, representan una amenaza para los usuarios. Descubrimos canales encubiertos y laterales utilizados en la naturaleza que comprometen tanto los datos de ubicación de los usuarios como los identificadores persistentes.
- Descubrimos compañías que obtienen las direcciones MAC de las estaciones base WiFi a través de la caché ARP. Esto se puede usar como sustituto para datos de ubicación. Encontramos 5 aplicaciones que explotan esta vulnerabilidad y 5 con el código pertinente para hacerlo.
- Descubrimos que Unity obtiene la dirección MAC del dispositivo usando llamadas del sistema ioctl. La dirección MAC se puede usar para identificar de forma exclusiva el dispositivo. Encontramos 42 aplicaciones que explotan esta vulnerabilidad y 12,408 aplicaciones con el código pertinente para hacerlo.
- También descubrimos que las bibliotecas de terceros provistas por dos compañías chinas, Baidu y Salmonads, hacen uso independiente de la tarjeta SD como un canal secreto, de modo que cuando una aplicación puede leer el IMEI del teléfono, lo almacena para otras aplicaciones que no pueden. Encontramos 159 aplicaciones con potencial para explotar este canal secreto y empíricamente encontramos 13 aplicaciones haciéndolo.
- Encontramos one aplicación que usaba los metadatos de imágenes como un canal lateral para acceder a la información de ubicación precisa a pesar de no tener permisos de ubicación.

Estas prácticas engañosas permiten a los desarrolladores acceder a los datos privados de los usuarios sin consentimiento, socavando la privacidad del usuario y dando lugar a preocupaciones tanto legales como éticas. La legislación de protección de datos en todo el mundo, incluido el Reglamento General de Protección de Datos (RGPD) en Europa, California Consumer Pri- Vacy Act (CCPA) y las leyes de protección al consumidor, como la Comisión Federal de Comercio (FTC) hacen cumplir la transparencia en la recopilación, procesamiento y prácticas de envío por parte de las aplicaciones móviles.

Este documento está organizado de la siguiente manera: la sección 2 brinda más información básica sobre los conceptos discutidos en la introducción. La sección 3 describe nuestra sistema para descubrir vulnerabilidades en detalle. La sección 4 proporciona los resultados de nuestro estudio, incluidos los canales secundarios y encubiertos que descubrimos y su prevalencia en la práctica. La sección 5 describes el trabajo relacionado. La sección 6 discute las posibles implicaciones legales. La sección 7 discute las limitaciones de nuestro método y concluye con el trabajo futuro.

2. Antecedentes

El sistema de permisos de Android ha evolucionado a lo largo de los años desde una consulta a la hora de instalar la aplicación hasta preguntar en el primer uso. Si bien este cambio impacta la forma en la que usuarios otorgan los permisos y cómo los usuarios pueden usar la información contextual para razonar sobre la solicitud de permiso, el backend hace cumplir Los mecanismos de mantenimiento han permanecido en gran medida sin cambios. Nos fijamos en cómo el diseño y las aplicaciones han explotado la implementación del modelo de permisos para pasar estas protecciones.

2.1. Permisos de Android

El sistema de permisos de Android se basa en el principio de seguridad de menor privilegio , es decir, una entidad solo debe tener las capacidades mínimas que necesita para realizar su tarea. Este principio de diseño estándar para la seguridad implica que si una aplicación actúa maliciosamente, el daño será limitado. Los desarrolladores deben declarar los permisos que sus aplicaciones necesitan de antemano, y el usuario tiene una oportunidad para revisarlos y decidir si instalar la aplicación. Sin embargo, la plataforma de Android no juzga si el conjunto de permisos solicitados son solamente los estrictamente necesario para que la aplicación funcione. Los desarrolladores son libres de solicitar más permisos de los que realmente necesitan y se espera que los usuarios juzguen si estos son razonables.

El modelo de permiso de Android tiene dos aspectos importantes: obtener consentimiento del usuario antes de que una aplicación pueda acceder a cualquiera de los recursos protegidos por permisos, y luego asegurarse de que la aplicación no pueda acceder a los recursos para los cuales el usuario no ha otorgado su consentimiento. Hay una larga línea de trabajo para descubrir problemas sobre cómo interactúa el modelo de permisos con el usuario: los usuarios no son adecuadamente informados sobre por qué las aplicaciones necesitan permisos en el momento de la instalación, los usuarios no pueden comprender exactamente cuál es el propósito de los diferentes permisos, y hay una falta de transparencia sobre cómo las aplicaciones utilizarán en última instancia el contenido obtenido a través de los permisos [30, 77, 85, 24]. Si bien todos estos son problemas críticos que necesitan atención, el enfoque de nuestro trabajo es comprender cómo las aplicaciones están eludiendo el sistema que verifica que las aplicaciones hayan recibido varios permisos.

Cuando una aplicación solicita un recurso protegido con permiso, el administrador de recursos(*e.g.*, `LocationManager`, `WiFiManager`, etc.) contacta a `ActivityServiceManager`, que es el monitor de referencia en Android. La solicitud de recurso se origina en la aplicación de forma asolada, y la verificación final ocurre dentro de la plataforma Android. La plataforma es un sistema operativo Java que se ejecuta en el espacio del sistema y actúa como una interfaz para un kernel Linux personalizado, aunque las aplicaciones también pueden interactuar con el kernel directamente. Para algunos recursos protegidos con permiso, como los sockets de red, el monitor de referencia es el kernel y la solicitud de dichos recursos omite el marco de la plataforma y contactan directamente con el kernel. Nuestro trabajo analiza cómo las aplicaciones del mundo real eluden estas comprobaciones del sistema colocado en el núcleo y

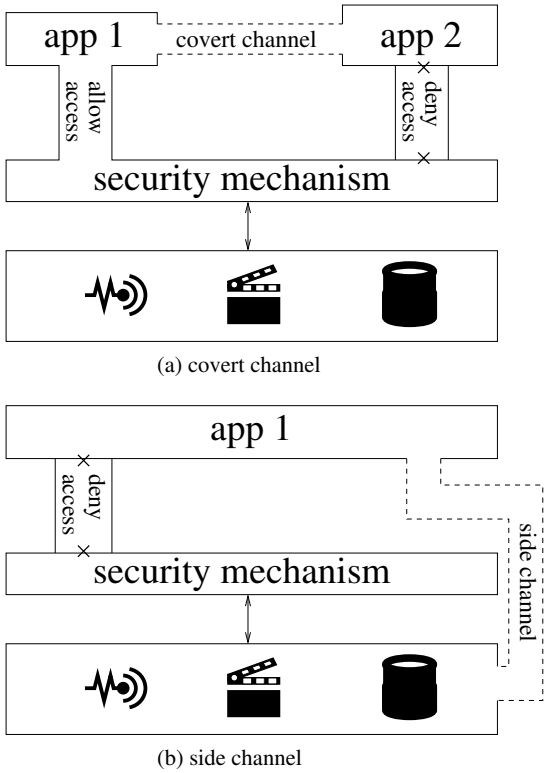
las capas de la plataforma.

El sistema de permisos de Android tiene un propósito importante: proteger la privacidad de los usuarios y los recursos confidenciales del sistema de actores engañosos, maliciosos y abusivos Por lo menos, si un usuario niega a una aplicación un permiso, entonces esa aplicación no debería poder acceder a los datos protegidos por ese permiso [80, 24].En la práctica, este no es siempre el caso.

2.2. Circunvención

Las aplicaciones pueden eludir el modelo de permisos de Android de diferentes maneras [49, 69, 3, 17, 54, 73, 71, 51, 52]. Sin embargo, el uso de canales encubiertos y laterales son particularmente problemáticos ya que su uso indica prácticas que podrían engañar incluso a los usuarios más diligentes, al tiempo que subraya una vulnerabilidad de seguridad en el sistema operativo. De hecho, la Comisión Federal de Comercio de los Estados Unidos (FTC) ha multado a desarrolladores móviles y bibliotecas de terceros por explotar canales secundarios: utilizando la dirección MAC del punto de acceso WiFi para inferir la ubicación del usuario [81]. La figura 1 ilustra la diferencia entre canales encubiertos y laterales y muestra cómo una aplicación a la que un mecanismo de seguridad le niega el permiso todavía puede acceder a esa información.

Canal encubierto Un canal encubierto es una ruta de comunicación entre dos partes (por ejemplo, dos aplicaciones móviles) que les permite transferir información que el mecanismo de seguridad relevante considera que el destinatario no está autorizado para recibir [18]. Por



(a) covert channel

(b) side channel

Figura 1: Canales encubiertos y laterales. (a) Un mecanismo de seguridad permite que la app1 acceda a los recursos pero niega el acceso a la app2; app2 se salta esta restricción usando a app1 como fachada para obtener acceso a través de un canal de comunicación no monitorizado por el mecanismo de seguridad. (b) Un mecanismo de seguridad niega el acceso de la app1 a recursos; esto se evita al acceder a los recursos a través de un canal lateral que evita el mecanismo de seguridad.

ejemplo, imagine que a AliceApp se le ha otorgado permiso a través de la API de Android para acceder al IMEI del teléfono (un identificador persistente), pero a BobApp se le ha denegado el acceso a esos mismos datos. Un canal secreto es creado cuando AliceApp lee legítima-

mente el IMEI y luego se lo da a BobApp, a pesar de que a BobApp ya se le ha denegado el acceso a estos mismos datos cuando los ha solicitdo a través de las API de Android protegidas por permisos.

En el caso de Android, se han propuesto diferentes canales encubiertos para habilitar la comunicación entre aplicaciones. Estos incluye medios exóticos como el ultrasonido. balizas de audio y vibraciones [26, 17]. Las aplicaciones también pueden comunicarse usando un servidor de red externo para intercambiar información cuando no hay otra oportunidad .Nuestro trabajo, sin embargo, expone que los canales encubiertos rudimentarios, como el almacenamiento compartido, se utilizan en la práctica a escala.

Canal lateral Un canal lateral es una ruta de comunicación que permite a una parte obtener información privilegiada sin que se realicen verificaciones de permisos relevantes. Esto puede deberse a la posibilidad de inferir dicha información a través de métodos no convencionales, así como a versiones anteriores de la misma información disponibles sin protección. Un ejemplo clásico de un ataque de canal lateral es el ataque temporal para extraer una clave de cifrado del almacenamiento seguro [42]. El sistema bajo ataque es un algoritmo que realiza el cálculo con la clave secreta y gotea información involuntariamente, es decir, según el tiempo que tarde el cálculo se revela información crítica sobre la clave.

Los canales laterales son típicamente una consecuencia no intencional de un sistema complicado. (Las ”Puertas traseras”son canales laterales creados intencionalmente). En Android, una API grande y complicada resulta en que los mismos datos aparezcan en diferentes ubicaciones, cada uno gobernado por un control de acceso diferente. Cuando una API está protegida con permisos, otra no protegida se puede utilizar para obtener los mismos datos u otra versión de los mismos.

2.3. Métodos de análisis de aplicaciones

Los investigadores usan dos técnicas principales para analizar el comportamiento de la aplicación: análisis estático y análisis dinámico En resumen, el análisis estático estudia el software como datos a través del código; el análisis dinámico estudia el software como código ejecutándolo. Ambos enfoques tienen el objetivo de comprender el comportamiento final del software, pero ofrecen ideas con diferente certeza y granularidad: el análisis estático informa del comportamiento hipotético; el análisis dinámico proporciona informes del comportamiento observado.

Análisis estático El análisis estático implica escanear el código en busca de todas los posibles combinaciones de flujos de ejecución para comprender los posibles comportamientos en tiempo de ejecución: los comportamientos de interés pueden incluir varias violaciones de privacidad (por ejemplo, acceso a datos confidenciales del usuario). Varios estudios han utilizado el análisis estático para analizar diferentes tipos de software en busca de comportamientos maliciosos y filtraciones de privacidad [21, 10, 39, 20, 22, 37, 45, 9, 11, 19, 32, 41, 4, 91]. Sin embargo, el análisis estático no produce observaciones reales de violaciones de privacidad; solo puede sugerir que una violación puede ocurrir

si una parte determinada del código se ejecuta en tiempo de ejecución. Esto significa que el análisis estático proporciona un límite superior de comportamientos hipotéticos (es decir, produce falsos positivos).

La mayor ventaja del análisis estático es que es fácil de realizar automáticamente y a escala. Los desarrolladores, sin embargo, tienen opciones para evadir la detección por análisis estático porque el comportamiento en tiempo de ejecución de un programa puede diferir enormemente de su apariencia superficial. Por ejemplo, pueden usar ofuscación de código [23, 29, 48] o alterar el flujo del programa para ocultar la forma en que el software funciona en la realidad [23, 29, 48]. La función de reflexión de Java permite la ejecución de instrucciones creadas dinámicamente y código cargado dinámicamente que evade de manera similar el análisis estático. Recientes estudios han demostrado que alrededor del 30 % de las aplicaciones procesan código dinámicamente [46], por lo que el análisis estático puede ser insuficiente en esos casos.

Desde la perspectiva del análisis de una aplicación, el análisis estático carece de la perspectiva contextual, es decir, no puede observar las circunstancias que rodean cada observación de acceso y uso compartido de recursos confidenciales, lo cual es importante para comprender cuándo es probable que ocurra una violación de privacidad determinada. Por estas razones, el análisis estático es útil, pero está bien complementado por análisis dinámico para aumentar o confirmar resultados.

Análisis dinámico El análisis dinámico estudia una aplicación ejecutándo el código y auditando su comportamiento en tiempo de ejecución. Por lo general, el análisis dinámico se beneficia de ejecutar la aplicación en un entorno controlado, como un SO móvil instrumentado [27, 84], para obtener observaciones del comportamiento de una aplicación [16, 32, 46, 47, 50, 65, 72, **?**, 84, 86, 87, 88].

Existen varios métodos que pueden usarse en el análisis dinámico, como el análisis de contaminación [27, 32] que puede ser ineficiente y propenso a ataques de control de flujo [67, 70]. Un desafío para realizar análisis dinámico es la logística de realizarlo a escala. Analizando una sola aplicación de Android de forma aislada es sencillo, pero escalarlo para que ejecutar automáticamente decenas de miles de aplicaciones no lo es. El análisis dinámico de escala se facilita con la ejecución automatizada y la creación de informes de comportamiento. Esto significa que el análisis dinámico efectivo requiere la construcción de un marco de instrumentación para encontrar posibles comportamientos interesantes y luego diseñar un sistema para gestionar el esfuerzo de probar las aplicaciones.

Sin embargo, algunas aplicaciones son resistentes a ser auditadas cuando se ejecutan en modo virtual o en ambientes privilegiados [12, 67]. Esto ha llevado a nuevas técnicas de auditoría que implican la ejecución de aplicaciones en teléfonos reales, como reenviar tráfico a través de un VPN para inspeccionar las comunicaciones de red [63, 44, 60]. Las limitaciones de este enfoque son el uso de técnicas robustas para evitarr los ataques de hombre en el medio [28, 31, 61] y la dificultad de escalabilidad debido a la necesidad de ejecutar aplicaciones con input del usuario.

Una herramienta para ejecutar aplicaciones automáticamente en la plataforma Android es el Monkey ejercitador de IU [6]. El Monkey ge-

nera entradas de usuarios sintéticas, asegurando que ocurran interacciones automáticamente con la aplicación. Sin embargo, el Monkey no tiene contexto a la hora de realizar acciones con la interfaz de usuario, por lo que algunas rutas de código importantes pueden no ejecutarse debido a la naturaleza aleatoria de sus interacciones con la aplicación. Como resultado, esto da un límite inferior de los posibles comportamientos de la aplicación, pero a diferencia del análisis estático, no produce falsos positivos.

Análisis híbrido Los métodos de análisis estático y dinámico se complementan. De hecho, algunos tipos de análisis se benefician de un enfoque híbrido, en el que la combinación de ambos métodos puede aumentar la cobertura, la escalabilidad o la visibilidad de los análisis. Este es el caso de las aplicaciones maliciosas o engañosas que intentan activamente derrotar un método de análisis concreto (p. ej., utilizando ofuscación o técnicas para detectar entornos virtualizados o interceptación TLS). Un enfoque sería primero llevar a cabo un análisis dinámico para clasificar posibles casos sospechosos, basado en observaciones recolectadas, para luego ser examinadas minuciosamente usando análisis estático. Otro enfoque es realizar primero un análisis estático para identificar ramas de código interesantes que luego pueden ser instrumentadas para el análisis dinámico para confirmar las observaciones.

3. Entorno de prueba y tubería de análisis

Nuestra tubería de instrumentación y procesamiento, representada y descrita en la Figura 2, describe en líneas generales nuestra línea de análisis. Las aplicaciones se ejecutan automáticamente y las transmisiones de datos confidenciales se comparan con los datos a los que se permitió el acceso. Aquellas apps sospechosas de utilizar un canal lateral o encubierto se analizan manualmente combinando las ventajas de las técnicas de análisis estático y dinámico para clasificar las aplicaciones sospechosas y analizar en profundidad sus comportamientos. Utilizamos esta metodología para encontrar evidencia del uso de canales encubiertos y laterales en 252,864 versiones de 88,113 diferentes aplicaciones de Android, todas descargadas de la Google Play Store de EE. UU.cutilizando un programa especialmente diseñado para descargar apps de Google Play. Ejecutamos cada versión de la aplicación individualmente en un teléfono móvil físico equipado con un sistema operativo personalizado y un monitor de red. Este banco de pruebas nos permite observar las comportamientos en tiempo de ejecución de las aplicaciones tanto en el sistema operativo como a nivel de red. Podemos observar como las aplicaciones solicitan y acceden a recursos confidenciales y sus prácticas para compartir datos. También tenemos una herramienta de análisis de datos integral para desencriptar los datos recopilados de la red para descubrir posibles prácticas engañosas.

Antes de ejecutar cada aplicación, recopilamos los identificadores y datos protegidos por permisos. Luego ejecutamos cada aplicación mientras recopilamos todo su tráfico de red. Nosotros aplicamos un conjunto de decodificaciones a los flujos de tráfico y buscamos los datos protegidos por permisos en el tráfico descodificado. Grabamos todas las transmisiones y luego buscamos aquellas que contengan datos protegidos con permiso enviados sin que ha-

yamos dado los permisos necesarios. Suponemos que esto se debe al uso de canales laterales y canales encubiertos; es decir, no estamos buscando estos canales, sino más bien buscando evidencia de su uso (es decir, transmisiones de datos protegidos). Entonces, agrupamos las transmisiones sospechosas por el tipo de datos enviados y el destino a donde fueron enviados, porque encontramos que el mismo par de datos-destino refleja que el mismo canal lateral o canal encubierto fue utilizado. Tomamos un ejemplo por grupo y aplicamos ingeniería inversa para determinar cómo la aplicación obtuvo acceso a la información protegida sin el permiso correspondiente.

Finalmente, tomamos huellas de las aplicaciones y bibliotecas encontradas usando canales encubiertos y laterales para identificar estáticamente la presencia del mismo código en otras aplicaciones de nuestro cuerpo de aplicaciones. Una huella digital es cualquier constante, como nombre de archivo específico o mensaje de error, que puede usarse para analizar estáticamente nuestro corpus para determinar si la misma técnica existe en otras aplicaciones que no se activaron durante nuestra fase de análisis dinámico.

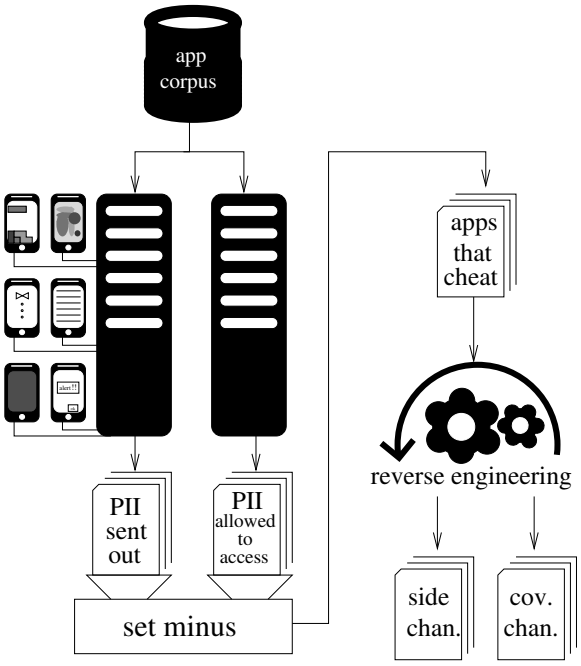


Figura 2: Descripción general de nuestra tubería de análisis. Las aplicaciones se ejecutan automáticamente y las transmisiones de datos confidenciales se comparan con lo que se permitiría. Aquellos sospechosos de utilizar un canal lateral o encubierto son manualmente comprobados con ingeniería inversa.

3.1. Recolección de aplicaciones

Escribimos un programa para descargar de Google Play Store las aplicaciones más populares en cada categoría. Debido a que la distribución de popularidad de las aplicaciones es larga, es probable que el análisis de las 88,113 aplicaciones más populares cubra la mayoría de las aplicaciones que la gente usa actualmente. Esto incluye 1,505 aplicaciones no gratuitas que compramos para otro estudio [38]. Instruimos al programa para que inspeccione Google Play Store para obtener ejecutables de aplicaciones (archivos APK) y sus metadatos asociados (por ejemplo, número de instalaciones, categoría, información del desarrollador, etc.).

Los desarrolladores tienden a actualizar su software de Android para agregar nuevas funcionalidades o para parchear errores [64], pero estas actualizaciones también se pueden usar para introducir nuevos canales laterales y canales encubiertos. Por lo tanto, es importante examinar diferentes versiones de la misma aplicación, porque pueden exhibir comportamientos diferentes. Por lo tanto, nuestro programa ve-

rifica periódicamente si hay una nueva versión de una aplicación ya descargada y la descarga. Este proceso nos permitió crear un conjunto de datos que consta de 252,864 versiones diferentes de 88,113 aplicaciones Android.

3.2. Entorno de análisis dinámico

Implementamos el entorno de pruebas dinámicas descrito en la Figura 2, que consiste en aproximadamente una docena de teléfonos Android Nexus 5X con una versión instrumentada de la plataforma Android Marshmallow [1] Este entorno especialmente diseñado nos permite monitorizar exhaustivamente el comportamiento de cada una de las 88,113 aplicaciones Android en el kernel y a nivel de tráfico de red. Ejecutamos cada aplicación automáticamente usando el Android Automator Monkey [6] para lograr escala eliminando cualquier intervención humana. Almacenamos los registros del sistema operativo resultantes de la ejecución de la app y el tráfico de red en una base de datos para análizarlo más adelante, como discutimos en la Sección 3.3. El análisis dinámico se realiza ampliando una plataforma que hemos utilizado en trabajos anteriores [**?**].

Instrumentación a nivel de plataforma Construimos una versión instrumentada de la plataforma Android 6.0.1 (Marshmallow). La instrumentación monitorizada del código fuente se registra cuando las aplicaciones se instalan y ejecutan. Corremos las aplicaciones de una en una y luego las desinstalamos. Independientemente de la ofuscación, las técnicas que utilizan las aplicaciones para interrumpir el análisis estático, ninguna aplicación puede evitar nuestra instrumentación ya que se ejecuta en el espacio del sistema del marco de Android. En un sentido, nuestro entorno es un honeypot que permite que las aplicaciones se ejecuten de forma natural. Con el propósito de preparar nuestros informes de errores a Google para la divulgación responsable de nuestros hallazgos, volvimos a probar nuestros hallazgos en un Pixel 2 de stock con Android Pie: la versión más reciente en ese momento, para demostrar que aún eran válidos.

Instrumentación a nivel de kernel Construimos e integramos una versión Linux con un kernel personalizado en nuestro entorno de prueba para registrar el acceso de las aplicaciones al sistema de archivos. Este módulo nos permitió saber cada vez que una aplicación abría un archivo para leer o escribir en él. Debido a que instrumentamos las llamadas del sistema para abrir archivos, nuestra instrumentación registró el acceso a tanto archivos regulares como archivos especiales, así como los archivos de la interfaz de dispositivo, y el sistema de archivos /proc debido a la filosofía de ”todo es un archivo”de UNIX. También registramos cada vez que se emitía una llamada ioctl al sistema de archivos. Algunos de los canales laterales para omitir la verificación de permisos en la plataforma de Android puede implicar el acceso directo al kernel y, por lo tanto, a nivel del kernel nuestra instrumentación proporciona evidencia clara de que se utilizan en la práctica.

Ignoramos el archivo especial del dispositivo /dev/ashmem (implementación de memoria compartida asíncrona para la comunicación en-

[1] Mientras que al momento de escribir esto, Android Pie es la versión actual. [35], la mayoría de los usuarios usaban las versiones Marshmallow y anteriores en el momento en que comenzamos la recopilación de datos.

tre procesos Android) porque abruma los registros debido a su uso frecuente. Como Android asigna un usuario separado (es decir, uid) a cada aplicación, podemos atribuir con precisión el acceso a dichos archivos a la aplicación responsable.

Monitoreo a nivel de red Monitorizamos los flujos protegidos por TLS, utilizando una herramienta de monitorización de red desarrollada para nuestras actividades de investigación anteriores [63]. [61]. Este módulo de monitorización de red aprovecha la API VPN los dispositivos Android para redirigir todo el tráfico de red del dispositivo a través de un servicio de host local que inspecciona todo el tráfico de red, independientemente del protocolo utilizado, a través de la inspección de paquetes en espacio de usuario. Reconstruye los flujos de red y los atribuye a la aplicación de origen relacionando a la aplicación que posee el socket de red a través UID según lo informado por el sistema de archivos proc. Además, también realiza interceptación TLS mediante la instalación de un certificado raíz en el sistema certificador central. Esta técnica permite descifrar el tráfico TLS a menos que la aplicación realice técnicas avanzadas de defenda, como la verificación de certificados fijados. Podemos identificar estas técnicas mediante el seguimiento de registros TLS y excepciones de proxy.

Ejecución automática de aplicaciones Dado que nuestro marco de análisis se basa en análisis dinámico, las aplicaciones deben ejecutarse para que nuestra instrumentación pueda monitorizar sus comportamientos. Para escalar a cientos de miles de aplicaciones probadas, nosotros no podemos confiar en la interacción real de un usuario con cada aplicación que se prueba. Como tal, nosotros usamos el Exerciser Monkey de Android, una herramienta proporcionada por el Android SDK para automatizar y paralelizar la ejecución de aplicaciones a base de simular entradas del usuario (es decir, toques, deslizamientos, etc.)

El Monkey inyecta una secuencia pseudoaleatoria de eventos de entrada de usuario simulados en la aplicación, es decir, es un fuzzer de IU. Usamos el Monkey para interactuar con cada versión de cada aplicación por un período de diez minutos, durante el cual las herramientas mencionadas registran la ejecución de la aplicación como resultado de los eventos aleatorios de IU generados por el Monkey. Las aplicaciones se vuelven a ejecutar si la operación falla durante la ejecución. Cada versión de cada aplicación se ejecuta una vez de esta manera; nuestro sistema también vuelve a ejecutar aplicaciones si hay espacio no utilizado.

Después de ejecutar la aplicación, se recopilan los registros del núcleo, la plataforma y la red. La aplicación se desinstala junto con cualquier otra aplicación que pueda haber sido instalada a través del proceso de exploración automática. Hacemos esto con una lista de aplicaciones permitidas; Todas las demás aplicaciones son desinstaladas. Los registros se borran y el dispositivo está listo para usarse para la próxima prueba.

3.3. Información personal en flujos de red

Para detectar si una aplicación ha accedido legítimamente a un recurso dado comparamos su comportamiento en tiempo de ejecución con los permisos que tenía. Tanto los usuarios como los investigadores evalúan los riesgos de

privacidad de las aplicaciones examinando sus permisos solicitados, sin embargo, esto presenta una imagen incompleta porque solo indica a qué datos puede acceder una aplicación, y no dice nada sobre con con quién puede compartirlo y bajo qué circunstancias. La única forma de contestar estas preguntas es mediante la inspección del tráfico de red de las aplicaciones. Sin embargo, identificar la información personal dentro de las transmisiones de red requiere un esfuerzo considerable porque las aplicaciones y los SDK de terceros incorporados a menudo usan diferentes codificaciones y técnicas de ofuscación para transmitir datos. Por lo tanto, es un desafío técnico significativo poder descifrar todo el tráfico de red y buscar la información personal. Esta subsección discute cómo abordamos estos desafíos en detalle.

Información Personal Definimos ïnformación personalçomo cualquier dato. que potencialmente podría identificar a un individuo específico y distinguirlo de otro. Empresas de Internet, como los desarrolladores de aplicaciones móviles y redes de anuncios de terceros, desean este tipo de información para rastrear a los usuarios a través de dispositivos, sitios web y aplicaciones, ya que esto les permite obtener más información sobre consumidores individuales y generar más ingresos a través de publicidad dirigida. Por esta razón, estamos interesados principalmente en examinar el acceso de las aplicaciones a los identificadores persistentes que permiten el seguimiento a largo plazo, así como la información de ubicación.

Centramos nuestro estudio en detectar aplicaciones que utilizan canales encubiertos y laterales para acceder a tipos específicos de datos altamente sensibles, incluyendo los identificadores persistentes e información de geolocalización. En particular, la colección no autorizada de geolocalización en Android ha sido objeto de una acción reguladora previa [81]. La tabla 1 muestra los diferentes tipos de información personal que buscamos en las transmisiones de red , para qué se puede usar cada una, el permiso de Android que lo protege y la subsección en este documento donde discutimos los hallazgos sobre los canales lateral y encubiertos usados para acceder a ese tipo de datos.

Decodificar material ofuscado En nuestro trabajo anterior [**?**], encontramos casos de aplicaciones y bibliotecas de terceros (SDK) que utilizan técnicas de ofuscación para transmitir información personal a través de la red con diversos grados de sofisticación. Para identificar e informar de tales casos, automatizamos la decodificación de un conjunto estándar de codificaciones HTTP para identificar información personal codificada en los flujos de red, como gzip, base64 y hexadecimal codificado en ASCII. Además, nosotros buscamos información personal directamente, así como los hashes MD5, SHA1 y SHA256 de esta.

Después de analizar miles de trazas de red, todavía encontramos nuevas técnicas usadas por los SDK y las aplicaciones para ofuscar y cifrar las transmisiones de red. Mientras que reconocemos su esfuerzo por proteger los datos de los usuarios, las mismas técnicas podrían ser usadas para ocultar prácticas engañosas. En tales casos, usamos una combinación de ingeniería inversa y análisis estático para comprender la técnica precisa. Nosotros encontramos frecuentemente un uso adicional del cifrado AES aplicado a la os datos antes de enviarlos a través de la red, a menudo con claves AES

Cuadro 1: Tipos de información personal que buscamos, los permisos que protegen el acceso a ellos y el propósito por el cual son recogidos generalmente. También informamos de la subsección donde reportamos cada canal lateral y encubierto para acceder a cada tipo de datos, cuando los encontramos, y el número de aplicaciones que utilizan cada uno. La columna dinámica muestra la cantidad de aplicaciones en las que directamente observamos el acceso inapropiado a información personal, mientras que la columna estática muestra la cantidad de aplicaciones que contienen código que explota la vulnerabilidad (aunque no observamos que se ejecutara durante las pruebas)

Data Type	Permission	Purpose/Use	Subsection	Nº of Apps		Nº of SDKs		Chann
				Dynamic	Static	Dynamic	Static	Covert
IMEI	READ_PHONE_STATE	Persistent ID	4.1	13	159	2	2	2
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.2	42	12,408	1	1	0
Email	GET_ACCOUNTS	Persistent ID	Not Found					
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found					
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found					
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3	5	355	2	10	0
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found					
GPS	ACCESS_FINE_LOCATION	Location Data	4.4	1	1	0	0	0

codificadas.

Algunas bibliotecas siguieron las mejores prácticas al generar una sesión aleatoria de claves AES para cifrar los datos y luego cifrar la clave de sesión con una clave pública RSA permanente, que envía los datos cifrados y la clave de sesión cifrada juntos. Para descifrar sus transmisiones de red, instrumentamos las bibliotecas Java relevantes. Encontramos dos ejemplos de SDK de terceros .^{en}criptando"sus datos haciendo XOR con una palabra clave en un cifrado de estilo Vigin'ere. En un caso, esto fue aañadido a que ambos usaron encriptación estándar para los datos y TLS en la transmisión. Otros enfoques interesantes incluyeron revertir la cadena después de codificarla en base64 (a la que nos referimos como "46esab"), usando base64 varias veces (basebase6464), y usando una versión de alfabeto permutado de base64 (sa4b6e). Cada nuevo descubrimiento se agrega a nuestro conjunto de decodificaciones y todo nuestro conjunto de datos se vuelve a analizar.

3.4. Encontrar canales laterales y encubiertos

Una vez que tengamos ejemplos de transmisiones que sugieran que el sistema de permisos fue evitado (es decir, datos transmitidos por una aplicación a la que no se le habían otorgado los permisos necesarios para hacerlo), luego realizamos ingeniería inversa de la aplicación para determinar cómo eludió el sistema de permisos. Finalmente, usamos análisis estático para medir cuán frecuente es esta práctica entre el resto de nuestro conjunto de apps.

Ingeniería inversa Después de encontrar un conjunto de aplicaciones que exhiben compor-

tamientos coherentes con la existencia de canales laterales y encubiertos, analizamos manualmente cada caso. Si bien el proceso de ingeniería inversa lleva mucho tiempo y no es fácil de automatizar, es necesario determinar cómo la aplicación realmente obtuvo información fuera del sistema de permisos. Porque muchas de las transmisiones son causadas por el mismo código SDK, solo necesitamos hacer ingeniería inversa para cada técnica de elusión única: no es necesario analizar todas las aplicaciones, sino una cantidad de SDK mucho más pequeña. El punto final de destino para el tipo de tráfico de red identifica al responsable del SDK.

Durante el proceso de ingeniería inversa, nuestro primer paso fue usar apktool [7] para decompilar y extraer el código de bytes smali para cada aplicación sospechosa. Esto permitió analizar e identificar dónde se crearon las cadenas que contienen PII y de qué fuentes de datos. Para algunas aplicaciones y bibliotecas particulares, nuestro trabajo también necesitaba código de ingeniería inversa C ++; utilizamos IdaPro[1] para este propósito.

El proceso típico era buscar en el código las cadenas correspondientes a filtraciones vistas en las transmisiones de red y otros aspectos de los paquetes. Esta reveló dónde los datos estaban en la memoria, y luego el análisis estático del código revela de donde proviene ese valor. Como el código intencionalmente ofuscado es más complicado para la ingeniería inversa, también agregamos los registros generados durante el tiempo de ejecución y cadenas de errores como código nuevo en la aplicación descompilada, la volvemos a compilar y la ejecutamos dinámicamente para tener una idea de cómo funcionaba.

Medición de prevalencia El paso final de nuestro proceso fue determinar la prevalencia del canal lateral o canal encubierto en la práctica. Utilizamos nuestro análisis de ingeniería inversa para crear una huella digital única que identifique la presencia de un exploit en un SDK integrado. Este es robusto contra los falsos positivos. Por ejemplo, una huella digital es una constante de cadena que corresponde a un valor fijo clave de cifrado utilizada por un SDK, o el mensaje de error específico producido por otro SDK si la operación falla.

Luego descompilamos todas las aplicaciones en nuestro corpus y buscamos la cadena en los archivos resultantes. Dentro del código de bytes, buscamos la cadena en su totalidad con una instrucción const-string. Para bibliotecas de objetos compartidos como Unity, utilizamos el comando de cadenas para generar sus cadenas imprimibles. Incluimos la ruta y nombre del archivo como criterios coincidentes para protegernos contra falsos positivos. El resultado es un conjunto de todas las aplicaciones que pueden explotar el canal lateral o encubierto en la práctica pero para que no encontramos durante nuestro análisis dinámico, por ejemplo, debido a que a la aplicación se le había otorgado el permiso requerido, el Monkey no exploró esa rama de código en particular, etc.

4. Resultados

En esta sección, presentamos nuestros resultados agrupados por el tipo de permiso que la aplicación debe pedir para acceder a los datos; primero hablamos de canales encubiertos y laterales que permiten el acceso a identificadores de usuario o dispositivo persistentes (particularmente el IMEI y la dirección MAC del dispositi-

vo) y concluimos con los canales utilizados para acceder a la geolocalización de los usuarios (p. ej., a través de la infraestructura de red o metadatos presentes en contenido multimedia).

Nuestro entorno de prueba nos permitió identificar cinco tipos diferentes de canales laterales y canales encubiertos en uso entre las 88,113 diferentes aplicaciones de Android en nuestro conjunto de datos. La tabla 1 resume nuestros hallazgos e informa del número de aplicaciones y SDK de terceros que encontramos explotando estas vulnerabilidades en nuestro análisis dinámico y aquellos en los que nuestro análisis estático revela código que puede explotar estos canales. Nótese que esta última categoría, aquellas que pueden explotar estos canales, no descubrimos evidencias como lo hacían a través de nuestra instrumentación; Esto puede deberse a que el Automator Monkey no active el código en el que se usa dicho canal o porque la aplicación tenía el permiso requerido y, por lo tanto, la transmisión no se consideró sospechosa.

4.1. IMEI

La identidad internacional del equipo móvil (IMEI por sus siglas en inglés) es un valor numérico que identifica los teléfonos móviles de manera única. El IMEI tiene muchos usos válidos y legítimos. para identificar dispositivos en una red 3GPP, incluyendo la detección y bloqueo de teléfonos robados. El IMEI también es útil para los servicios de Internet como un identificador de dispositivo persistente usado para rastrear teléfonos individuales. El IMEI es un identificador poderoso ya que requiere esfuerzos extraordinarios para cambiar su valor o incluso falsificarlo. En algunas jurisdicciones, es ilegal cambiar el IMEI [56]. La colección del IMEI por terceros facilita el seguimiento en casos en los que el propietario intenta proteger su privacidad mediante el restablecimiento de otros identificadores, como el ID de publicidad.

Android protege el acceso al IMEI del teléfono con el permisos para leer el estado del teléfono. Identificamos dos servicios de Internet de terceros que utilizan diferentes canales encubiertos para acceder al IMEI cuando la aplicación no tiene el permiso requerido para acceder al IMEI.

Salmonads y almacenamiento externo Salmonads es un ”plataforma auxiliar para desarrolladores en la Gran China”que ofrece análisis y servicios de monetización para desarrolladores de aplicaciones [66]. Identificamos flujos de red a salmonads.com provenientes de cinco aplicaciones móviles que contenían el IMEI del dispositivo, a pesar del hecho que las aplicaciones no tenían permiso para acceder a ellas.

Estudiamos una de estas aplicaciones y confirmamos que contenía Salmonads SDK, y luego estudiamos el funcionamiento del SDK más a fondo. Nuestro análisis reveló que el SDK explota canales encubiertos para leer esta información del siguiente archivo oculto en la tarjeta SD: /sdcard/.googlex9/.xamdecoq0962. Si no está presente, este archivo es creado por el SDK de Salmonads. Entonces, cada vez que el usuario instala otra aplicación con el SDK de Salmonads integrado y con acceso legítimo al IMEI, el SDK -a través de la aplicación- lee y almacena el IMEI en este archivo.

El canal secreto es el acceso compartido de las aplicaciones a la tarjeta SD. Una vez que el archivo está escrito, todas las demás aplicaciones con el mismo SDK simplemente pueden leer el archivo a través de la API de Android,

que está regulada por el sistema de permisos. Más allá del IMEI, Salmonads también almacena la identificación publicitaria, un ID reiniciable para fines publicitarios y analíticos que permite optar al usuario por no participar en la publicidad y personalización basada en intereses, y la dirección MAC del teléfono, que está protegido con el permiso acceso al estado de la red. Nosotros modificamos el archivo para almacenar nuevos valores aleatorios y observamos que el SDK fielmente los envió a los dominios de Salmonads. La colección de la identificación publicitaria junto con otros identificadores y datos persistentes no reiniciables, como el IMEI, rompe la función de preservación de la privacidad de la ID de publicidad, que es la posibilidad de restablecerlo. También puede ser una violación de los Términos de servicio de Google [36],

Nuestra instrumentación nos permitió observar cinco aplicaciones diferentes que envían el IMEI sin permiso a Salmonads usando esta técnica. El análisis estático de todo nuestro corpus de aplicaciones reveló que seis aplicaciones contenían .xamdecoq0962, el nombre de archivo, codificado en el SDK como una cadena. La sexta aplicación había sido otorgada el permiso requerido para acceder al IMEI, razón por la cual no la identificamos inicialmente, así que puede ser la aplicación responsable de haber escrito inicialmente el IMEI al archivo. Tres de las aplicaciones fueron desarrolladas por la misma compañía, de acuerdo con los metadatos de Google Play, mientras que uno de ellos ha sido eliminado de Google Play. El límite inferior en la cantidad de veces que estas aplicaciones fueron instaladas es de 17,6 millones, según los metadatos de Google Play.

Baidu and External Storage Baidu es una gran corporación china cuyos servicios incluyen, entre muchos otros, un motor de búsqueda en línea, publicidad, servicios de mapas [14] y una API de geocodificación [13]. Observamos que los flujos de red la aplicación del parque Disneyland Hong Kong (com.disney.hongkongdisneyland_goo) envían el IMEI del dispositivo a los dominios Baidu. Esta aplicación ayuda a los turistas a navegar por el parque temático, y la aplicación hace uso del SDK de mapas de Baidu. Mientras Baidu Maps inicialmente solo ofrecía mapas de China continental, Hong Kong, Macao y Taiwán, a partir de 2019, ofrece servicios globales.

El SDK de Baidu usa la misma técnica que Salmonads para eludir el sistema de permisos de Android y acceder al IMEI sin permiso. Es decir, utiliza un archivo compartido en la tarjeta SD, por lo que una aplicación que contiene Baidu con el permiso correcto puede almacenarlo para otras aplicaciones que contienen Baidu y no tienen ese permiso. Específicamente, Baidu usa el siguiente archivo para almacenar y compartir estos datos: `/sdcard/backups/.SystemConfig/.cuid2`. El archivo es una cadena codificada en base64 que, cuando se decodifica, es un objeto JSON cifrado con AES que contiene el IMEI así como el hash MD5 de la concatenación de çom.baiduƶ el ID de Android.

Baidu usa AES en modo CBC con una clave estática y el mismo valor estático para el vector de inicialización (IV). Estos valores son, en hexadecimal, `33303231323130326469637564696162`. La razón por la cual este valor no es superficialmente representativo de un hex aleatorio se debe a que la clave de Baidu se calcula a partir de la representación binaria de la cadena AS-

CII `30212102dicudiab` —observe que cuando se invierte, se lee como `baidu cid 2012 12 03`. Al igual que con Salmonads, confirmamos que se puede cambiar el contenido (encriptado) de este archivo y los identificadores resultantes son enviado fielmente a los servidores de Baidu.

Observamos ocho aplicaciones enviando el IMEI del dispositivo a Baidu sin los permisos necesarios, pero encontramos 153 aplicaciones diferentes en nuestro repositorio que han codificado la constante correspondiente a la clave de cifrado. Esto incluye dos de Disney: una aplicación para los parques temáticos de Hong Kong y Shangai (com.disney.shanghaidisneyland_goo). De esos 153, las dos aplicaciones más populares fueron Samsung's Health (com.sec.android.app.shealth) y el navegador de Samsung (com.sec.android.app.sbrowser), ambas con más de 500 millones de instalaciones. Hay un límite inferior de 2.600 millones de instalaciones para las aplicaciones identificadas como que contienen el SDK de Baidu. De estas 153 aplicaciones, todas menos 20 tienen el permiso de lectura del estado del teléfono. Esto significa que tienen legítimo acceso al IMEI y pueden ser las aplicaciones que realmente crean el archivo que almacena estos datos. Los 20 que no tienen el permiso solo pueden obtener el IMEI a través de este canal secreto. Estas 20 aplicaciones tienen un límite inferior total de 700 millones de instalaciones

4.2. Direcciones de red MAC

La dirección de control de acceso a medios (dirección MAC) es un identificador de 6 bytes que se asigna de forma exclusiva al controlador de interfaz de red (NIC) para establecer comunicaciones de capa de enlace. Sin embargo, la dirección MAC también es útil para anunciantes y compañías de análisis como un identificador persistente basado en hardware, similar al IMEI.

Android protege el acceso a la dirección MAC del dispositivo con el permiso de acceso al estado de la red. A pesar de esto, observamos aplicaciones que transmiten la dirección MAC del dispositivo sin tener permiso para acceder. Las aplicaciones y los SDK obtienen acceso a esta información utilizando el código nativo de C ++ para invocar una serie de Sistema de llamadas UNIX.

Unity y IOCTLs Unity es un motor de juegos multiplataforma desarrollado por Unity Technologies y muy utilizado por los juegos móviles de Android [76]. Nuestro análisis de tráfico identificó varios juegos basados en Unity que envían el hash MD5 de la dirección MAC a los servidores de Unity y se referiren a él como un uuid en la transmisión (por ejemplo, como un nombre de clave de parámetro HTTP GET). En este caso, el acceso estaba sucediendo dentro de la biblioteca nativa C ++ de Unity. Realizamos ingeniería inversa de libunity.so para determinar cómo estaba obteniendo la dirección MAC.

Invertir la biblioteca de C ++ compilada de MiB de Unity 18 tiene más que ver con el código de Android. Sin embargo, pudimos aislar dónde se estaban utilizando los datos procesados precisamente porque codifica la dirección MAC con MD5. Unity utilizó su propia implementación MD5 sin etiquetar que encontramos al buscar los números constantes asociados con MD5; en este caso, las constantes del estado inicial.

Unity abre un zócalo de red y utiliza un `ioctl`

(UNIX “input-output control”) para obtener la dirección MAC de la interfaz de red WiFi. En efecto, `ioctl`s crean un gran numero de llamadas API ”numeradas”que técnicamente no son diferentes de las llamadas de sistema bien nombradas como “bind” o “close” pero se usan con poca frecuencia características utilizadas El comportamiento de un ioctl depende de la ”solicitud.[es]pecífica. Específicamente, Unity usa el siocgifconf [2] ioctl para obtener la red de interfaces, y luego usa el siocgifhwaddr [3] ioctl para obtener la dirección MAC correspondiente.

Observamos que 42 aplicaciones obtenían y enviaban a servidores de Unity la Dirección MAC de la tarjeta de red sin tener el permiso *access_network_state*. Para cuantificar la prevalencia de esta técnica en nuestro corpus de aplicaciones de Android, identificamos este comportamiento a través de una cadena de error que hace referencia al código ioctl que acaba de fallar. Esto nos permitió encontrar un total de 12,408 aplicaciones que contienen esta cadena de error, de las cuales 748 aplicaciones no contienen el permiso *access_network_state*.

4.3. Dirección MAC del enrutador

El acceso a la dirección MAC del enrutador WiFi (BSSID) está protegido por el permiso *access_wifi_state*. En la Sección 2, ejemplificamos canales laterales con direcciones MAC del enrutador que son utilizados como datos de ubicación encubiertos, y ejemplos del FTC promulgando millones de dólares en multas para aquellos involucrados en la práctica de usar estos datos para inferir engañosamente las ubicaciones de los usuarios. Android Nougat agregó un requisito de que las aplicaciones tengan un permiso de ubicación adicional para escanear para redes WiFi cercanas [34]; Android Oreo requirió además un permiso de ubicación para obtener el SSID y la dirección MAC de la red WiFi conectada. Además, conocer la dirección MAC de un enrutador permite vincular diferentes dispositivos que comparten acceso a Internet, pudiendo revelar relaciones personales por parte de sus propietarios, o habilitar el seguimiento entre dispositivos.

Nuestro análisis reveló dos canales laterales para acceder a la información del enrutador WiFi: leer el caché ARP y preguntar directamente al enrutador. No encontramos canales laterales que permitieron escanear otras redes WiFi. Tenga en cuenta que este problema afecta a todas las aplicaciones que se ejecutan en versiones recientes de Android, no solo a aquellas sin el permiso de acceso al estado wifi. Esto se debe a que afecta a las aplicaciones sin un permiso de ubicación, y afecta a las aplicaciones con un permiso de ubicación que el usuario no ha aceptado cuando fue preguntado por primera vez.

Lectura de la tabla ARP El Protocolo de resolución de direcciones (ARP) es una protocolo de trabajo que permite descubrir y mapear la dirección de la capa MAC como asociado a una dirección IP dada. Para mejorar el rendimiento de la red, el ARP El protocolo utiliza un caché que contiene una lista histórica de entradas ARP, es decir, un histórico de direcciones IP resueltas en la dirección MAC, incluida la dirección IP y la dirección MAC del enrutador inalámbrico al que está conectado el dispositivo (es decir, su BSSID).

La lectura de la caché ARP se realiza abrien-

[2] Configuración de la interfaz IOCT get

[3] Dirección de hardware de la interfaz IOCT get

Cuadro 2: SDK que se ven enviando direcciones MAC del enrutador y que también contienen código para acceder a la caché ARP. Como referencia, informamos la cantidad de aplicaciones y una menor límite del número total de instalaciones de esas aplicaciones. Hacemos esto para todas las aplicaciones. que contiene el SDK; aquellas aplicaciones que no tienen acceso al estado WiFi, que significa que el canal lateral elude el sistema de permisos; y esos aplicaciones que tienen un permiso de ubicación, lo que significa que el canal lateral evita la revocación de la ubicación.

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

do el pseudo archivo */proc/net/arp* y procesando su contenido. Este archivo no está protegido por ningún mecanismo de seguridad, por lo que cualquier aplicación puede acceder y analizarlo para obtener acceso a geolocalización basada en la información del enrutador sin tener un permiso de ubicación. Construimos una aplicación de prueba y la probamos para Android Pie usando una aplicación que no requiere permisos. También demostramos que cuando se ejecuta una aplicación que solicita tanto el permiso de acceder el estado de wifi como a la ubicación aproximada y se deniegan esos permisos, la aplicación accederá a los datos de todos modos. Divulgamos de manera responsable nuestros hallazgos a Google en septiembre de 2018

Descubrimos esta técnica durante el análisis dinámico, cuando observamos una biblioteca que usa este método en la práctica: OpenX [57], una compañía que dice en su sitio web que çrea mercados programáticos en los que las publicaciones premium y los desarrolladores de aplicaciones pueden rentabilizar mejor su contenido conectándose con anunciantes líderes que valoran su público ". El código SDK de OpenX no estaba ofuscado y así observamos que habían nombrado la función responsable `getDeviceMacAddressFromArp`. Además, un análisis detallado del código demuestra que primero trata de obtener los datos legítimamente usando el permiso ; esta vulnerabilidad solo se usa después de que la aplicación ha sido denegada explícitamente el acceso a estos datos.

OpenX no envia directamente la dirección MAC, sino el hash MD5. Sin embargo, es trivial calcular una dirección MAC a partir de su hash: debido al pequeño número de direcciones MAC estas son vulnerables a un ataque de fuerza bruta sobre funciones hash (es decir, un límite superior de 48 bits de entropía) [4] Además, en

[4] Usando hardware básico, se puede calcular el MD5 para cada dirección MAC posible en cuestión de minu-

la medida en que la dirección MAC del enrutador se utiliza para resolver la geolocalización de un usuario a través de mapeo dirección MAC a ubicación, uno necesita solo necesita el hash de las direcciones MAC en esta asignación (o almacenar los hash en el tabla) y comprobar si coincide con el valor recibido.

Si bien OpenX fue el único SDK que observamos explotando este canal lateral , buscamos en todo el corpus de la aplicación la cadena `/proc/net/arp` , y encontramos múltiples bibliotecas de terceros que lo incluyeron. En el caso de uno de ellos, Igexin, hay informes existentes de su comportamiento depredador [15]. En nuestro caso, los archivos de registro indicaron que después de que se denegara el permiso a igexin para buscar WiFi, leyó `/system/xbin/ip`, ejecutó `/system/bin/ifconfig` y luego ejecutó `cat /proc/net/ar`. La tabla 2 muestra la prevalencia de bibliotecas de terceros con código para acceder a la caché ARP.

Enrutador UPnP Una SDK en la Tabla 2 incluye otra técnica para obtener la dirección MAC del punto de acceso WiFi: utiliza protocolos de descubrimiento UPnP / SSDP. Tres de las aplicaciones de controles remotos inteligentes de Peel (tv.peel.samsung.app, tv.peel.smartremote y tv.peel.mobile.app) se conectan a 192.168.0.1, la dirección IP del enrutador ya que es su puerta de enlace a Internet. El enrutador en esta configuración era un enrutador doméstico básico que admite plug-and-play universal; la aplicación solicitó el igd.xml (archivo de configuración del dispositivo de puerta de enlace) a través del puerto 1900 en el enrutador. El enrutador respondió con, entre otros detalles de fabricación, su dirección MAC como parte de su UUID. Estas aplicaciones también enviaron direcciones MAC WiFi a sus propios servidores y un dominio alojado por Amazon Web Services.

El hecho de que el enrutador proporcione esta información a los dispositivos alojados en la red doméstica no es un defecto de Android per se. Más bien es una consecuencia de considerar que todas las aplicaciones en cada teléfono conectado a una red WiFi están encendidas en el lado seguro del firewall.

4.4. Geolocalización

Identificamos 70 aplicaciones diferentes que envían datos de ubicación a 45 dominios diferentes sin tener ninguno de los permisos de ubicación. La mayoría de estas transmisiones de ubicación no fueron causadas por elusión del sistema de permisos, sino que los datos de ubicación se proporcionaron dentro de los paquetes entrantes: los servicios mediadores de anuncios de la región proporcionaron los datos de ubicación integrados en el enlace del anuncio. Cuando nosotros volvimos a probar las aplicaciones en una ubicación diferente, sin embargo, la ubicación devuelta no es más precisa, por lo que sospechamos que estos mediadores publicitarios estaban utilizando geolocalizacióna través de la IP, aunque con un grado de precisión mucho mayor de lo normal. Una aplicación utilizó explícitamente la geolocalización basada en IP de www.googleapis.com y encontramos que la ubicación devuelta era precisa a unos pocos metros; Una vez más, sin embargo, esta precisión no se repitió cuando volvimos a probar en otro lugar [59]. Sin embargo, descubrimos un canal lateral genuino a través de datos EXIF de fotos.

tos [40].

Shutterfly y Metadatos EXIF Observamos que la aplicación Shutterfly (com.shutterfly) envía datos precisos de geolocalización a su propio servidor (apcmobile.thislife.com) sin tener un permiso de ubicación. En cambio, envió metadatos de fotos desde la biblioteca de fotos, incluyenfo la ubicación precisa del teléfono a través del Intercambio de Datos de Formato de Archivo de Imagen (EXIF por sus siglas en inglés). La aplicación realmente procesó el archivo de imagen: agregó los metadatos EXIF, incluida la ubicación, en un objeto JSON con los campos de latitud y longitud y lo transmitió a su servidor.

Si bien esta aplicación puede no tener la intención de eludir el sistema de permisos, esta técnica puede ser explotada por un actor malicioso para obtener acceso a la ubicación del usuario. Cada vez que el usuario toma una nueva imagen con la geolocalización habilitada, cualquier aplicación con acceso de lectura a la biblioteca de fotos (es decir, leer almacenamiento externo) puede conocer la ubicación precisa del usuario cuando se tomó dicha foto. Además, también permite obtener un historial de geolocalización con marcas de tiempo del usuario, que luego podría usarse para inferir información confidencial sobre ese usuario.

5. Trabajo relacionado

Nos basamos en una vasta literatura en el campo de los ataques de canales laterales y encubiertos para Android. Sin embargo, si bien los estudios anteriores generalmente solo informaron casos aislados de tales ataques o abordaron el problema desde un ángulo teórico, nuestro trabajo combina análisis estático y dinámico para detectar automáticamente casos de mal comportamiento y ataques en el mundo real.

Canales encubiertos Marforio *et al.* [49] propuso varios escenarios para transmitir datos entre dos aplicaciones de Android, incluido el uso de sockets UNIX y almacenamiento externo como un búfer compartido. En nuestro trabajo vemos que el almacenamiento compartido es de hecho utilizado en la naturaleza. Otros estudios se han centrado en el uso de ruidos móviles [26, 69] y vibraciones generadas por el teléfono (que podrían ser inaudibles para los usuarios) como canales encubiertos [3, 17]. Tales ataques generalmente involucran dos dispositivos físicos comunicandose entre ellos. Esto está fuera del alcance de nuestro trabajo, ya que nos centramos en las vulnerabilidades del dispositivo que están siendo explotadas por aplicaciones y terceras partes en el espacio del usuario.

Canales laterales Spreitzer *et al.* proporcionó una buena clasificación de canales laterales específicos para dispositivos móviles presentes en la literatura [73]. El trabajo anterior ha demostrado que los recursos de Android no privilegiados podrían usarse para inferir datos personales sobre los usuario, incluidos los identificadores únicos [71] o el género [51]. Los investigadores también demostraron que es posible identificar la ubicación de los usuarios. controlando el consumo de energía de sus teléfonos [52] y detectando recursos de Android disponibles públicamente [90]. Más recientemente, Zhang *et al.* demostró un ataque de huellas dactilares de calibración del sensor que utiliza datos no protegidos de calibración recopilados de sensores como el acelerómetro, el giroscopio y el magnetómetro [89]. Otros han demostrado que la información desprotegida de todo

el sistema es suficiente para inferir el texto de entrada en teclados basados en gestos [71]. Otros trabajos de investigación también han informado de técnicas que aprovechan información de red poco protegida ción para geolocalizar usuarios a nivel de red [54, 81, 2]. Extendemos el trabajo previo informando de bibliotecas y aplicaciones móviles de terceros que obtienen acceso a identificadores únicos e información de ubicación en la naturaleza mediante la explotación lateral y canales encubiertos.

6. Discusión

Nuestro trabajo muestra una serie de canales secundarios y encubiertos que están siendo utilizados por aplicaciones para eludir el sistema de permisos de Android. El número de usuarios potencialmente afectados por estos hallazgos están en los cientos de millones. En esta sección, discutimos cómo es probable que estos problemas desafíen las expectativas razonables de los usuarios, y cómo estos comportamientos pueden constituir violaciones de varias leyes.

Observamos que estos exploits pueden no ser necesariamente maliciosos e intencionales. La aplicación Shutterfly que extrae información de geolocalización de metadatos EXIF puede no estar haciendo esto para conocer la información de ubicación sobre el usuario o no estar utilizando estos datos más tarde para cualquier propósito. Por otro lado, casos donde un la aplicación contiene ambos códigos para acceder a los datos a través del sistema de permisos y el código que implementa una evasión no admite fácilmente una explicación inocente. Aún menos para aquellos que contienen código para acceder legítimamente a los datos y luego guárdarlos para que otros puedan acceder. Esto es particularmente malo porque cualquier aplicación que conozca el canal encubiertos puede explotarlo, no solo las que comparten el mismo SDK. El hecho de que Baidu escriba el IMEI del usuario en un almacenamiento de acceso público permite que cualquier aplicación acceda a ella sin permiso, no solo otras aplicaciones que contienen Baidu.

6.1. Expectativas de privacidad

En los EE. UU., Las prácticas de privacidad se rigen por el marco de “notificación y consentimiento”: las empresas pueden avisar a los consumidores sobre sus prácticas de privacidad (a menudo en forma de una política de privacidad), y los consumidores pueden consentir estas prácticas mediante el uso de los servicios de la empresa. Si bien las políticas de privacidad del sitio web son ejemplos canónicos de este marco en acción, el sistema de permisos en Android (o en cualquier otra plataforma) es otro ejemplo del marco de notificación y consentimiento , porque cumple dos propósitos: (i) proporcionar transparencia sobre los recursos confidenciales a los que las aplicaciones solicitan acceso (aviso) y (ii) que requieren consentimiento explícito del usuario antes de que una aplicación pueda acceder, recopilar y compartir recursos confidenciales y datos (consentimiento). Que las aplicaciones pueden eludir el aviso y el consentimiento es una prueba más del fracaso de este marco. En términos prácticos, sin embargo, estos comportamientos de la aplicación pueden conducir directamente a violaciones de privacidad porque es probable que desafíen las expectativas de los consumidores.

El marco de ”Privacidad como integridad contextual”de Nissenbaum define la privacidad vio-

laciones como flujos de datos que desafían las normas de información contextual [55]. En el marco de Nissenbaum, los flujos de datos son modelados por remitentes, destinatarios, sujetos de datos, tipos de datos y principios de transmisión en contextos específicos (por ejemplo, proporcionar funcionalidad de la aplicación, publicidad, etc.). Al eludir el sistema de permisos, las aplicaciones pueden filtrar datos a sus propios servidores e incluso a terceros en formas que probablemente desafíen las expectativas de los usuarios (y las normas sociales), en particular principalmente si ocurre después de haber denegado la solicitud de permiso explícito de una aplicación. Es decir, independientemente del contexto, si se le preguntara explícitamente a un usuario sobre la concesión de acceso a la información personal para una aplicación y luego se denegara explícitamente, sería razonable esperar que los datos no sean accesibles para la aplicación. Por lo tanto, los comportamientos que documentamos en este documento constituyen una violación de privacidad clara. Desde una perspectiva legal y política, es probable que estas prácticas sean consideradas engañosas o ilegales.

Tanto una reciente decisión CNIL (autoridad de protección de datos de Francia), con respecto a los requisitos de notificación y consentimiento de GDPR, y varios casos de la FTC, con respecto a las prácticas injustas y engañosas según la ley federal de los EE. UU. en la siguiente sección: enfatizamos la función de aviso de los permisos de Android desde la perspectiva de las expectativas del consumidor. Además, estos problemas están también en el medio de una queja reciente presentada por el Abogado del condado de Los Ángeles (LACA) bajo la Ley de Competencia Desleal del Estado de California. La queja de LACA fue presentada contra una aplicación de clima popular relacionada con los jardines. El caso se centra además en la función de notificación del sistema de permisos, al tiempo que señala que ”los usuarios no tienen ninguna razón para buscar [recopilación de datos de geolocalización] en la aplicación revisando la extensa [política de privacidad] de la aplicación, enterrada en discusiones opacas de la transmisión potencial [del desarrollador] de datos de ubicación a terceros y su uso para fines comerciales adicionales. De hecho, sobre información y creencias, la gran mayoría de los usuarios no leen esos secciones en absoluto -[75].

6.2. Asuntos legales y de política

Las prácticas que destacamos en este documento también destacan varios aspectos legales y cuestiones de política de protección de datos. En los Estados Unidos, por ejemplo, pueden entrar en conflicto con las prohibiciones de la FTC contra prácticas engañosas y / o leyes estatales que rigen prácticas comerciales justas. En la Unión Europea, pueden constituir violaciones del Reglamento General de Protección de Datos (GDPR)

La Comisión Federal de Comercio (FTC), que se encarga de proteger los intereses del consumidor, ha presentado una serie de casos bajo la Sección 5 de la Federal Trade Commission (FTC) Act [78] en este contexto. Estas quejas han declarado que la elusión de los permisos de Android y la recopilación de la información en ausencia del consentimiento de los usuarios o de manera engañosa es un acto injusto y engañoso [83]. Un caso sugirió que las aplicaciones que solicitan permisos más allá de lo que los usuarios esperan o lo que se necesita para operar el servicio se consideraron ïrrazo-

nables"bajo la Ley FTC. En otro caso, la FTC persiguió una queja bajo la Sección 5 alegando que un fabricante de dispositivos móviles, HTC, permitió a los desarrolladores recopilar información sin obtener el permiso de los usuarios a través del sistema de permisos de Android, y no pudo proteger a los usuarios de posibles explotaciones por terceros de un fallo de seguridad relacionada [80]. Finalmente, la FTC tiene casos perseguidos que involucran declaraciones falsas de los consumidores con respecto a la exclusión voluntaria de mecanismos de publicidad a medida en aplicaciones móviles en general [82].

También en los Estados Unidos, Los estatutos de la leyes y prácticas desleales y engañosas (UDAP) también pueden aplicarse a nivel estatal. Estos típicamente reflejan y complementan la ley federal correspondiente. Finalmente, con el creciente público regulador y atención pública a los problemas relacionados con la privacidad y seguridad de los datos, la recopilación de datos que socava las expectativas de los usuarios y su consentimiento informado también puede estar violando varias normas generales de privacidad, como la publicación en línea de Children's Privacy Protection Act (COPPA) [79], la reciente California Privacy Protection Act (CCPA), y potencialmente leyes de notificación de violación de datos que se centran en ls recopilación no autorizada, según el tipo de información personal recopilada.

En Europa, estas prácticas pueden violar el GDPR. En una descisión recinete , el regulador de datos francés, CNIL, impuso una multa de 50 millones de euros por una violación de los requisitos de transparencia de GDPR, subrayando la necesidad de el conocimiento informado como requisito para la recopilación de datos para anuncios personalizados [25]. Este fallo también sugiere que, en el contexto del consentimiento y la transparencia del GDPR disposiciones las solicitudes de permisos cumplen una función clave para informar a los usuarios de prácticas de recopilación de datos y como mecanismo para proporcionar el consentimiento informado [80].

Nuestro análisis saca a la luz métodos novedosos de elusión de permisos usados habitualmente por aplicaciones Android legítimas. Estas elusiones permiten la recopilación de información sin pedir consentimiento o después de que el usuario se haya negado explícitamente a dar su consentimiento, probablemente rompiendo las expectativas de los usuarios y potencialmente violando los requisitos de privacidad y protección de datos en un estado, a nivel federal e incluso global. Al descubrir estas prácticas y hacer nuestros datos públicos,[5] esperamos proporcionar suficientes datos y herramientas para que los reguladores realizen acciones, para que la industria puede identificar y solucionar problemas antes de lanzar aplicaciones, y para permitir a los consumidores tomar decisiones informadas sobre las aplicaciones que utilizan.

7. Limitaciones y trabajo futuro

Durante el curso de la realización de esta investigación, tomamos ciertas decisiones de diseño que pueden afectar a la exhaustividad y generalización de este trabajo. es decir, todos los hallazgos en este documento representan límites inferiores al número de canales encubiertos y laterales que pueden existir en la naturaleza.

Nuestro estudio considera un subconjunto de

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

los permisos etiquetados por Google como peligrosos aquellos que controlan el acceso a los identificadores de usuario y la información de geolocalización. Según la documentación de Android, estos son los que regulan el acceso a los datos más preocupante e intrusivos para la privacidad. Sin embargo, puede haber otros permisos que, aunque no estén etiquetadoS como peligroso, pueden dar acceso a datos confidenciales del usuario. Un ejemplo es el permiso de Bluetooth; permite que las aplicaciones descubran otros dispositivos con Bluetooth cercanos, algo que puede ser útiles para el seguimiento físico del perfil del consumidory el seguimiento entre dispositivos. Además, no examinamos todos los permisos peligrosos, específicamente los datos guardados por los proveedores de contenido, como como contactos de la libreta de direcciones y mensajes SMS.

Nuestros métodos se basan en observaciones de transmisiones de red que sugieren la existencia de tales canales, en lugar de buscarlos directamente a través de análisis estático. Debido a que muchas aplicaciones y bibliotecas de terceros usan técnicas de ofuscación para disfrazar sus transmisiones, puede haber transmisiones que nuestra instrumentación no marca que contenga información protegida con permiso. Además, puede haber canales explotados, pero durante nuestras pruebas las aplicaciones no transmitieron los datos personales accedidos. Además, las aplicaciones podrían estar exponiendo canales, pero nunca abusar de ellos durante nuestras pruebas. Aunque nosotros no informaría tal comportamiento, esto sigue siendo una violación inesperada de Android modelo de seguridad.

Muchas aplicaciones populares también usan la fijación de certificados [61, 28], lo que resulta en rechazar el certificado personalizado utilizado por nuestro proxy man-in-the-middle; nuestra sistema permite que estas aplicaciones continúen sin interferencias. La fijación de certificados es un comportamiento razonable desde un punto de vista de seguridad; es posible, sin embargo, que se está utilizando para frustrar los intentos de analizar y estudiar el tráfico de red de un teléfono móvil del usuario.

Nuestro análisis dinámico utiliza el Monkey de Android como un difusor de interfaz de usuario para generar eventos de IU aleatorios para interactuar con las aplicaciones. Mientras que en nuestro trabajo anterior descubrimos que el Monkey exploró ramas de código similares a las de un humano en un 60de las aplicaciones probadas [**?**],es probable que aún no explore algunas ramas de código que pueden explotar canales encubiertos y laterales. Por ejemplo, el Monkey no puede interactuar con aplicaciones que requieren que los usuarios interactúen con pantallas de inicio de sesión o, más generalmente, requieren entradas específicas para proceder. Tales aplicaciones no son consecuentemente tan exhaustivamente probadas como aplicaciones susceptibles de exploración automatizada. El trabajo futuro debería comparar nuestros enfoques con herramientas más sofisticadas para automatizar la exploración, como el Crashscope de Moran *et al.* [53], que genera entradas diseñadas para desencadenar eventos de bloqueo.

En última instancia, estas limitaciones solo dan como resultado la posibilidad de que haya y canales encubiertos que aún no hemos descubierto (es decir, falsos negativos). Eso no tiene ningún impacto en la validez de los canales que descubrimos (es decir, no son falsos positivos) y las mejoras en nuestra metodología solo pueden resultar en el descubrimiento de más de estos canales.

En el futuro, debe haber un esfuerzo colectivo proveniente de todas las partes interesadas para evitar que las aplicaciones eludan el sistema de permisos. Google, a su crédito, han anunciado que están abordando muchos de los problemas que les informamos [33].Sin embargo, estas correcciones solo estarán disponibles para los usuarios a partir de Android Q, protegiendo solo a aquellos usuarios con los medios para comprar un nuevo teléfono inteligente. Esto, por supuesto, posiciona la privacidad como un bien de lujo, algo que está en conflicto con los pronunciamientos públicos de Google [58]. En cambio, deberían tratar las vulnerabilidades de privacidad con la misma seriedad que tratan las vulnerabilidades de seguridad y emitir revisiones a todas las versiones de Android compatibles.

Los reguladores y los proveedores de plataformas necesitan mejores herramientas para monitorizar las aplicaciones. Comportar y responsabilizar a los desarrolladores de aplicaciones garantizando que las aplicaciones cumplan con leyes aplicables, a saber, protegiendo la privacidad de los usuarios y respetando la recopilación de datos opciones de lectura. La sociedad debería apoyar más mecanismos, técnicos y de otro tipo, que potencian la toma de decisiones informada de los usuarios con mayor transparencia en qué aplicaciones están haciendo en sus dispositivos. Para este fin, hemos hecho la lista de todas las aplicaciones que explotan o contienen código para explotar los canales secundarios y encubiertos que descubierto disponible en línea [8].

Expresiones de gratitud

Este trabajo fue apoyado por la Ciencia de la Agencia de Seguridad Nacional de EE. UU. Programa de seguridad (contrato H98230-18-D-0006), el Departamento de Patria Seguridad (contrato FA8750-18-2-0096), la National Science Foundation (becas CNS-1817248 y concesión CNS-1564329), la Fundación Rose, el europeo Programa de acción de innovación Horizonte 2020 de la Unión (Acuerdo de subvención no 786741, Proyecto SMOOTH), el Laboratorio de Transparencia de Datos y el Centro para Largo Plazo Ciberseguridad en UC Berkeley. Los autores desean agradecer a John Aycock, Irwin Reyes, Greg Hagen, René Mayrhofer, Giles Hogben y Refjohürs Lykkewe.

Referencias

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

[2] J. P. Achara, M. Cunche, V. Roca, and A. Francillon. WifiLeaks: Underestimated privacy implications of the access wifi state Android permission. Technical Report EURECOM+4302, Eurecom, 05 2014.

[3] A. Al-Haiqi, M. Ismail, and R. Nordin. A new sensors-based covert channel on android. *The Scientific World Journal*, 2014, 2014.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[5] Android Documentation. App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro, 2019. Accessed: February 12, 2019.

[6] Android Studio. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017. Accessed: October 12, 2017.

[7] Apktool. Apktool: A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

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

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of PLDI*, pages 259–269, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[11] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[12] G. S. Babil, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–8, July 2013.

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

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

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

[16] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving Apps to Test the Security of Third-Party Components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, San Diego, CA, 2014. USENIX Association.

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

[18] S. Cabuk, C. E. Brodley, and C. Shields. IP covert channel detection. *ACM Transactions on Information and System Security (TISSEC)*, 12(4):22, 2009.

[19] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proc. of NDSS*, 2015.

[20] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[21] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[22] M. Christodorescu, S. Jha, S. A Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[23] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2017.

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

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

[26] Luke Deshotels. Inaudible sound as a covert channel in mobile devices. In *USENIX WOOT*, 2014.

[27] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[29] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 414–421. IEEE, 2014.

[30] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, page 3, New York, NY, USA, 2012. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[32] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

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

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

[35] Google, Inc. Distribution dashboard. `https://developer.android.com/about/dashboards`, May 7 2019. Accessed: June 1, 2019.

[36] Google Play. Usage of Google Advertising ID. `https://play.google.com/about/monetization-ads/ads/ad-id/`, 2019. Accessed: February 12, 2019.

[37] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[38] C. Han, I. Reyes, A. Elazari Bar On, J. Reardon, Á. Feal, S. Egelman, and N. Vallina-Rodriguez. Do You Get What You Pay For? Comparing The Privacy Behaviors of Free vs. Paid Apps. In *Workshop on Technology and Consumer Protection*, ConPro '19, 2019.

[39] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

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

[41] J. Kim, Y. Yoon, K. Yi, J. Shin, and SWRD Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.

[42] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[43] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[44] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In *Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, pages 15–20, 2015.

[45] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads! Balancing Privacy in an Ad-Supported Mobile Application Market. In *Proc. of ACM HotMobile*, page 2, 2012.

[46] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *badgers*, pages 3–17, 2014.

[47] M. Liu, H. Wang, Y. Guo, and J. Hong. Identifying and Analyzing the Privacy of Apps for Kids. In *Proc. of ACM HotMobile*, 2016.

[48] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[49] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[50] E. McReynolds, S. Hubbard, T. Lau, A. Saraf, M. Cakmak, and F. Roesner. Toys That Listen: A Study of Parents, Children, and Internet-Connected Toys. In *Proc. of ACM CHI*, 2017.

[51] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security Symposium*, pages 1053–1067, 2014.

[52] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, pages 785–800, 2015.

[53] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.

[54] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang. Unlocin: Unauthorized location inference on smartphones without being caught. In *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, pages 1–8. IEEE, 2013.

[55] Helen Nissenbaum. Privacy as contextual integrity. *Washington Law Review*, 79:119, February 2004.

[56] United Kingdom of Great Britain and Northern Ireland. Mobile telephones (re-programming) act.

http://www.legislation.gov.uk/ukpga/2002/31/introduction, 2002.

[57] OpenX. Why we exist. https://www.openx.com/company/, 2019.

[58] Sundar Pichai. Privacy Should Not Be a Luxury Good. The New York Times, May 7 2019. https://www.nytimes.com/2019/05/07/opinion/google-sundar-pichai-privacy.html.

[59] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM Computer Communication Review*, 41(2):53–56, 2011.

[60] A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 65–66. ACM, 2012.

[61] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362. ACM, 2017.

[62] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[63] A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In Situ Mobile Traffic Analysis in User Space. *arXiv preprint arXiv:1510.01419*, 2015.

[64] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug fixes, improvements,... and privacy leaks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[65] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In *Proceedings of the ACM SIGMOBILE MobiSys*, pages 361–374, 2016.

[66] Salmonads. About us. http://publisher.salmonads.com, 2016.

[67] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[68] Sarah Schafer. With capital in panic, pizza deliveries soar. The Washington Post, December 19 1998. https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm.

[69] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang.

Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[70] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[71] L. Simon, W. Xu, and R. Anderson. Don't interrupt me while i type: Inferring text entered through gesture typing on android keyboards. *Proceedings on Privacy Enhancing Technologies*, 2016(3):136–154, 2016.

[72] Y. Song and U. Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *Proc. of ACM SPSM*, 2015.

[73] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2017.

[74] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. `https://www.statista.com/statistics/266136`, 2019. Accessed: February 11, 2019.

[75] COUNTY OF LOS ANGELES SUPERIOR COURT OF THE STATE OF CALIFORNIA. Complaint for injunctive relief and civil penalties for violations of the unfair competition law. `http://src.bna.com/EqH`, 2019.

[76] Unity Technologies. Unity 3d. `https://unity3d.com`, 2019.

[77] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.W. Chen. Turtle guard: Helping android users apply contextual privacy preferences. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 145–162. USENIX Association, 2017.

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

[79] U.S. Federal Trade Commission. Children's online privacy protection rule ("coppa"). `https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule`, November 1999.

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

[81] U.S. Federal Trade Commission. Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission. `https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network`

-inmobi-settles-ftc-charges-it-tracked, June 22 2016.

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

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

[84] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, Washington, D.C., August 2015. USENIX Association.

[85] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, May 2017.

[86] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[87] B. Yankson, F. Iqbal, and P.C.K. Hung. Privacy preservation framework for smart connected toys. In *Computing in Smart Toys*, pages 149–164. Springer, 2017.

[88] S. Yong, D. Lindskog, R. Ruhl, and P. Zavarsky. Risk Mitigation Strategies for Mobile Wi-Fi Robot Toys from Online Pedophiles. In *Proc. of IEEE SocialCom*, pages 1220–1223. IEEE, 2011.

[89] J. Zhang, A. R. Beresford, and I. Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[90] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of 2013 ACM SIGSAC conference on Computer & Communications Security*. ACM, 2013.

[91] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg. Automated analysis of privacy requirements for mobile apps. In *24th Network & Distributed System Security Symposium*, 2017.