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 [75], 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, 86].

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 [16, 62]. 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

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 [72]. 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 [90]. 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 [82].

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 respon-

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.

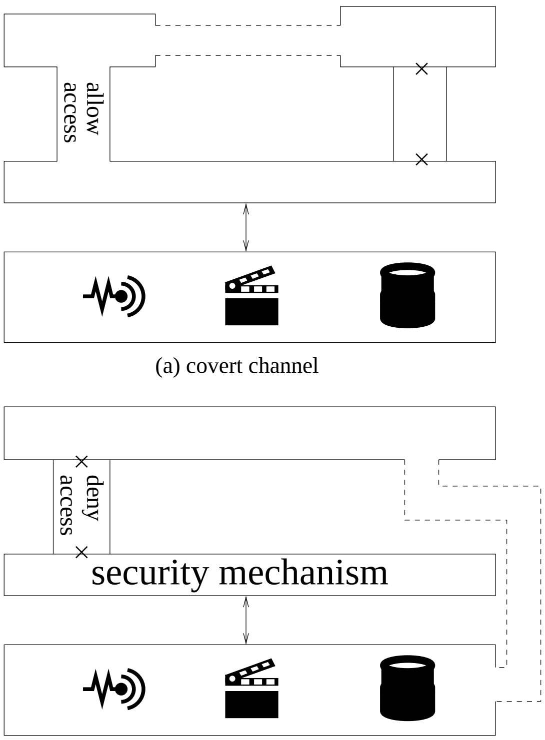
The Android permission model has two important aspects: obtaining user consent before an app is able to access any of its requested permission-protected resources, and then ensuring that the app cannot access resources for which the user has not granted consent. There is a long line of work uncovering 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 [24,30,78,86]. 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 [24,81]. In practice, this is not always the case.

2.2 Circumvention

Apps can circumvent the Android permission model in different ways [3,17,49,51,52,54,70,72,74]. 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 ad-



(a) covert channel

(b) side channel

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.

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 approaches 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 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 [16,28]. 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 [40,44,63]. 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

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

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 traf c. However, identifying personal information inside network transmissions requires signi cant effort because apps and embedded third-party SDKs often use different encodings and obfuscation techniques to transmit data. Thus, it is a signi cant technical challenge to be able to de-obfuscate all network traf c and search it for personal information. This subsection discusses how we tackle these challenges in detail.

Personal Information We de ne “personal information” as any piece of data that could potentially identify a speci c 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 identi ers 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 speci c types of highly sensitive data, including persistent identi ers and geolocation information. Notably, the unauthorized collection of geolocation information in Android has been the subject of prior regulatory action [82]. Table 1 shows the different types of personal information that we look for in network transmissions, what each can be used for, the Android permission that protects it, and the subsection in this paper where we discuss ndings that concern side and covert channels for accessing that type of data.

Decoding Obfuscations In our previous work [66], 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 ows, 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 nd 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 nding 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

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		Channel Type	
				Dynamic	Static	Dynamic	Static	Covert	Side
IMEI	READ_PHONE_STATE	Persistent ID	4.1	13	159	2	2	2	0
Device MAC	ACCESS_NETWORK_STATE	Persistent ID	4.2	42	12,408	1	1	0	1
Email	GET_ACCOUNTS	Persistent ID	Not Found						
Phone Number	READ_PHONE_STATE	Persistent ID	Not Found						
SIM ID	READ_PHONE_STATE	Persistent ID	Not Found						
Router MAC	ACCESS_WIFI_STATE	Location Data	4.3	5	355	2	10	0	2
Router SSID	ACCESS_WIFI_STATE	Location Data	Not Found						
GPS	ACCESS_FINE_LOCATION	Location Data	4.4	1	1	0	0	0	1

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

Android protects access to the phone's IMEI with the READ_

4.2 Network MAC Addresses

The Media Access Control Address (MAC address) is a 6-byte identi er 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 identi er, 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

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	huq.io						

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

While this app may not be intending to circumvent the permission system, this technique can be exploited by a malicious actor to gain access to the user's location. Whenever a new picture is taken by the user with geolocation enabled, any app with read access to the photo library (i.e., READ_EXTERNAL_STORAGE) can learn the user's precise location when said picture was taken. Furthermore, it also allows obtaining historical geolocation xes 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 eld o18 0 0 9.9626T53 (o189steHyphe-J254 614.032 Tm [(li0 9.962l63 637i6)-4250 9l1(10

other permissions that, while not labeled as dangerous, can still give access to sensitive user data. One example is the BLUETOOTH

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

[8] AppCensus Inc. Apps using Side and Covert Channels. https://appcensus.mobi/usenix2019, 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 speci cation. 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 nancial 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- ow 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.

[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]

C..22.005Tnam2017 I339.459p588.596 Sk. C.

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

[75] 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.

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

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

[78] 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.

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

[80] 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.

[81] U.S. Federal Trade Commission. In the Matter of HTC America, Inc. https://www.ftc.gov/sites/default/files/
naj EMTd ([75])Tj EMC /Span <<99626 75.579 271.681 Tm [2 [(Statista.)-110 (Global0logies.)-110 (Unity 310 (Children')55 (s online)]TJ 2.166 -1.2