Julien Gamba

"Do Androids Dream of Electric Sheep?"

On Privacy in the Android Supply Chain

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

Universidad Carlos III de Madrid 2022

"Do Androids Dream of Electric Sheep?"

On Privacy in the Android Supply Chain

Prepared by:

Julien Gamba, IMDEA Networks Institute, Universidad Carlos III de Madrid

contact: julien.gamba@imdea.org

Under the advice of:

Narseo Vallina-Rodriguez, IMDEA Networks Institute

This work has been supported by:

Android logo source: Google

Pour Anne-Laure

Acknowledgements

It would be impossible to thank all the people that helped me get through this journey, even though I have to try. First and foremost, I want to thank my family for their love and encouragement over the last few years. Without their support I surely would not have reached this milestone. This is especially true for my wife, Anne-Laure, who knows firsthand the true amount of work that went into this thesis. Thank you.

I would like to thank my advisor, Narseo Vallina-Rodriguez, for his help and support during my PhD, and all of my co-authors, for their contributions to this thesis. I also want to thank Cristel Pelsser, Pascal Mérindol, and Stéphane Cateloin for introducing me to the world of scientific research back at the University of Strasbourg.

Finally, I want to thank all my friends and colleagues for bearing with me for the past few years, Álvaro Feal, Aniketh Girish, Benjamin Chetioui, and Constantine Ayimba in particular. There are too many of you to list here, but if you recognize yourself along these lines: thank you.

Julien

Published and Submitted Content

The following list includes other research papers I have co-authored during the course of my PhD, and that are included in this thesis.

1. *An Analysis of Pre-installed Android Software*

   Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Narseo Vallina-Rodriguez and Juan Tapiador

   In IEEE Symposium on Security and Privacy 2020, 18-20 May, San Francisco, CA, USA

   DOI: https://doi.org/10.1109/SP40000.2020.00013

   - The author's role in this work is focused on designing the data collection strategy, the ecosystem and static analysis of apps.
   - This paper is fully included in this thesis as chapter 5.
   - The material from this source included in this thesis is not singled out with typographic means and references.

   This paper has been awarded the **best practical paper award** at the symposium, the **CNIL-INRIA Privacy Protection Award** and the 2020 **AEPD Emilio Aced Prize**.

2. *Mules and Permission Laundering in Android: Dissecting Custom Permissions in the Wild*

   Julien Gamba, Álvaro Feal, Eduardo Blazquez, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez

   Submitted to IEEE Transactions on Dependable and Secure Computing on the 2nd of May, 2022

   - The author's role in this work is focused on designing the data collection strategy, the ecosystem analysis, and the design and implementation of the app static analysis tools.
   - This paper is fully included in this thesis in Chapters 7.

- The material from this source included in this thesis is not singled out with typographic means and references.

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

   DOI: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00095

   - The author's role in this work is focused on the data collection of pre-installed apps, as well as the initial permission analysis.
   - This paper is partly included in this thesis in Chapters 2 and 5.
   - The material from this source included in this thesis is not singled out with typographic means and references.

Other Research Merits

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

1. *Mixed Signals: Analyzing Software Attribution Challenges in the Android Ecosystem*
   Kaspar Hageman, Álvaro Feal, **Julien Gamba**, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, Narseo Vallina-Rodriguez
   Submitted to IEEE Transactions on Software Engineering on the 13th of April, 2022

2. *Not Your Average App: A Large-scale Privacy Analysis of Android Browsers*
   Amogh Pradeep, Álvaro Feal, **Julien Gamba**, Ashwin Rao, Martina Lindorfer, Narseo Vallina-Rodriguez, and David Choffnes
   Submitted to Privacy Enhancing Technologies Symposium (PETS) 2023, Lausanne, Switzerland.

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

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

5. *Don't Accept Candy from Strangers: An Analysis of Third-party SDKs*
   Álvaro Feal, **Julien Gamba**, Narseo Vallina-Rodriguez, Primal Wijesekera, Joel Reardon, Serge Egelman and Juan Tapiador
   In Computers, Privacy and Data Protection Conference (CPDP), 22-24 January 2020,

Brussels, Belgium

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

7. *A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists*
   Quirin Scheitle, Oliver Hohlfeld, **Julien Gamba**, Jonas Jelten, Torsten Zimmermann, Stephen Strowes and Narseo Vallina-Rodriguez
   In ACM IMC 2018, Oct. 31 - Nov. 2, Boston, MA, USA

   This paper has been awarded the **ACM SIGCOMM Community Contribution Award.**

8. *An Analysis of Pre-installed Android Software*
   Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Narseo Vallina-Rodriguez and Juan Tapiador
   In National Cybersecurity Research Conference (JNIC) 2018, San Sebastián, Spain.

9. *This Is My Private Business! Privacy Risks on Adult Websites*
   Pelayo Vallina-Rodriguez, **Julien Gamba**, Álvaro Feal, Narseo Vallina-Rodriguez and Antonio Fernández-Anta
   In National Cybersecurity Research Conference (JNIC) 2018, San Sebastián, Spain.

Abstract

The Android Open Source Project (AOSP) was first released by Google in 2008 and has since become the most used operating system [Andaf]. Thanks to the openness of its source code, any smartphone vendor or original equipment manufacturer (OEM) can modify and adapt Android to their specific needs, or add proprietary features before installing it on their devices in order to add custom features to differentiate themselves from competitors. This has created a complex and diverse supply chain, completely opaque to end-users, formed by manufacturers, resellers, chipset manufacturers, network operators, and prominent actors of the online industry that partnered with OEMs. Each of these stakeholders can pre-install extra apps, or implement proprietary features at the framework level.

However, such customizations can create privacy and security threats to end-users. Pre-installed apps are privileged by the operating system, and can therefore access system APIs or personal data more easily than apps installed by the user. Unfortunately, despite these potential threats, there is currently no end-to-end control over what apps come pre-installed on a device and why, and no traceability of the different software and hardware components used in a given Android device. In fact, the landscape of pre-installed software in Android and its security and privacy implications has largely remained unexplored by researchers.

In this thesis, I investigate the customization of Android devices and their impact on the privacy and security of end-users. Specifically, I perform the first large-scale and systematic analysis of pre-installed Android apps and the supply chain. To do so, I first develop an app, Firmware Scanner [Sca], to crowdsource close to 34,000 Android firmware versions from 1,000 different OEMs from all over the world. This dataset allows us to map the stakeholders involved in the supply chain and their relationships, from device manufacturers and mobile network operators to third-party organizations like advertising and tracking services, and social network platforms. I could identify multiple cases of privacy-invasive and potentially harmful behaviors. My results show a disturbing lack of transparency and control over the Android supply chain, thus showing that it can be damageable privacy- and security-wise to end-users.

Next, I study the evolution of the Android permission system, an essential security feature

of the Android framework. Coupled with other protection mechanisms such as process sandboxing, the permission system empowers users to control what sensitive resources (e.g., user contacts, the camera, location sensors) are accessible to which apps. The research community has extensively studied the permission system, but most previous studies focus on its limitations or specific attacks. In this thesis, I present an up-to-date view and longitudinal analysis of the evolution of the permissions system. I study how some lesser-known features of the permission system, specifically permission flags, can impact the permission granting process, making it either more restrictive or less. I then highlight how pre-installed apps developers use said flags in the wild and focus on the privacy and security implications. Specifically, I show the presence of third-party apps, installed as privileged system apps, potentially using said features to share resources with other third-party apps.

Another salient feature of the permission system is its extensibility: apps can define their own *custom permissions* to expose features and data to other apps. However, little is known about how widespread the usage of custom permissions is, and what impact these permissions may have on users' privacy and security. In the last part of this thesis, I investigate the exposure and request of custom permissions in the Android ecosystem and their potential for opening privacy and security risks. I gather a 2.2-million-app-large dataset of both pre-installed and publicly available apps using both Firmware Scanner and purpose-built app store crawlers. I find the usage of custom permissions to be pervasive, regardless of the origin of the apps, and seemingly growing over time. Despite this prevalence, I find that custom permissions are virtually invisible to end-users, and their purpose is mostly undocumented. While Google recommends that developers use their reverse domain name as the prefix of their custom permissions [Gpla], I find widespread violations of this recommendation, making sound attribution at scale virtually impossible. Through static analysis methods, I demonstrate that custom permissions can facilitate access to permission-protected system resources to apps that lack those permissions, without user awareness. Due to the lack of tools for studying such risks, I design and implement two tools, `PermissionTracer` [Pere] and `PermissionTainter` [Perd] to study custom permissions. I highlight multiple cases of concerning use of custom permissions by Android apps in the wild.

In this thesis, I systematically studied, at scale, the vast and overlooked ecosystem of pre-installed Android apps. My results show a complete lack of control of the supply chain which is worrying, given the huge potential impact of pre-installed apps on the privacy and security of end-users. I conclude with a number of open research questions and future avenues for further research in the ecosystem of the supply chain of Android devices.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"*The story so far:*
In the beginning the Universe was created.
This has made a lot of people very angry and been widely regarded as a bad move."

— Douglas Adams, *The Restaurant at the End of the Universe* (1980)

The Android operating system (OS) started as Android Inc, a California based start-up found in October 2003 [Andw] by Andy Rubin, Rich Miner, Nick Sears, and Chris White. The founders' initial goal was to create an OS for smart camera based on the Linux kernel, but then later adapted it to smartphones [Andl]. Google bought the company in 2005 [Goob], and went on to develop the operating system.

In 2007, Google, along with 34 other tech companies, unveiled the Open Handset Alliance (OHA), a consortium dedicated to developing open standards for mobile devices. The announced goal was to create "*the first truly open and comprehensive platform for mobile devices*" [Andy]. A preview source code of the first Android Software Development Kit (SDK) was released a week after that announcement to attract developers [Andag]. The first official version of Android was released in 2008 [Andk], along with the first Android-powered smartphone, the HTC Dream [Htc].

Android has since grown to be the most used OS, with at least three billion active devices as of May 2021 [Andaf].[1] A major factor to this rapid adoption of Android is its open source model [Aosb]. Each component of the OS can be modified by a phone manufacturer or an original equipment manufacturer (OEM, a contractor that can produce a device on behalf of another company) before being installed on a device. Such customization of the OS are en-

[1] There are reasons to believe that the true number of active Android devices is even higher, as the three billion figure only includes devices that use Google's Play Store, and so does not take into account devices that use alternative app stores.

Figure 1: Simplified illustration of the Android supply chain

couraged by Google [Andu]. In practice, manufacturers take advantage of the openness of the OS to distinguish themselves from their market competitors.

Because of this openness, the supply chain of Android devices involves a large and diverse number of stakeholders, from the creation of the device to its manufacturing and distribution. In Figure 1, I illustrate the typical stakeholders involved in the supply chain, from the design to the end user. While the actual number of stakeholders can vary depending on the complexity of the model and the commercial partnerships between them and other companies, the commercialization of most devices involves the following actors:

- **Chipset manufacturers:** at the beginning of the supply chain are the chipset manufacturers such as Qualcomm or MediaTek. These companies are responsible for the manufacturing of essential electronic components and provide software (including apps and drivers) to interact with said components.
- **Device manufacturers:** these actors are the most visible ones, as it involves the brands known to the end user. Device manufacturers are the companies that actually assemble the components and load the firmware. Here I also include Original Equipment Manufacturers (OEM) and Original Device Manufacturers (ODM) which are contractors hired by the phone vendors to externalize the manufacturing of the phones (an OEM will manufacture a device based on the vendor's design, while an ODM will create a design, potentially from scratch, and manufacture the devices).
- **App markets:** certified devices can come with the Google apps suite pre-installed (e.g., Google Play Store, YouTube, Gmail), which also includes the Google Play Store app. Only phones certified by the Google Android team can pre-install the Google Play Store [Andb]. Devices can also pre-install alternative app marketplaces, such as the Amazon Appstore [Amaa]. Devices can also come pre-installed with regional app markets [Wan+18a] (e.g., Chinese devices may come pre-installed with app stores from Baidu [Baia] or Tencent [Tena]).
- **Mobile Network Operators (MNO):** MNOs can create strategic partnerships with ven-

dors to sell devices to users at lower prices in exchange for a subscription to their services. In these cases, MNOs can pre-install apps to add value to the device or to ease access to MNOs-related services (e.g., a companion app to keep track of data consumption and data plan).

- **Resellers and distributors:** finally, at the end of the supply chain are resellers and distributors. This includes brick-and-mortar shops as well as online shops such as Amazon or eBay.

These stakeholders are the ones of the typical supply chain of an Android device, but the actual supply chain of a given device can vary across brands. In fact, what makes the Android supply chain unique is the variable number of stakeholders that can be involved at any point, and its diversity: for instance, two copies of the same device model might have a different set of pre-installed apps depending on the country in which they were bought. Moreover, the supply chain can be dynamic, with extra apps installed without interaction with the user when they first boot the device. Any of the stakeholders can also pre-install software from their partners, expose features to other apps on the device, or even change core Android components, thus giving these stakeholders access to a privileged vantage point to get information on the user. Indeed, pre-installed[2] apps are trusted by the system by default, and can even be pre-granted permissions, without user interaction. Once installed, it is very difficult for a user to remove them, if possible at all.

Android Supply Chain Issues

Not all pre-installed software is deemed as wanted by users, and the term "bloatware" is often applied to such software. The process of how a particular set of apps end up packaged together in the firmware of a device is not transparent, and various isolated cases reported over the last few years suggest that it lacks end-to-end control mechanisms to guarantee that shipped firmware is free from vulnerabilities [Huaa; Sama] or potentially malicious and unwanted apps. For example, at Black Hat USA 2017, Johnson et al. [Joh; Kryb] gave details of a powerful backdoor present in the firmware of several models of Android smartphones, including the popular BLU R1 HD. In response to this disclosure, Amazon removed Blu products from their Prime Exclusive line-up [Amac]. A company named Shanghai Adups Technology Co. Ltd. was pinpointed as responsible for this incident. The same report also discussed the case of how vulnerable core system services (e.g., the widely deployed MTKLogger compo-

[2]Throughout this thesis, I use the terms "*pre-installed*" and "*pre-loaded*" interchangeably to designate system apps. This covers any app in the system partition, including those which could be installed dynamically after purchase through FOTA components

nent developed by the chipset manufacturer MediaTek) could be abused by co-located apps. The infamous Triada trojan has also been recently found embedded in the firmware of several low-cost Android smartphones [Dr ; Tri]. Other cases of malware found pre-installed include Loki (spyware and adware) and Slocker (ransomware), which were spotted in the firmware of various high-end phones [Lok].

Android handsets also play a key role in the mass-scale data collection practices followed by many actors in the digital economy, including advertising and tracking companies. OnePlus has been under suspicion of collecting Personally Identifiable Information (PII) from users of its smartphones through exceedingly detailed analytics [Oned; Onec], and also deploying the capability to remotely root the phone [Oneb; Onea]. In July 2018 the New York Times revealed the existence of secret agreements between Facebook and device manufacturers such as Samsung [Fac] to collect private data from users without their knowledge. This is currently under investigation by the US Federal authorities [Nyt]. Additionally, users from developing countries with lax data protection and privacy laws may be at an even greater risk. The Wall Street Journal has exposed the presence of a pre-installed app that sends users' geographical location as well as device identifiers to GMobi, a mobile-advertising agency that engages in ad-fraud activities [New; Ups]. Recently, the European Commission publicly expressed concern about Chinese manufacturers like Huawei, alleging that they were required to cooperate with national intelligence services by installing backdoors on their devices [Huab]. In March 2019, it was reported that hackers managed to hijack the update process of Asus computers to install malware [Asub]. While this does not involve pre-installed apps, it is a prime example of the consequences of the size and lack of control over the agents forming the supply chain.

To make sure that every device can properly run any app regardless of their level of customization, Google has set up a compatibility program, that states the minimum requirements that the modified OS must meet to stay compatible with standard Android apps [Andg]. However, this compatibility program only sets software requirements and does not consider security and privacy implications for the end user. Google also created a certification for devices, to assess for their security and performance. Both phone vendors and ODMs can make their devices certified [Cer]. This certification is mandatory in order to pre-install Google apps and the Google Play Store on a device. Unfortunately, there is little information available regarding the tests that are actually performed by Google before certifying devices, and it is not clear at which stage of the manufacturing the tests are performed.

The Android Permission System and its Extensibility

Another vector for customization available to stakeholders of the supply chain is the Android permission system. The Android OS implements a permission-based mechanism to control how apps can access sensitive data and dangerous system features [Gplb] such as user contacts, the camera, location sensors, or the system settings. Coupled with other protection mechanisms such as process sandboxing, the permission system empowers users to control what sensitive resources are accessible to which apps. The Android Open Source Project (AOSP) defines a standard set of permissions that are supported by most Android devices. Any Google-certified device [Ande; Andf] must implement the whole set of AOSP permissions to guarantee their compatibility with the standard Android platform [Andh]. A decade of research in the use, enforcement, and usability of AOSP permissions has revealed severe privacy and security shortcomings inherent to the Android permission model [Fel+11a; Wij+15; Fel+12; Au+12a; SC13; Rea+19; BNN17; Des14]. Consequently, many vulnerabilities were fixed gradually across Android releases.

Android's permission model possesses an interesting, overlooked feature: its extensibility. By design, the Android framework allows any app developer to share features implemented in their software with other apps in a "controlled" way by defining custom permissions [Gpla]. Therefore, custom permissions allow extending the capabilities offered by the Android OS and the creation of new features exposed by pre-installed apps and facilitate the flourishment of an open software ecosystem in which apps (and third-party libraries or SDKs) can share data and components with other developers. However, custom permissions also pose security and privacy risks as they can be (ab)used–intentionally or by mistake–to circumvent the standard permission system and provide backdoored access to privileged data and features to apps that are otherwise not permitted to do so, in a way akin to how covert and side channels operate [Rea+19].

The control and transparency mechanisms implemented by the Android operating system are insufficient to protect users from abusive or insecure implementations of custom permissions. Google recommends using the reverse domain name as the prefix of such permissions, and supplying a description of the custom functionality or data protected by the permission [Gpla; Andj], but, in practice, there is no enforcement of such recommendations [Gam+20]. Consequently, it is not possible to automatically know what precise function or resource is protected by a custom permission, and how they are being integrated and used across Android apps. This lack of control and transparency also manifests at installation time, which translates into profound implications in terms of user awareness and control: unlike official AOSP

permissions custom permissions are not listed in the app stores, and end users cannot grant or deny apps access to them at runtime unless the developer willingly defines them with a `dangerous` protection level.

Despite more than a decade of research into the Android ecosystem, the ecosystem of pre-installed Android software and its associated privacy and security concerns have remained neglected by the research community. This ecosystem has remained largely unexplored due to the inherent difficulty to access such software at scale and across vendors. This state of affairs makes this work even more relevant, since *i*) these apps – typically unavailable on app stores – have mostly escaped the scrutiny of researchers and regulators; and *ii*) regular users are unaware of their presence on the device, which could imply lack of consent in data collection and other activities. Similarly, the research literature focused on the evolution of the permission system or on custom permissions is significantly narrow. As of now, no app analysis tool has been able to capture the asynchronous behavior of custom permissions. Prior work demonstrated, using proof-of-concept implementations, how custom permissions can enable permission re-delegation and confused deputy attacks [Bag+15; Bag+18; Tun+18; Li+21]. Yet, our understanding of the Android custom permissions landscape has remained low, particularly in terms of their prevalence, usage, and potential misuse.

1.1 Research Questions and Objectives

Analyzing at scale the customization of Android devices poses a certain number of challenges. As discussed previously, the openness of the Android ecosystem has led to the complexification of the supply chain. There is a myriad of stakeholders that can pre-install apps, each with their own business model and practices, therefore gaining privileged access to system resources and potentially users' personal data.

Moreover, pre-installed apps differ from publicly available apps: while a publicly available app is standalone, pre-installed apps developers know in advance the environment in which their app will run, i.e., the software and hardware specification of the device. Pre-installed apps can safely rely on specific libraries or even other apps that will also come pre-loaded on the device, for specific operations. As a consequence, pre-installed apps tend to use more features of the Android OS such as shared user IDs[3], to pool resources with other apps, or custom permissions, to expose some of their components to a specific set of other apps on the device; such features are typically less common in publicly available apps. I develop in

[3]The `sharedUserId` is a manifest attribute that allows a developer to specify the UID of their app, instead of using a random one assigned by the OS [Sha]. Two apps signed by the same signing certificate that have the same UID can share data more easily, as I will explain further in Chapter 2 (page 15).

depth such aspects in Chapter 2 (page 15). This can hinder the use of state of the art static and dynamic analysis tools, as such tools expect a standalone entity to analyze, and might miss inter-component and inter-app communication, or call to functions defined by non-standard libraries that are otherwise not present in their emulated environment [Sto].

RQ1: Exploring the system Android apps ecosystem

The majority of system apps are not publicly available and have therefore escaped the scrutiny of the research community. This is especially worrying, as apps installed on system partitions hold a privileged position in the Android operating system. However, the state of the art has not produced any method to gather pre-installed apps directly from users' devices, and relied instead on crawling firmware images or on buying devices. None of those methods scale well. It is, therefore, necessary to design novel, scalable methods to gather system apps.

There is also a dynamic component to the supply chain, which further complicates its analysis. Modern devices include mechanisms to install updates for system apps, usually under the form of a Fimware Over The Air (FOTA) app, which has the possibility of updating apps even if said apps are installed on a system partition, or installing new apps on those partitions. This implies that a FOTA app also has the ability to install new system apps, possibly after the user start using the device, with the same issues as pre-installed ones.

The exploration of the modern Android supply chain is a necessary first step to uncover its stakeholders and the relationship between them. I design an innovative crowdsourcing method to collect pre-installed apps on users' devices in a privacy-respecting manner, giving us an accurate overview of the ecosystem in the wild, and allowing us to study its main stakeholders.

RQ2: Measuring the consequences on user privacy and security

Given the scale of the Android supply chain, and the high number of third parties that can pre-install extra system apps, it is paramount to conduct a privacy and security analysis of these apps. I first use static and dynamic analysis to try and understand the purpose of these apps and highlight numerous potentially harmful behaviors in both high and low-end devices. I specifically focus on apps that can access the full, unfiltered system logs–which might contain private information–and manually analyze their code to understand in which circumstances they access them, and whether they upload them on the Internet.

Another critical part of users' security and privacy is the Android permission system. This system has evolved over time, and it is unclear what the impact is on users' security and privacy. Specifically, I study how third-party apps make use of lesser-known features of the

permission system to potentially make features available to other apps.

Finally, custom permission can open the door to privacy and security abuses. The aforementioned limitations make the automatic detection of privacy-invasive or malicious behaviors due to custom permissions challenging at best. I first evaluate the usability of state of the art tools for the analysis of pre-installed apps and show that they are not suitable for these purposes. I then develop my own analysis tools targeted specifically at these apps. Once I have suitable tools for analysis, I conduct a large-scale privacy analysis of the pre-installed apps ecosystem.

1.2 Contributions and Organization

In this thesis, I answer the research questions discussed above and make several contributions to advance the state of the art.

Exploration of the pre-installed apps ecosystem

I first design a novel crowdsourcing method to create a dataset of pre-installed Android apps. I created an app, Firmware Scanner, freely available on Google Play, that scans the system partitions of Android phones and uploads pre-loaded apps to our server, along with metadata about the device (e.g., information about the brand and model of the device, or the MCC and MNC codes and country code of the SIM card). This metadata allows to identify stakeholders of the supply chain and attribute customizations back to them. With this tool, I was able to gather 1,309,968 unique apps (according to their MD5 hash) from 33,915 unique devices (according to their build fingerprint), coming from 1,050 unique vendors. This app relies on crowdsourcing mechanisms, which allows me to capture also system apps dynamically installed. This data is coming from devices from every continent, giving us an unprecedented overview of this ecosystem, including regional customizations.

Armed with this dataset, I present in Chapter 5 (page 47) the first large-scale study of pre-installed software and the supply chain on a global scale. This dataset allows us to characterize the stakeholders involved in the supply chain, from device manufacturers and mobile network operators to third-party organizations like advertising and tracking services, and social network platforms. To do this, I mainly rely on the analysis of information available in the manifest of the app packages, their signing certificates, and the third-party libraries (TPLs) they embed. my analysis covers 1,200 unique developers associated with the major manufacturers, vendors, and Internet service companies. I also uncover a vast landscape of third-party services (11,665 unique third-party libraries) revolving around advertisement, analytics, and

social networking services. This chapter also explores the relationships between these stakeholders, by analyzing the custom permissions defined by hardware vendors, MNOs, third-party services, security firms, industry alliances, chipset manufacturers, and Internet browsers. Such permissions can potentially expose data and features to over-the-top apps and can be used to access privileged system resources and sensitive data in a way that circumvents the Android permission model. A manual inspection illuminates a complex supply chain that involves different stakeholders and commercial partnerships between handset vendors and online service providers. Overall, I show evidence that the supply chain around Android's open source model lacks transparency and has facilitated potentially harmful behaviors and backdoored access to sensitive data and services without user consent or awareness.

Privacy analysis of pre-installed Android apps

In Chapter 5, I report numerous potentially harmful behavior in pre-installed apps, coming from both first and third parties. I find that user tracking is prevalent in the pre-installed apps ecosystem and that some apps abuse their privileged position by requesting permissions usually reserved for system apps. In particular, I show in depth how some apps access the full, unfiltered system logs, and then either store them on the SD card of the device or even upload it on the Internet.

I then present in Chapter 6 a temporal evolution of the permission system, both in terms of the number of AOSP permissions but also in complexity. I show the vast number of flags that can be used by developers to slightly alter the permission granting algorithm, and their impact on the permission granting algorithm which I formalize. Then, I show how these flags are used by pre-installed apps in the wild, including by third-party system apps, which can then make some features available to other apps on the device.

I then focus on custom permissions, which could potentially be used to circumvent the AOSP permission system. The use of custom permissions is not limited to pre-installed apps, however, and there could in fact be collusion between such apps and publicly available ones. Therefore, I decide to investigate the custom permissions ecosystem as a whole, including apps from any origin. In Chapter 7, I gather a 2.2-million-app-large dataset of both pre-installed and publicly available apps from 8 different app stores, complemented with apps downloaded from the Androzoo project [All+16]. With this dataset, I present the first longitudinal and large-scale analysis of the usage of custom permissions in the Android ecosystem. I find that both pre-installed and public apps both define and request a large number of custom permissions. Namely, 58% and 67% of pre-installed and public apps request at least one, and 26% and 4%

define at least one, respectively. I find widespread violations of the naming recommendations set by Google for custom permissions: 45% of definitions do not follow that recommendation. For example, I find 722 custom permissions with the `android.permission` prefix, which is explicitly forbidden by the Android Compatibility Definition Document (CDD). Despite this prevalence, I find that custom permissions are virtually invisible to end users, and their purpose is mostly undocumented. While there is a `description` tag to describe the purpose and functionality of the custom permission, its usage is optional and I find that it is rarely used by developers (missing in 75% of the cases).

This lack of transparency can lead to serious security and privacy problems: I show that custom permissions can facilitate access to permission-protected system resources to apps that lack those permissions without user awareness. However, there were no available tools to trace and understand the type of data or capability that is protected by a given custom permission, making it difficult to assess their risk from a user's privacy and security perspective. To fill this gap, I present a novel method to triage apps that are potentially misusing custom permissions to access personal data, or perform other actions potentially detrimental to users' privacy and security. My method relies on two custom-made tools: (1) `PermissionTracer`, a tool that reports potentially-dangerous custom permissions and detects potential cases of a privilege escalation attack in which an attacker can access permission-protected information using custom permissions; and (2) `PermissionTainter`, a static taint analysis tool that inspects the bytecode of apps that define custom permissions, to identify potential privacy leaks due to those permissions. Thanks to these tools, I identify several potentially harmful implementations where an attacker could access sensitive data such as the location, Wi-Fi MAC address, or contacts without requesting the corresponding AOSP permission.

Finally, I conduct a small-scale survey of app developers who defined some of these custom permissions in order to understand their use case and rationale. My findings suggest that most developers lack a clear understanding of their purpose and functioning. As a result, custom permissions are often used due to poor software development practices or because they are required to define them in order to integrate third-party SDKs.

1.3 Outline of this Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of the Android operating system. I also present the different test suites that Google has created to ensure that modified Android devices remain compatible with an unmodified version, and the different mechanisms put in place to push updates to the system. The second part of this

chapter is dedicated to the Android permission system. In Chapter 3, I present the studies from the state of the art most relevant to my contributions.

The second part of this thesis contains the main contributions of my work. This part starts with Chapter 4, where I describe my novel method to collect pre-installed apps from users' devices, relying on crowdsourcing mechanisms. Chapter 5 presents the first large-scale exploration of the ecosystem of pre-installed apps, relying on a dataset of more than 82,000 apps from more than 1,700 unique devices. I uncover the vast landscape of various organizations that compose the supply chain of Android devices, including companies with a data-driven business model. In Chapter 6, I present the temporal evolution of the Android permission system and highlight multiple cases of privileged pre-installed apps making features available that are potentially used by other third-party apps. Finally, I present in Chapter 7 a deep dive into custom permissions and their usage in the wild. I present the design and functioning of two tools I created, `PermissionTracer` and `PermissionTainter`, to detect custom permissions that potentially facilitate access to permission-protected resources.

I conclude this thesis with a discussion of my main results in Chapter 8 and present the main conclusions and possible future research directions in Chapter 9.

I

The Android Operating System

Chapter 2

Android

"*Would it save you a lot of time if I just gave up and went mad now?*"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

Since its first release, Android became increasingly powerful and complex. This added complexity is due in part to new features being added at every new release, but also to harden the system to mitigate new security vulnerabilities or privacy issues. Moreover, the openness of the operating system allows for any vendor to modify it, adding another layer of complexity to the OS. This translates into heterogeneous OS versions and potential security issues, a problem otherwise known as "fragmentation" of the OS. In this chapter, we first describe the architecture of the Android OS and its different components (§2.1). We will also discuss customization and compatibility issues, and focus on the parts of the OS that we find are heavily modified in the wild, pre-loaded apps and the Android framework in particular. Finally, we present an in-depth description of the Android permission system, a central element of the operating system that can be extended by stakeholders of the supply chain, most notably by the introduction of extra custom permissions defined by pre-loaded apps (§2.2).

2.1 Android Architecture

2.1.1 The Layers of Android

Modern operating systems are not monolithic pieces of software, and Android is no exception. The Android OS is made of multiple pieces organized in different layers, as illustrated in Figure 2.

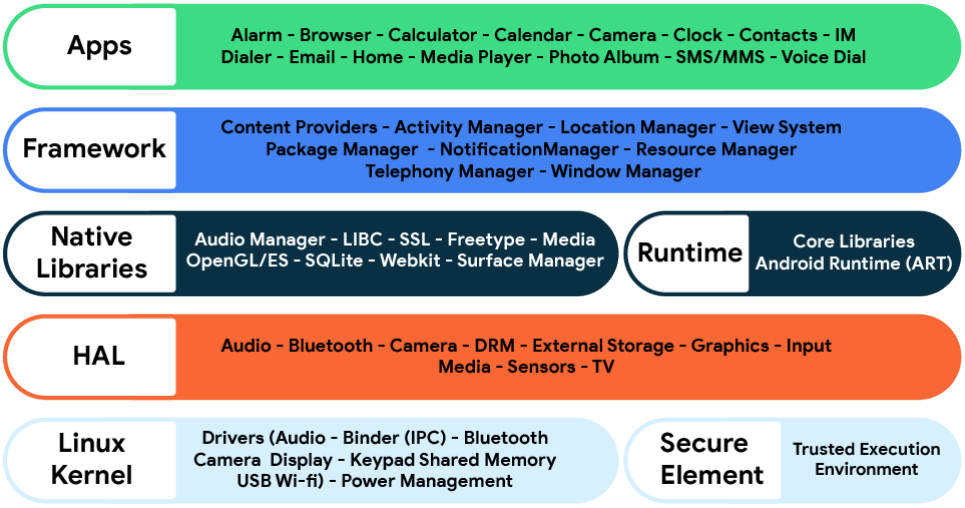


Figure 2: Android stack (source: https://source.android.com/)

Android apps

At the top layer of the stack are Android apps. We can distinguish two categories of apps: user-installed and system apps, i.e., apps installed by any stakeholder of the supply chain on one of the system partitions, which are read-only. Users cannot install new system apps on their devices unless they root it first. Only FOTA apps, a critical system component responsible for installing new apps or updating existing ones, can dynamically modify the content of the system partitions after the phone is built. We detail more the role of FOTA apps in section 2.1.3.

Each Android app must embed a *manifest* file that contains essential information about the app, such as its package name and version, the list of its components, and the list of permission it requests [Appa]. In addition, each app must be signed with at least one digital certificate [Sig]. Android uses this certificate as a proxy to identify the author of an app. However, Android accepts self-signed certificates, and there is no validation of the information inputted in the certificate prior to publishing which can lead to attribution issues, as we will see in depth in Chapter 5 (page 47).

There are no technical differences between system or user-install apps. However, system apps are implicitly more trusted by the system, as they can only be installed by a trusted party (e.g., the phone manufacturer or a FOTA component). As a result, there are some permissions that are only available to system apps. Such restricted permissions protect the most sensitive resources of the OS (for instance, `android.permission.READ_LOGS` gives access to the system logs, which can contain sensitive information). We dig more in-depth into the Android permission system in Section 2.2.

The framework

The Android framework bridges the gap between the operating system and the apps. This allows developers to use higher-level languages such as Java or Kotlin to develop their Android apps. It includes all of the application programming interfaces (APIs) that are made available for developers to interact with the hardware (e.g., to get access to the user's geolocation) but also contains several APIs to facilitate app development (e.g., support for user accounts or UI components). In practice, the framework is bundled into several `.jar` files loaded into the `/system/framework` folder.

Some of the APIs defined by the framework can allow access to sensitive data or system components (e.g., the GPS of the device). Offering unfettered access to these APIs and sensors would pose grave security and privacy problems to users. The Android OS solves this problem by implementing a permission system, which allows the user to allow (or not) access to a given protected API. We describe in detail this permission system later in this chapter. We will also study its temporal evolution in Chapter 6 (page 83), and focus on its extensibility (and the underlying privacy and security issues) in Chapter 7 (page 93).

Native libraries

The Android OS includes several libraries by default, for a wide range of tasks (e.g., cryptography, database management). This also includes drivers for the hardware components of the device. Such libraries, as most Linux distributions, are compiled from C or C++ into ELF binaries, which are then stored in the system partitions of the device. Some vendors can decide to include extra libraries, either for specific hardware that their device embeds, or to enable extra functionalities for developers.

Android runtime

The Android RunTime (ART) is Android's runtime environment. It is the part of the OS that executes the code of the app on the device. Initially, the OS was using the Dalvik Virtual Machine (DVM) as its runtime environment. With the DVM, the Java bytecode of Android apps would be translated into Dalvik bytecode, which would then be interpreted by the DVM. The DVM would also profile apps as they run, and compile the most used segments of the code into native code to increase performance (this process is called trace-based just-in-time compilation). Android versions 5.0 and up do not use the DVM, and instead use ART, which relied on ahead-of-time compilation. ART still uses the Dalvik bytecode format, to maintain backward compatibility.

Android apps can also use native libraries directly through the use of the Java Native Interface (JNI). This allows apps to escape the virtual machine and execute code directly from native libraries, which can be helpful if an app needs to interact with a hardware component. Google provide the Native Development Kit (NDK) [Andm], to help developers build native libraries and include C or C++ code into their Android Package (APK).

Hardware abstraction layer

The Hardware Abstraction Layer (HAL) is another layer of abstraction, on top of the Linux kernel. It is a set of standard interfaces that must be implemented by chipset manufacturers, in order to function properly, regardless of the potential modifications added to the rest of the operating system. Essentially, the HAL defines the interface that the Android OS expects for a given category of device (e.g., camera, GPS).

Secure element

Some devices provide a secure element, which provides cryptographically secure storage of data on the device (e.g., digital keys, credentials). The secure element is based on a separate tamper-resistant hardware chipset and a standard API: Open Mobile API [Oma].

Linux kernel

Finally, at the bottom layer is the Linux kernel. The precise version of the kernel used varies depending on the Android version. As for any Linux distribution, this layer provides a level of abstraction for the hardware and contains essential hardware drivers. Note that Android runs on a slightly modified version of Linux, to take into account the limitations of mobile devices, such as a memory management system more aggressive in preserving memory than usual, or the Binder Inter-Process Communication (IPC) driver, a crucial part of the operating system which allow the exchange of messages between processes.

2.1.2 Android Compatibility Program

The openness of the Android source code makes it possible for any manufacturer to ship a custom version of the OS. Google does not forbid this behavior, and in fact encourages it through its Android Compatibility Program [Andg], which sets the requirements that the modified OS must fulfill in order to remain compatible with standard Android apps, regardless of the modifications introduced. The requirements are listed in the Compatibility Definition Document

Table 2.1: Test suites for Android devices compatibility and certification [Mst]

Test suite	Purpose	Approval for compatibility	Approval for certification
CTS — Compatibility Test Suite [Bgc]	Ensuring compatibility with AOSP	●	●
GTS — Google mobile services requirements Test Suite	Requirements for any devices that want to pre-install Google suite of apps	○	●
VTS — Vendor Test Suite [Bgv]	Tests if partner devices are compatible with the hardware abstraction layer	○	●
BTS — Build Test Suite	Security review for potentially harmful behaviors in binaries or the framework	○	●
STS — Security Test Suite	Verification of the correct app of security patches	○	●

(CDD), which enumerates the modifications that can be made to the OS while ensuring compatibility with other devices. Vendors who wish to ship a modified version of the Android OS can use the Compatibility Test Suite (CTS), a set of open-source tests to ensure that their modified version remains compatible with AOSP [Bgc].

In addition to the Android compatibility program, Google created a certification program for phone manufacturers, that not only ensures compatibility but also gives a number of security guarantees. Internally, Google uses the CTS along with other test suites for this certificate, as listed in Table 2.1. Unfortunately, not all of these test suites are open-source, and Google does not provide technical details as to what is actually tested. Moreover, there is no test to verify the level of privacy of devices. In reality, regardless of which tests are actually conducted, these test suites are simply not enough to prevent privacy and security issues, as we will show in detail in the rest of this thesis.

Devices made by vendors that are part of the Android Certified Partners program [Andb] come pre-loaded with Google's suite of apps (e.g., the Play Store and Youtube). Companies that want to include the Google Play service without passing the certification themselves can outsource the design of their product to a certified Original Design Manufacturer (ODM) [Cer]. Such devices can still be certified, as long as they pass the same test suite.

2.1.3 System Updates and FOTA Apps

Android system partitions are read-only to prevent tampering by an outside party. The Android framework includes mechanisms to update the system and system apps, which are traditionally handled by device vendors. However, many vendors have not been able to ship critical

security patches and provide support to new Android releases at a reasonable pace, resulting in a substantial number of outdated and unpatched Android versions [MN21]. Google has recently put forward two initiatives to alleviate the problems for vendors to adapt their code to AOSP and improve the distribution of updates. Project Treble [Andx], announced in 2017, tries to help vendors to build their own Android version from a new AOSP release by separating customized vendor software (provided by silicon manufacturers and other vendor-specific suppliers) from the core Android OS framework. This was implemented through a new vendor interface and a Vendor Test Suite (VTS) [Bgv] to ensure forward compatibility. In addition, Project Mainline [Andv], launched in 2019 on top of Project Treble, allows updating core OS components through Google Play, similar to app updates. This allows Google to deliver critical security updates directly on user devices without intervention from the manufacturer.

The main motivation behind these two projects is to push critical security updates and other enhancements faster, without requiring the manufacturer to integrate the changes in their code-base and ship an OTA update. These updates are implemented through a new file format introduced in Android 10 called Android Pony EXpress (APEX) [Apea]. Its installation involves Android's package manager, as APEX packages share the same structure as APKs. Once the package manager recognizes the file format, it makes the APEX manager apply the update after a reboot [Apeb]. These mechanisms improve devices' security by reducing the time it takes to push an update. However, only phone vendors certified by Google [Mst; Rsa] can implement this update mechanism at the time of this writing [Andv].

In practice, system updates are managed by a FOTA app, regardless of whether or not the device makes use of projects Treble or Mainline. Such apps are highly privileged and play a critical role in maintaining devices secured and updated. The Android operating system offers standard mechanisms–available to OEMs–to implement their own FOTA apps but such vendor-specific implementations could be a source of security and privacy issues due to poor software engineering practices. We discuss more in detail issues related to FOTA apps in Chapter 5 (page 47).

2.2 The Android Permission System

In this section, we present in depth the many features of the Android permissions system, relying on both the official documentation and AOSP source code. The Android operating system runs each app in a sandbox to prevent unwanted interaction between apps. Every app runs with its own user ID (UID) and group ID (GID), and the kernel puts additional restrictions in place to limit access to some system resources. Some of these resources are protected by

permissions (e.g., the GPS of the phone, or access to the user's SMS). By default, apps do not have access to permission-protected resources, in an effort to secure the device and protect the user's privacy [Gplb]. Apps must *request* permission from the user to be able to interact with the protected resource.

2.2.1 Requesting a Permission

There are multiple ways for an app to request a permission. The most common one is by using the `<uses-permission>` tag in the app's manifest, along with the permission name [Gplb]. System apps can also use the `<adopt-permissions>` tag, along with the package name of the app they wish to adopt the permissions from. Both apps must be signed by the same certificate. This feature was introduced as a way to migrate data from one app to another when a new app replaces an older app [Ado].

2.2.2 Permission Enforcement

Each Android app has a unique UID assigned to it at installation time. When the app runs, its UID and GID are set to that UID. Permissions are mapped to (Linux) GIDs in Android. The permission-to-GID mapping for built-in permissions is located in `/etc/permission/`, in the `platform.xml` file. When an app is granted a permission, the permission's GID is assigned as a supplementary GID to the app process by the package manager. The kernel and system services use the UID, GID, and supplementary GIDs of the app process to determine whether it has access to standard system objects and functions, including regular files, devices, and local sockets. Note that the system does not make a distinction between the app and the third-party libraries it embeds. Therefore, any embedded SDK can take advantage of the permissions granted to the app, and the resources said permissions protect.

Shared User IDs. Apps that are signed with the same key can also add the `sharedUserId` attribute in their manifest, which will cause the system to run both apps under the same UID. This maps to the same Linux UID at runtime, so the apps can run in the same process and access the same system resources, even if only one of the apps requests the permission. Shared UIDs are objects managed by the Android package manager, an OS component that provides APIs to query installed packages, or permissions [Aosa]. Each shared UID has an associated list of permissions, which is the union of the permissions requested by all currently installed packages that have this UID. Hence, apps with the same shared UID inherit that superset of permissions, allowing one app access to protected APIs for which it has not requested the appropriate permission if another app with the same shared UID has already requested and

obtained this permission. The package manager dynamically adds and removes permissions from the shared UID object as packages are installed and uninstalled. This feature was deprecated in API level 29 and may be removed from future versions of Android, but is currently still available for apps to use [Sha].

2.2.3 Protection Levels

Android permissions each have a protection level that characterizes the potential risk of the permission. Each protection level consists of one mandatory *base permission type* associated with zero or more *flags* [Andj]. We define these flags in details in Section 6.2.1. There are three base permission types: *normal*, *dangerous*, and *signature*.

Normal permissions

These permissions protect resources that do not pose a significant risk to the user's privacy or the operation of other apps (e.g., `INTERNET` which is mandatory to get access to the Internet). If an app requests one such permission, the system automatically grants access to it at install time. The system does not prompt the user to grant normal permissions, and users cannot revoke them.

Dangerous permissions

These permissions protect resources and types of personal data that could harm the user's privacy (e.g., the location) or could affect the user's stored data or the operation of other apps. Whenever an app requires access to a dangerous permission, the user has to explicitly grant the permission to the app, either all at once at install time for devices running Android at a version older than 6.0, or one by one via a system dialog otherwise. Until the user approves the permission, the requesting app cannot use the associated functionality. Moreover, if the device is running Android 6.0 or higher, the user can later withdraw its approval in the app's profile in the device's settings.

Signature permissions

These permissions are the most restricted ones. To get access to such permissions, an app must meet any one of the following conditions [Gra]: (i) share the same signing certificates as the defining app; (ii) be signed with a certificate that was rotated from the defining app; or (iii) be signed with a certificate that used to sign the defining app, and that is still trusted by it.

If the app meets any of these conditions with the system package, then the permission will also be granted. An app can also, in some cases, use the system service associated with the protected component. For instance, the permission `BIND_VPN_SERVICE` has a signature protection level, but an app can still integrate a VPN by using the system's `VpnService` [Vpn]. Some of the signature permission cannot be used by third-party apps (e.g., `DUMP` which allows an app to get state dump information from system services [Dum]).

Internal permissions

The upcoming version of Android, Android 13, will see the arrival of a new protection level called *internal* permissions [Coml]. Those permissions will work in a very similar way as signature permissions but will be tightly tied to the notion of *role* in permissions [Comk]. Internal permissions will not be granted based on signatures but rather based on the roles of the requesting and defining apps, thus allowing the creation of role-only permissions.

2.2.4 Permission Groups

Permission groups are categories to organize sets of related permissions according to the device's features they refer to [Pera]. For example, the SMS group includes the `READ_SMS` and the `RECEIVE_SMS` permissions. Permission requests are handled at the group level, even if each single permission definition appears in the manifest.

In Android 6.0 (API level 23) and higher, when an app requests for the first time a dangerous permission, the OS checks its group. If no permission from that group is already granted to the app, the user is shown a dialog that shows the group name and description but not the specific permission. If the app already has a dangerous permission in the same group, the request is granted automatically. Note that the app must still explicitly request the permission. In Android 5.1 (API level 22) and lower, the same process takes place at install time. For instance, an app requesting the permission to read contacts (`READ_CONTACTS`) will show a dialog with the contacts permission group instead of the permission. However, if approved by the user, only the `READ_CONTACTS` permission will be granted to the app, not the other permissions from the permission group.

2.2.5 Permission Trees

Permission trees are namespaces used to define the base name (prefix) for a tree of permissions [Perc]. The app defining a permission tree owns all names that belong to the tree and can add new permissions to that tree either statically in the manifest or programmatically by

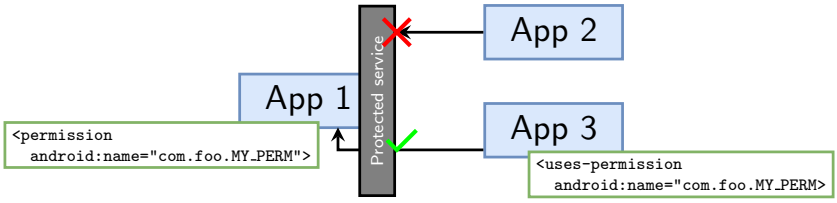


Figure 3: Example of an app defining a custom permission and protecting a service with it. Only app_3, which requests the permission, can interact with the service exposed by app_1.

calling `PackageManager.addPermission()`, and remove them with `removePermission()`. A permission added dynamically is added to the package database and, therefore, persists across reboots until removed by the app.

2.2.6 Custom Permissions

One overlooked feature of Android's permission model is its *extensibility*. By design, the Android framework allows any app developer to share features implemented in their software with other apps either explicitly using the `android:exported` attribute in the component definition, or implicitly by setting one or more `Intent-Filter`. Apps can manage access to such components in a *"controlled"* way by defining *custom permissions* [Gpla].

Apps can define custom permissions in their manifest file by using the `<permission>` tag and protect a component with them by using the `android:permission` in the component definition [Cusa]. Apps requesting access to custom permissions must also do so in their manifest using the `<uses-permission>`, just like they do for regular AOSP permissions. Once an app requests access to a custom permission, it can interact with the protected component, for instance by sending an intent [Int] or by instantiating the component directly (e.g., for a protected activity). By default, access to custom permissions is regulated by the Android package manager, but the app defining them can implement further access controls to only grant access to authorized apps, regardless of the protection level of the permission, by calling `checkPermission`, `enforcePermission`, or one of their variants [Andt]. Apps can also define custom permissions groups by using the `<permission-group>` tag in their manifest by specifying a name and a description, among other fields [Perb]. A permission can then use its group with the `permissionGroup` attribute in its definition.

Figure 3 shows an example of communication between apps regulated by a custom permissions. In this example, app_1 defines a permission called `com.foo. MY_PERM` to protect a service. At first, app_2 tries to interact with the protected service by sending an intent but does not hold the required permission: the intent is rejected. App_3 on the other hand does hold the `com.foo.MY_PERM`: when it sends an intent to the protected service, the intent is accepted.

At the framework level, there are two types of custom permission enforcement. *Static permission enforcement* applies when an app (the caller) interacts with a component defined by another app (the callee) and that component has defined which permissions the caller process must possess. In this case, the `ActivityManagerService` resolves the calling intent and determines the permission associated with the target component. The package manager will then check if the caller does hold that permission. *Dynamic permission enforcement* occurs when a component does not delegate permission checks to the system but instead does it itself programmatically, using a number of helper methods in the `Context` class, most notably `checkPermission` or `enforcePermission`.

Naming conventions for custom permissions. The Android operating system does not impose any restrictions on custom permission names or the features and data they can enable. However, Google recommends using the app's package name as the prefix for the custom permissions that it defines (e.g., an app with the package name `com.foo` should name their custom permissions `com.foo.MY_PERM`), which itself should use a "reverse-domain-style name", to ensure package and permission name uniqueness [Gpla]. Google also recommends adding a description of the purpose of their permissions to "explain the permission to the user" [Andab] when defined in the Android Manifest file. However, no active policy enforcement seems to be applied in the case of pre-installed apps, as we show in Chapter 5. Note that embedded Software Development Kits (SDK) embedded on Android apps may also define custom permissions with the collaboration of the developer, in which case the permissions they request or define will be merged in the manifest of the host app [Mer] (e.g., an SDK from `sdk.com` could define the `com.sdk.MY_PERM` permission). The presence of SDK-defined custom permissions adds another layer of complexity to their analysis. We will investigate in depth such custom permissions later in this thesis, first from an attribution perspective (Chapter 5, page 47), then studying their privacy and security implications (Chapter 7, page 93).

Chapter 3

Related Work

"Science is made up of so many things that appear obvious after they are explained"

— Frank Herbert, *Dune* (1965)

Android, being the most used mobile operating system in existence, has attracted the attention of the research community. Since its first release, there has been a significant number of studies, both theoretical and practical, that measure, analyze, and break down various aspects of Android. In this chapter, we present previous studies of relevance to the contributions we make in this thesis. We first discuss pre-installed apps and supply chain issues (§3.1). Then, we present the seminal papers that examine the Android permissions system (§3.2), a domain that has long been the focus of the research community. We conclude this chapter with a presentation of state-of-the-art tools in the area of Android app analysis techniques (§3.3).

3.1 Studying and Characterizing the Android Supply Chain

The research community has spent some efforts on quantifying the effects of OEMs customization on the security and privacy guarantees of the OS. The initial main focus of previous work was on image customization [Gra+12; Wu+13; ZSL14; Aaf+15; Zho+14; PWL20a], while a few other works focused on discovering vulnerabilities present in pre-loaded apps themselves [Tia+18; Wu+19]. We note that since we published our first paper on pre-installed apps [Gam+20], there has been a renewed interest on studying the privacy and security implications of such apps [Els+20] and vendor customizations [Ji+21; Pos+21; Lee+21; Lei21; LPL21; Lei22]

3.1.1 Android Images Customization

Grace et al. investigated the quality of the implementation of the permission system in eight different devices [Gra+12]. Specifically, the authors focused on what they called "capability leaks", i.e., situations where an untrusted app (e.g., user installed) can leverage unprotected components exposed by privileged pre-installed apps to access permission-protected APIs without actually holding said permission. They develop WoodPecker, a system to detect such cases by building the app's control flow graph and looking for possible paths in that graph from the unprotected components. Using their system on 8 popular devices at the time, they uncovered capability leaks in all of them.

Zheng et al. studied the presence of malware in Android firmware images [ZSL14]. The authors presented DroidRay, a system that relies on both static and dynamic analysis to evaluate the security of a given firmware image. Specifically, they rely on signature analysis and Virus Total [Vir] to detect malware in pre-installed apps, and privilege escalation vulnerabilities at the framework level. They use DroidRay on 250 Android firmware images, containing 24k pre-installed apps, and discovered that 8% of said images contain pre-installed malware samples. More worrying, they found that virtually all of the images they analyzed (99.6%) were vulnerable to at least one version of the MasterKey bug [Jayb; Jaya; Jayc].

Among the papers that are part of the new effort to investigate vendor customizations, Ji et al. presented the first analysis of init routines in Android firmware images [Ji+21]. Init on Android (and Linux) is the process that initializes the user space and can execute custom routines defined by stakeholders of the supply chain. Such custom routines can also be triggered by pre-installed apps. The authors presented DefInit, a tool to detect and analyze apps that expose init routines, and found 89 instances of insufficiently protected custom routines. The authors show that some of these routines could be exploited by other apps on the device, and could therefore lead to privilege escalation vulnerabilities.

Possemato et al. focused on OEMs customization of Android images and presented a framework to measure their level of compliance with regard to the requirements set out by Google in the CDD [Pos+21]. Specifically, the authors do not focus on pre-installed apps themselves, but rather look at modifications of security-enhanced Linux (SELinux) policies, init scripts, and assess the security of the kernel by extracting its version and hardening options. The authors then use their framework on 2,907 Android images they crawled from manufacturers' websites. The authors have found that about 20% of the images they analyzed break at least one of the rules of the CDD, including images created for Google-certified devices, which calls into question the efficiency of the certification itself.

3.1.2 Privacy and Security of Pre-installed Apps

Tian et al. investigated the use of custom AT commands in modern Android devices across 11 vendors [Tia+18]. They found that some of these commands enable very powerful functionalities, including unlocking the screen, performing screen touch events, or even rewriting the device firmware, which represents a broad attack surface, even for modern Android devices. The authors found that such commands could be issued using the USB connection of the device. However, the authors did not discuss whether AT commands could also be issued by apps installed on the device (either system or user-installed apps), which would make the attack surface even broader.

Elsabagh et al. presented FirmScope, a static analysis tool that aims at discovering privilege escalation vulnerabilities in Android firmware images [Els+20]. Their tool extracts APKs from an image and then applies taint analysis to these apps to detect the hidden features we highlighted in our paper on pre-installed Android apps [Gam+20]. The authors managed to find 850 unique privilege escalation vulnerabilities by analyzing over 330K apps. In some cases, such vulnerabilities are caused by components exposed by pre-installed apps, which are not properly protected using a permission. Those vulnerabilities can be therefore triggered by other apps installed on the device. Google and affected phone manufacturers recognized the severity of such vulnerabilities, and the authors received 147 CVEs after responsible disclosure of their results. Unfortunately, they did not release the source code of their tool, making it impossible to reproduce their work or to build upon it.

Lau et al. released Uraniborg, a framework to quantify the privacy and security risks posed by the apps pre-loaded on a phone [Lau+20]. They take into account the permissions requested by or pre-granted to pre-loaded apps, or the number of apps that are signed by the same certificate as the platform. However, the authors fail to take into account the role of pre-loaded apps. For instance, any app that requests the `INSTALL_PACKAGES` permission will increase the risk score significantly; however, it is perfectly normal for a device to have at least one app able to install other apps (e.g., the Google Play Store), which will increase the score even though that behavior is expected.

Finally, recent papers discuss the data collection that occurs by default on Android devices. Leith et al. [Lei21] compares the network traffic of a stock Pixel device sent by default and compares it to the traffic sent by an iPhone under the same conditions. They find unique, non-resettable user IDs sent over the wire, despite explicitly opting out of any data collection. In another paper, Liu et al. [LPL21] conduct a similar analysis, this time comparing the telemetry sent by default by six variants of the Android OS, from major phone manufactur-

ers (e.g., Samsung, Xiaomi) but also known user-developed images (e.g., LineageOS). They find substantial sharing of PII even when the device stays in an idle state, not only back to the manufacturer but also to third parties. Finally, Leith et al. [Lei22] conduct a study of the default Google message and dialer app that comes pre-installed on handsets from various manufacturers. The data collected could theoretically allow for linking devices after a phone call, and otherwise uniquely identify users and the relationship between said users.

Overall, these papers confirm some of the findings of this thesis: in Chapter 5, we present evidence of pre-installed apps contacting domains associated with well-known advertising and tracking services. However, the results of these papers are limited in scale, due to the difficulty to conduct dynamic analysis of system apps in an emulated environment. For their experiments, the authors instrumented real devices, but this prevented them from expanding their dataset to have a more exhaustive vision of the issues at hand. This also reflects the difficulty of using dynamic analysis techniques to conduct privacy analysis of pre-install apps, as we will further discuss in the next section.

3.2 The Android Permission System

The Android permission system has long been the focus of the academic community. There have been significant efforts to study the use and abuse of AOSP permissions [SC13; Au+12b]. Prior studies analyzed over-privileged apps [Fel+11b; VCC11; Chi+17; Leo+12; Ped+19] and assessed the efficiency and transparency of Android's permission model to empower users [Fel+12; FGW11; KGC13; Wij+15]. Others identified vulnerabilities in the permission system that allowed developers to get access to protected APIs, either by exploiting weaknesses of the permission system [NKZ10; Fel+11c; Bug+11; Gib+12; SBM15; Sad+18] or by exploiting side and covert channels to circumvent it [Mar+12; AHIN14; Des14; MBN14; Mic+15; SXA16; Spr+17; BNN17].

3.2.1 Characterization of the Permission System

In 2011, Felt et al. presented Stowaway, the first tool to determine if all permissions requested by a given app are actually used, based on dynamic analysis [Fel+11b]. The authors ran Stowaway on 900 Android apps and found that around 35% of them asked for unnecessary permissions (i.e., they were not used on the app's code). They showed that often over-privilege is due to developer errors (e.g., legacy code, or copied and pasted code). Au et al. used a different approach and introduce PScout, a static analysis tool to infer the specification of the permission system from Android 2.2 to Android 4 [Au+12b]. Their main objective was to determine if,

given the large number of permissions offered by the OS (79 at the time of publication), there was any overlap in the set of protected APIs for a given pair of permissions, and found only one such pair. The authors also noted the presence of undocumented APIs and permissions, as we highlight in Chapter 6 (page 83), but showed that such APIs are rarely used by apps.

Backes et al. built a static runtime model of the Android permission framework [Bac+16] to (1) build a more complete and more recent mapping of API calls to permissions, and (2) to study permission locality (i.e., whether permissions are enforced only by one particular service). The authors showed that 20% of the analyzed permissions are checked by more than one single class, making enforcement of permissions a more complex task.

Some studies focus on how efficient the Android permission system is at conveying information to the user about the permissions their apps use. Fealt et al. analyzed how effective install time permissions are at warning users about the potential risks of Android apps [Fel+12]. To do so, they relied on an Internet survey and a laboratory experiment in which they monitor users. They showed that below 20% of users paid attention to permissions during app installation, and only 24% of users showed a clear understanding of permissions. However, most participants claimed that they had decided not to install an app based on its permissions at least once. Wijesekera et al. performed a 36-person study on how often apps accessed protected information that the users were not expecting [Wij+15]. To do so, they instrumented the Android OS to learn when protected resources are accessed during the use of an app and, through exit interviews, found that 80% of participants would have liked to stop access to a given piece of data at least once. In general, users noted that they would have liked to stop a third of all data access.

Finally, Zhauniarovich et al. reviewed the implementation and temporal evolution of the Android permission system [ZG16]. This paper was published in 2016 after Android 6.0 was released, and therefore after the introduction of runtime permissions in Android, which was a major modification to the way permissions are enforced in Android. This was the first paper to give a complete overview of the permission and protection level flags, and the security and privacy implications. The permission system has however significantly evolved since the publication of this paper: we give an updated view of the permission system and its evolution ourselves in Chapter 6 (page 83).

3.2.2 Security and Privacy

Android permissions have been typically used as a proxy by the research community. To infer the privacy and security risks of apps. However, this method presents some limitations, as the

mere fact that an app requests a permission does not mean it will access the protected resource during its execution.

Johnson et al. focused on over-privileged apps, i.e., apps that request permissions that they do not make use of in their code. They analyzed over 140k apps for cases of over privilege [Joh+12]. The authors showed that around 54% of apps requested extra permissions. Interestingly, they also showed that 50% of these apps were also missing permissions that would be needed to make use of some of the APIs present in their code, hinting at developer errors rather than maliciousness. The authors also released a mapping between APIs and the permissions protecting them. To tackle this issue, Google introduced in 2019 a new system to the Google Play Store in which developers receive a warning if their app is requesting a permission that is often not requested by similar apps. Peddinti et al. reported that this approach is relatively useful, as 59% of the apps that received a warning removed the permission [Ped+19]. However, it suffers from serious limitations. For instance, if the majority of calculator apps request the `ACCESS_FINE_LOCATION` permission, then the system will assume that permission is needed for calculator apps when it clearly is not.

Sarma et al. proposed a system in which users are informed of whether the risk of installing an app is greater than its benefits by taking into account the permissions requested by the app, the app category, and the permissions requested by apps in the same category [Sar+12]. They tested their approach using apps from two datasets, one comprising 121 malware samples and another one with around 158k apps gathered from the marketplace. Similarly, Jeon et al. remarked that, while there is a large number of fine-grained permissions in Android, a permission may provide larger access than actually needed by an app [Jeo+12]. The authors therefore created a taxonomy of four major permission groups that encompass all existing permissions (e.g., sensors permissions, which include for instance `ACCESS_FINE_LOCATION` or `CAMERA`), along with strategies to infer new fine-grained permissions variants for each group, and therefore increase user security by only giving an app access to the APIs it needs. The authors also released several tools to implement their strategy, without having to change the Android OS.

Developers can make a conscious choice to request more permissions when releasing a new version of their apps but might want to avoid asking users for extra permissions. In such cases, they can make use of permissions groups' behavior. Calciati et al. investigated such permissions groups, i.e., AOSP permissions that are grouped by the OS since they relate to the same features (e.g., `RECEIVE_SMS` and `READ_SMS`, both in the `SMS` group) [Cal+20]. If an app is granted one such permission, and the app requests more permissions from the same group in

an update, the OS will automatically and silently grant all new requested permissions that are in the same group as already granted permissions. The authors investigated the prevalence and privacy risks of such a mechanism.

Another issue might be caused by embedded third-party libraries, which can leverage any permission requested to their host app. Liu et al. presented Pedal, a system to prevent third-party libraries to take advantage of the permissions granted to host app [Liu+15]. This system separates code from Third-Party Library (TPLs) and the host app, and blocks calls to protected APIs from the libraries by default. However, their detection and separation system relies on machine learning algorithms, which sometimes yield false positives and false negatives, which can negatively impact user experience.

Third-party libraries were also the focus of a more recent study by Feal et al. [Fea+21]. Specifically, the authors investigate the privacy risks due to piggybacking of privilege by embedded SDKs. Any third-party library embedded in an app inherits all the permissions that were granted to the host app. The authors evaluate the efficiency of SDK detection and auditing techniques and find that most tools do not provide accurate results in general, which highlights the need for more robust and accurate methods for SDK detection and analysis.

Finally, some developers try to simply bypass the permission system. In their paper, Reardon et al. focused on circumvention of the permission system using covert- or side-channels (e.g., getting the MAC address of the device without holding the otherwise required permission by calling `ioctl`) [Rea+19]. The authors ran 88,000 apps in an instrumented environment, where they log system calls at the kernel level, then exploited these logs to find evidence of circumvention of the permission system. After identifying circumvention techniques in an app, they statically checked the rest of their dataset of apps to discover other apps with the capability to use said technique.

3.2.3 Custom Permissions

Previous studies mostly focused on AOSP permissions, and custom permissions have been largely overlooked in the literature, with some notable exceptions.

Tuncay et al. highlighted several attacks on the official permission system by leveraging custom permissions [Tun+18]. They described a custom permission upgrade attack that exploits the permission groups to be able to enable any dangerous permission without user awareness and approval. They also discussed a confused deputy attack that exploits the lack of enforcement on naming conventions to access signature custom permissions with an app that is not signed with the same certificate as the defining app. Both attacks were acknowledged

and fixed by Google. However, the naming issue remains as there is still a lack of enforcement of naming conventions.

Bagheri et al. [Bag+15; Bag+18] showed the issues inherent to Android's permission model by creating a formal model of Android's permission protocol for automatically analyzing and verifying it. The authors showed that the lack of naming conventions for custom permissions allowed an attacker app to get access to a component protected by a custom permission in the victim app (in a way akin to the confused deputy attack described by Tuncay et al.). They also analyzed a subset of real-world Android apps to confirm their findings.

Finally, Li et al. showed how custom permissions can be used to gain access to APIs otherwise protected by AOSP permissions [Li+21]. The authors develop CuPerFuzzer, an automatic fuzzing tool that they use against the Android OS. This tool allowed them to discover four design shortcomings of the permissions system, which were reported to Google and fixed by the Android security team in Android 10. However, some of these attacks need user interaction multiple times to be carried out, which renders them less threatening in practice.

3.3 Android App Analysis Techniques

A decade of research in Android yielded numerous different tools and frameworks to analyze apps, either using static analysis such as CHEX [Lu+12], FlowDroid [Arz+14], or Amandroid [Wei+14] among others [Gib+12; Li+15; FCF09; Kim+12; Qia+15; Gor+15; YY12; Li+14; Kli+14]. Some other tools use dynamic analysis instead [Bug+12; Qia+14; He+19; Tam+15], TaintDroid [Enc+14] and DroidScope [YY12] in particular, while some use a combination of both static and dynamic analysis, such SMV-Hunter [GK14] or QUIRE [Die+11]. Other studies focus on Dalvik bytecode analysis [VRH98; JMF12; Bar+12; Wog+14], or present tools that are created for specific kinds of analysis, such as intents or other Inter-Component Communication (ICC) mechanisms [SR14; Jin+16; Bha+17; KZM17], or native code usage in Android apps [Wei+18; Sto18]. In this section, we focus only on flow analysis tools. Such tools are typically used for security and privacy analysis of Android apps, as they allow researchers to track pieces of data, either statically on during the execution of the app, and thus are suitable for privacy analysis [You+15; Avd+15].

3.3.1 Static Analysis

Static analysis is a popular choice of technique for researchers. A systematic literature review from 2017 found 124 papers using static analysis techniques alone [Li+17]. Another similar survey from 2021 found as many as 261 [Aut+21].

FlowDroid is a taint analysis tool that is context-, flow-, field-, object-sensitive, and life cycle-aware [Arz+14]. It models the life cycle of Android components, which rely on implicit method calls and asynchronous events (e.g., the user can choose to go back to the previous activity at any point during the execution). FlowDroid relies on a list of sources (i.e., methods that return sensitive information) and sinks (i.e., methods that leak data out of the app) to conduct its analysis. However, FlowDroid suffers from limitations, the least of which being its computational costs that can be needed to analyze some large Android apps. Previous studies reported needing as much as 730GB of RAM and a 64 CPU server for 24 hours to analyze a single app, even when configuring FlowDroid in such a way that would sacrifice some accuracy for better performance, which makes FlowDroid unsuitable for large scale studies [Avd+15]. Another major drawback is its inability to handle ICC or Inter-App Communication (IAC) since FlowDroid only conducts inter-procedural data flow analysis. This makes FlowDroid poorly adapted to study custom permissions which rely heavily on such mechanisms.

IccTA tries to overcome this last limitation [Li+15]. In their paper, the authors modified the intermediate representation of the code to add links to account for inter-component messages. They then use FlowDroid to conduct the actual taint analysis, but because of this, they also inherit the performance issues that plague FlowDroid. Amandroid (now renamed Argus-SAF) was developed around the same time as IccTA [Wei+14]. Amandroid is a framework for security vetting of Android apps. As IccTA, it can conduct taint analysis even between components or between apps.

3.3.2 Dynamic Analysis

TaintDroid is a dynamic taint analysis tool built on top of the Android OS [Enc+14]. It allows for tracking of data as the app runs and is therefore not susceptible to the limitations of static analysis tools, such as obfuscation, dynamically loaded classes, or the use of the Java reflection API [Ref]. However, using TaintDroid requires flashing a custom-built image on a device, and the last version supported is Android 4.3, released in 2013, which means it is unable to take into account subsequent developments in the Android OS (e.g., runtime permissions, which were introduced in Android 6). Moreover, it makes it difficult at best to use for the analysis of pre-installed apps, as such apps might rely on modification of the Android OS added by a stakeholder of the supply chain. Such modifications would not be present on TaintDroid which is built on top of code from AOSP.

NDroid is another taint analysis framework based on dynamic analysis [Qia+14]. NDroid focuses on another part of the Android stack: the JNI. Using the JNI, an app can call code from a

shared library either located in its assets or, in the case of system apps in particular, pre-loaded on the device by a stakeholder of the supply chain. Other frameworks, either based on static or dynamic analysis, can miss leaks due to such calls, as they only limit themselves to analyzing the app's bytecode. NDroid adds a module to track information flows that go through the JNI engine and relies on TaintDroid for taint analysis. This means that NDroid inherits the limitations of TaintDroid that we detailed in the previous paragraph. In 2018, the authors of the original NDroid paper released an updated version, along with an updated version of the code that supports up to Android 7 [Xue+18].

While dynamic analysis is useful to report on the actual behavior of apps, as opposed to potential behavior in the case of static analysis, a major drawback is the exercising of the app. A dynamic analysis tool can only analyze the code that is being run (e.g., the code associated with the activity currently displayed); it is therefore needed to generate events that allow for a full exploration of the app, or the analysis could miss some potentially dangerous behaviors. There are different approaches to event generation, from random input (i.e., clicking at random coordinates on the screen) [Andai], or model-based, in which the generator first build a representation of the UI to find interesting elements to test [MTN13; Ama+15; Su+17; SQH17].

3.3.3 Limitations for the Analysis of System Apps

System apps bring another set of difficulties for static or dynamic analysis tools. Such apps are different by nature: while the developers of apps available on app stores cannot make any assumption as to the environment their app will run on (e.g., presence of a specific library, or Android version), system apps developers can. They can therefore make use of specific features of the OS, or code from a shared object loaded alongside their app, or even split their app into multiple, smaller ones that will communicate with one another, which can be facilitated by the use of the `sharedUserId` attribute. Such behaviors can make like difficult for researchers, or even render some state-of-the-art tools unable to spot any malicious behaviors.

Another specificity of system apps is the use of external Optimized Dalvik Executable File (ODEX) files. Android APK usually contains at least one `classes.dex` file, which contains the Java bytecode converted into Smali, which uses the Dalvik Executable File (DEX) bytecode. However, system apps can instead use an external ODEX file stored alongside it on the filesystem, which can be loaded ahead of time by the OS, leading to performance gains (see Chapter 2.1, page 15, for details). This also can hinder static or dynamic analysis tools. Such tools expect the DEX file(s) to be present in the app; in the case of external ODEX files, such tools will simply fail to analyze the app. As part of our work on FOTA apps [Blá+21], we cre-

ated and released a tool, Dextripador [Dex], that can convert back an ODEX file into a DEX, that can then be loaded alongside the APK, thus allowing for static or dynamic analysis.

II

On the Impact of Customization on Users' Privacy and Security

Chapter 4

Collecting Pre-installed Apps at Scale

"*Lex Murphy: It's a UNIX system, I know this!*"

— Steven Spielberg, *Jurassic Park* (1993)

Obtaining pre-installed apps and other software artifacts (e.g., certificates installed in the system root store, or shared libraries) at scale is challenging. Such files are usually not publicly available on app stores, which might be missing apps potentially installed by other elements in the supply chain such as resellers, or by FOTA components offering third-party installations similar to Pay-per-Install (PPI) programs. They must therefore be collected from the firmware directly.

The majority of previous studies tackle this problem by crawling firmware images from manufacturers' websites or specialized websites [Wu+13; ZSL14; Aaf+15; Zho+14; Tia+18], or by extracting them from real devices in a lab environment [Gra+12]. However, these approaches suffer from serious limitations. First, using real devices has obvious scaling issues, as purchasing enough mobile handset models, and their many variations, to cover a significant portion of the market is unfeasible. Second, crawled firmware images would only contain a subset of pre-loaded apps and would miss most, if not all, of system apps installed afterward by either other actors of the supply chain or through FOTA components who might install extra apps automatically at first boot.[1] We decided to use a different approach and crowdsource the collection of pre-installed software using a purpose-built app: Firmware Scanner. In the remainder of this chapter, we first explain the design of our app (§4.1), and present statistics from the dataset we collected (§4.2). We conclude this chapter by discussing the ethical implications of our data collection (§4.3).

[1] This mechanism is otherwise called out-of-the-box experience (OOBE).

Table 4.1: List of data collected by Firmware Scanner. A * in a location denote a subfolder, i.e., a potential location in all existing system partitions

Data	Possible location(s)
Pre-installed apps	*/app, */priv-app
ODEX files	*/app, */priv-app (Stored alongside APKs files)
Native libraries	*/lib, */lib-64
Root certificates	/etc/security/cacerts/
Permission allowlists	/etc/permissions/*.xml, /etc/default-permissions/*.xml, /etc/sysconfig/*.xml
Framework APK	/system/app/framework-res.apk, /system/priv-app/framework-res.apk, /data/system-framework/framework-res.apk, /system/vendor/overlay/framework-res.apk, /system/vendor/overlay-subdir/framework-res.apk, /system/vendor/overlay-subdir/pg/framework-res.apk, /system/product/overlay/framework-res.apk, /vendor/overlay/framework-res.apk, /vendor/overlay/PG/android-framework-runtime-resource-overlay.apk, /data/resource-cache/system@vendor@overlay@framework-res.apk@idmap, /product/overlay/framework-res.apk

4.1 Firmware Scanner

Publicly available on Google Play [Sca], Firmware Scanner is a purpose-built Android app to extract pre-installed apps and other binaries. It scans the following system partitions: /system, /odm, /oem, /vendor, and /product. These partitions are among the official ones and were chosen as they can potentially contain system apps [Off]. Firmware Scanner also collect any native libraries, ODEX files, and root certificates that come pre-installed in these partitions. In addition, Firmware Scanner scan and collect files from /etc/default-permissions/, /etc/permissions/, and /etc/sysconfig/: these folders contain lists of pre-granted permissions. Finally, the app tries to collect as well the framework app. Note that the actual location of these files might differ, depending on the level of customization added by the vendor. Firmware Scanner only checks the default locations. Table 4.1 lists the different pieces of data collected by Firmware Scanner.

4.1.1 Workflow

Figure 4 summarizes the workflow of Firmware Scanner and Figures 5a to 5e show screenshots of the app in action. Firmware Scanner starts by asking the user for their consent (Figure 5a).

No data is uploaded before the app receives explicit consent from the user. We discuss more in depth the ethical concerns of our data collection in Section 4.3.

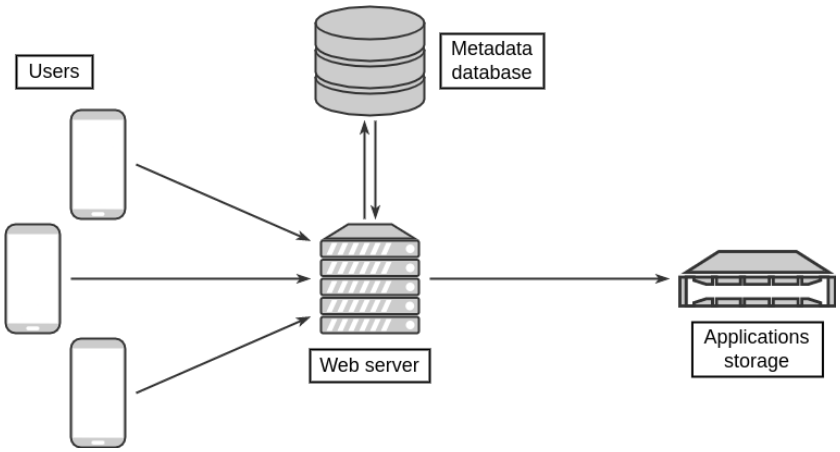


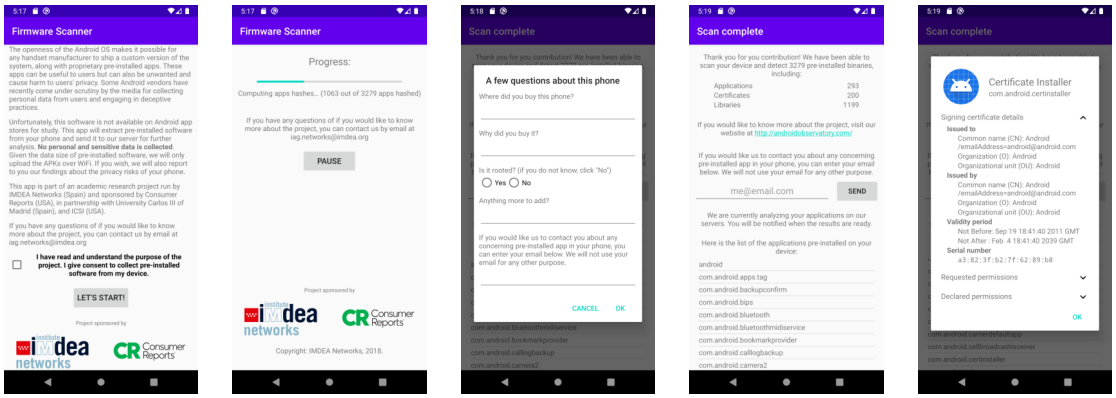
Figure 4: Workflow of Firmware Scanner's operating

After obtaining user consent, Firmware Scanner scans the system partitions and computes the MD5 hash of each file (Figure 5b). It then sends the list of hashes to our server, which will compare it against our metadata database, and return only the hashes of files to upload. The goal of this server-side comparison is twofold: first, to reduce the upload time, and therefore reduce the risk of users canceling the operation; and second, to reduce the total size of the data that has to be uploaded. In addition, Firmware Scanner only uploads files over a Wi-Fi connection to avoid affecting the user's data plan.

At this stage, Firmware Scanner will also upload some metadata about the device, specifically: the device's manufacturer name, its model and name, its build fingerprint, the version of Android it is running, its current timezone, and the MCC-MNC codes and country code of its SIM card (if any). We use this data to help us attribute pre-installed binaries back to a stakeholder of the supply chain. For instance, we can conclude with reasonable confidence that an app signed by a certificate mentioning `Samsung` in the subject is indeed coming from Samsung if the device was manufactured by Samsung.

Firmware Scanner also tries to detect if the device it is running on is rooted. On a rooted device, the user can remount a system partition as read-write and then install apps on it. Therefore, on a rooted device, we cannot be certain that an app located on a system partition was not manually installed by the user, instead of by an actor of the supply chain of the device. We consider that a given device is rooted according to two signals. First, after Firmware Scanner has finished the upload of pre-installed binaries, it asks the user directly if the handset is rooted according to their understanding (note that the user may choose not to answer the question). As a complement, we use the library RootBeer [Roob] to programmatically check if

the device is rooted or not. If any of these sources indicate that the device is potentially rooted, we consider it as such. Firmware Scanner also asks some questions to the user, which we use as a complement to the metadata automatically gathered (Figure 5c).



(a) Asking for user consent (b) Computing hashes on device (c) User questions about their device (d) Listing of files on the device (e) Example of app details

Figure 5: Screenshots from Firmware Scanner in operation on a device

Finally, the app uploads those files to the server for storage and later analysis. Firmware Scanner displays the number of files that were detected on the device, along with the list of system apps (Figure 5d). The user then has the possibility of clicking on any app to get more details, such as the information contained in the certificate used to sign the app and the list of defined and requested permissions (Figure 5e).

4.2 Data Collected

Table 4.2: Dataset collected by Firmware Scanner as of the 6th of May, 2022

		All devices	Non-rooted only
Number of ...	apps	1,309,968	983,875
	users	120,132	91,509
	vendors	1,050	651
	devices	33,915	26,142
Percentage of users in ...	Europe	23.8%	24.3%
	Asia	40.1%	38.8%
	Americas	25.3%	26.6%
	Africa	8%	8.3%
	Others	2.7%	2%

The initial version of Firmware Scanner was released in July 2018 on Google Play. At the time of this writing, the data collection is still ongoing. Table 4.2 summarizes our dataset as of the 11th of February, 2022. Our dataset covers as many as 26,142 unique device models from 651 phone manufacturers (excluding rooted devices). In addition, Figure 6 shows the percentage

of users of Firmware Scanner per country (this figure was generated using the country code of the SIM card of the device if such a card is present). This gives us an unprecedented view into the ecosystem of pre-installed Android apps, which would not have been possible to achieve by other means. Indeed, our approach allows us to collect apps that might only be available in certain regions of the world, or certain countries, which might in turn allow us to uncover more regional stakeholders of the supply chain of Android devices.

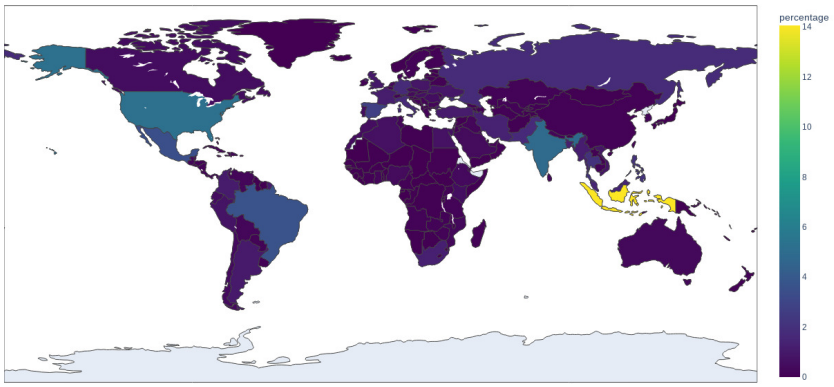


Figure 6: Percentage of users per country as of the 11th of February, 2022

4.3 Ethical Aspects

Our data collection depends on real users who organically installed Firmware Scanner on their devices. Therefore, we follow the principles of informed consent [DK12] and we avoid the collection of any personal or sensitive data. We sought the approval of our institutional Ethics Board and Data Protection Officer (DPO) before starting the data collection. The app also provides extensive privacy policies in their Google Play profile.

The app collects some metadata about the device to attribute observations to manufacturers (e.g., its model and fingerprint) along with some data about the pre-installed apps (extracted from the Package Manager), MNO, and user (the timezone, and the MCC and MNC codes from their SIM card, if available). To minimize the risks for users, we generate a unique UID for each device to identify duplicates and updated firmware versions for a given device.

Chapter 5

Pre-installed Apps in Android Devices

"*The electric things have their life too. Paltry as those lives are*"

— Philip K. Dick, *Do Androids Dream of Electric Sheep?* (1968)

The open source nature of the Android OS makes it possible for manufacturers to ship custom versions of the OS along with a set of pre-installed apps, often for product differentiation. Yet, the landscape of pre-installed software in Android has largely remained unexplored, particularly in terms of the security and privacy implications of such customizations, as we highlight in Chapter 3 (page 27). In this chapter, we present the first large-scale study of pre-installed software on Android devices from more than 200 vendors, relying on data collect with Firmware Scanner. We answer questions related to the stakeholders involved in the supply chain, from device manufacturers and mobile network operators to third-party organizations like advertising and tracking services, and social network platforms (§5.2). We then do a preliminary analysis of custom permissions in pre-installed apps (§5.3), attributing them to responsible parties, and study the behavior of these apps using both static and dynamic analysis (§5.4). Finally, we take a deep dive on a specific category of pre-installed apps to highlight the risks that come with add extra system apps: apps with the ability to read the full, unfiltered system logs (5.5).

5.1 Data Sources

In this chapter, we use a subset of the data collected by Firmware Scanner (see Chapter 4, page 41 for details) which contains pre-installed software from 1,742 device models. Table 5.1 summarizes the dataset we consider. This dataset contains 424,584 unique files (based on their

Vendor	Country	Certified partner	Device Fingerprints	Users	Files (med.)	Apps (med.)	Libs (med.)	DEX (med.)	Root certs (med.)	Files (total)	Apps (total)
Samsung	South Korea	Yes	441	924	868	136	556	83	150	260,187	29,466
Huawei	China	Yes	343	716	1,084	68	766	96	146	150,405	12,401
LGE	South Korea	Yes	74	154	675	84	385	89	150	58,273	3,596
Alps Mobile	China	No	65	136	632	56	385	46	148	29,288	2,883
Motorola	US/China	Yes	50	110	801	127	454	62	151	28,291	2,158
Xiaomi	China	Yes	49	104	1,506	98	1,114	92	156	61,004	2,876
Lenovo	China	Yes	45	92	602	48	385	46	150	30,132	2,050
Asus	Taiwan	Yes	43	88	689	92	423	60	156	35,237	2,165
ZTE	China	Yes	36	80	943	79	580	84	161	27,418	1,702
Sony	Japan	Yes	32	68	1,062	180	723	62	148	31,687	2,757
Total (214 vendors)	–	**22%**	**1,742**	**2,748**						**424,584**	**82,501**

Table 5.1: General statistics for the top-10 vendors in our dataset.

MD5 hash) as shown in Figure 7 for selected vendors. For each device we plot three dots, one for each type of file, while the shape indicates the major Android version that the device is running.[1] The number of pre-installed files varies greatly from one vendor to another. Although it is not surprising to see a large amount of drivers and native libraries due to hardware differences, some vendors embed hundreds of extra apps compared to other manufacturers running the same Android version. For instance, some Alps Mobile and LGE devices come with less than 50 pre-installed apps, while we found multiple Samsung devices that ship more than 350. The trend is the same for libraries. The number of root certificates stays similar across vendors for the same version of Android (the number of root certificates varies from an Android version to another), with some notable exceptions. For the rest of our study, we focus on 82,501 Android apps present in this dataset, leaving the analysis of root certificates and libraries for future work.

We complement this dataset with crowdsourced traffic logs collected using the Lumen Privacy Monitor, an app that aims to promote mobile transparency and enable user control over their mobile traffic [Raz+15; Lum] to obtain anonymized network flow metadata from real users. This allows us to correlate the information we extract from static analysis, for a subset of mobile apps, with realistic network traffic generated by mobile users in the wild and captured in user-space. In the remainder of this section, we explain the method implemented by Lumen and discuss the ethical implications of this data collection.

5.1.1 Lumen Privacy Monitor

Lumen is an Android app that was available on Google Play up until September 2021 and aims to promote mobile transparency and enable user control over their personal data and traffic. It leverages the Android VPN permission to intercept and analyze all Android traffic in user-

[1] We found that 5,244 of the apps do not have any activity, service, or receiver. These apps may potentially be used as providers of resources (e.g., images, fonts) for other apps.

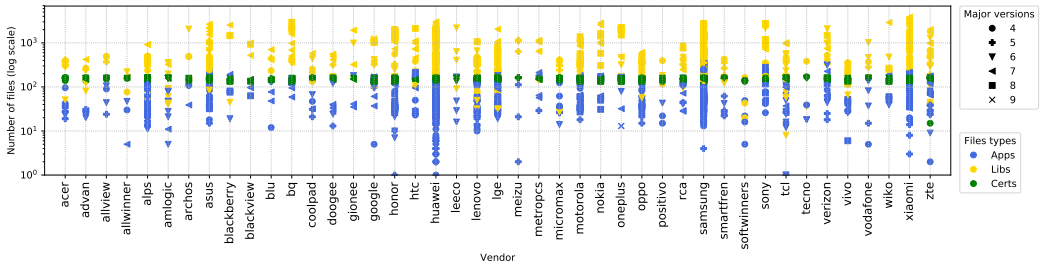


Figure 7: Number of files per vendor. We do not display the vendors for which we have less than 3 devices.

space and in-situ, even if encrypted, without needing root permissions. By running locally on the user's device, Lumen is able to correlate traffic flows with system-level information and app activity. Lumen's architecture is publicly available and described in [Raz+15]. Lumen allows us to accurately determine which app is responsible for an observed PII leak from the vantage point of the user and as triggered by real user and device stimuli in the wild. Since all the analysis occurs on the device, only processed traffic metadata is exfiltrated from the device, and no personal data and traffic payloads is collected.

For this study, we use anonymized traffic logs provided by over 20.4K users from 144 countries (according to Google Play Store statistics) coming from Android phones manufactured by 291 vendors. This includes 34,553,193 traffic flows from 139,665 unique apps (298,412 unique package name and version combinations). However, as Lumen does not collect app fingerprints or hashes of files, to find the overlap between the Lumen dataset and the pre-installed apps, we match records sharing the same package name, app version, and device vendor as the ones in the pre-installed apps dataset. While this method does not guarantee that the overlapping apps are exactly the same, it is safe to assume that phones that are not rooted are not shipped with different apps under the same package names and app versions. As a result, we have 1,055 unique pre-installed app/version/vendor combinations present in both datasets.

Ethical Concerns

Our study involves the collection of data from real users who organically installed Lumen on their devices. Therefore, we follow the principles of informed consent [DK12] and we avoid the collection of any personal or sensitive data. We sought the approval of our Intitutional Review Board (IRB) and Data Protection Officer (DPO) before starting the data collection. Lumen also provided extensive privacy policies in its Google Play profile, up until its removal from the platform in September, 2021.

Users are required to opt in twice before initiating traffic interception [DK12]. Lumen

preserves its users' privacy by performing flow processing and analysis on the device, only sending anonymized flow metadata for research purposes. Lumen does not send back any unique identifiers, device fingerprints, or raw traffic captures. To further protect user's privacy, Lumen also ignores all flows generated by browser apps which may potentially deanonymize a user; and allows the user to disable traffic interception at any time.

5.2 Supply Chain Analysis

The openness of Android OS has enabled a complex supply chain ecosystem formed by different stakeholders, be it manufacturers, MNOs, affiliated developers, and distributors. These actors can add proprietary apps and features to Android devices, seeking to provide a better user experience, add value to their products, or provide access to proprietary services. However, this could also be for (mutual) financial gain [Fac; New]. This section provides an overview of pre-installed Android packages to uncover some of the gray areas that surround them, the large and diverse set of developers involved, the presence of third-party advertising and tracking libraries, and the role of each stakeholder.

5.2.1 Developer Ecosystem

We start our study by analyzing the organizations signing each pre-installed app. First, we cluster apps by the unique certificates used to sign them and then we rely on the information present in the `Issuer` field of the certificate to identify the organization [Sig]. Despite the fact that this is the most reliable signal to identify the organization signing the software, it is still noisy as a company can use multiple certificates, one for each organizational unit. More importantly, these are self-signed certificates, which significantly lowers the trust that can be put on them.

We were unable to identify the company behind several certificates (denoted as *Unknown company* in Table 5.2) due to insufficient or dubious information in the certificate: e.g., the `Issuer` field only contains the mentions `Company` and `department`. We have come across apps that are signed by 42 different *"Android Debug"* certificates on phones from 21 different brands. This reflects poor and potentially insecure development practices as Android's debug certificate is used to automatically sign apps in development environments, hence enabling other apps signed with that certificate to access its functionality without requesting any permission. Most app stores (including Google Play) will not accept the publication of an app signed with a Debug certificate [Andi]. Furthermore, we also found as many as 115 certificates that only mention *"Android"* in the `Issuer` field. A large part (43%) of those certificates are supposedly

Company name	Number of certificates	Country	Certified partner?
Google	92	United States	N/A
Motorola	65	US/China	Yes
Asus	60	Taiwan	Yes
Samsung	38	South Korea	Yes
Huawei	29	China	Yes
TCL Corporation	23	China	Yes
Lenovo	20	China	Yes
LG Electronics	18	South Korea	Yes
Sony Corporation	18	Japan	Yes
ZTE Corporation	16	China	Yes
Total (vendors)	**740**	—	—

Company name	Number of certificates	Country	Number of vendors
MediaTek	19	China	17
Aeon	12	China	3
Tinno Mobile	11	China	6
Verizon Wireless	10	United States	5
Neffos (TP-Link)	7	China	1
Unknown company	7	China	1
Wingtech	5	China	2
Huaqin	5	China	5
Zhantang	4	China	14
Longcheer	4	China	1
Total	**460**	—	**214**

Table 5.2: **Left:** top-10 most frequent developers (as per the total number of apps signed by them), and **right:** for other companies.

issued in the US, while others seem to have been issued in Taiwan (16%), China (13%), and Switzerland (13%). In the absence of a public list of official developer certificates, it is not possible to verify their authenticity or know their owner, as discussed in Section 5.6.

With this in mind, we extracted 1,200 unique signing certificates out of our dataset. Table 5.2 shows the 5 most present companies in the case of phone vendors (left) and other development companies (right). Besides vendor certificates, Google certificates is a notable exception, although it is not surprising to see Google certificates, the company being at the origin of the Android project. 7 of the 10 most present companies are ODM companies or chipset vendors but also large telecommunication companies such as Verizon Wireless. This analysis uncovered a vast landscape of third-party software in the long-tail, including large digital companies (e.g., inkedIn, Spotify, and TripAdvisor), as well as advertising and tracking services. This is the case of ironSource, an advertising firm signing pre-installed software [Iroa] found in Asus, Wiko and other vendors, and TrueCaller, a service to block unwanted call or texts [Truc]. According to their website and also independent sources [Trub; Trud], TrueCaller uses crowdsourced mechanisms to build a large dataset of phone numbers used for spam and also for advertising. This can have nefarious consequences for end users' privacy, as highlighted by Privacy International [Trua]. Likewise, we have found 123 apps (by their MD5) signed by Facebook. These apps are found in 939 devices, 68% of which are Samsung's. We have also found apps signed by AccuWeather, a weather service previously found collecting personal data aggressively [Ren+18], Adups software, responsible for the Adups backdoor [Krya], and GMobi [Gmob], a mobile-advertising company previously accused of dubious practices by the Wall Street Journal [New].

Category	# libraries	# apps	# vendors	Example
Advertisement	164 (107)	11,935	164	Braze
Mobile analytics	100 (54)	6,935	158	Apptentive
Social networks	70 (20)	6,652	157	Twitter
All categories	**334**	**25,333**	**165**	—

Table 5.3: Selected third-party libraries categories present in pre-installed apps. In brackets, we report the number of TPLs when grouped by package name.

5.2.2 Third-party Services

As in the web, mobile app developers can embed in their pre-installed software third-party libraries (TPLs) provided by other companies, including libraries (SDKs) provided by ad networks, analytics services or social networks. In this section we use LibRadar++, an obfuscation-resilient tool to identify TPLs used in Android apps [Wan+18a], on our dataset to examine their presence due to the potential privacy implications for users: when present in pre-installed apps, TPLs have the capacity to monitor user's activities longitudinally [VR+12; Raz+18]. We exclude well-known TPLs providing development support such as the Android support library. First, we classify the 11,665 unique TPLs identified by LibRadar++ according to the categories reported by Li et al. [Li+16], AppBrain [Appg], and PrivacyGrade [Pria]. We manually classified those TPLs that were not categorized by these datasets.

We focus on categories that could cause harm to the users' privacy, such as mobile analytics and targeted advertisement libraries. We find 334 TPLs in such categories, as summarized in Table 5.3. We could identify advertising and tracking companies such as Smaato (specialized in geo-targeted ads [Sma]), GMobi, Appnext, ironSource, Crashlytics, and Flurry. Some of these third-party providers were also found shipping their own packages in Section 5.2.1 or are prominent actors across apps published in Google Play Store [Raz+18]. We found 806 apps embedding Facebook's Graph SDK which is distributed over 748 devices. The certificates of these apps suggests that 293 of them were signed by the device vendor, and 30 by an operator (only 98 are signed by Facebook itself). The presence of Facebook's SDKs in pre-installed apps could, in some cases, be explained by partnerships established by Facebook with Android vendors as the New York Times revealed [Fac].

We found other companies that provide mobile analytics and app monetization schemes such as Umeng, Fyber (previously Heyzap), and Kochava [Raz+18]. We also found instances of advanced analytics companies in Asus handsets such as Appsee [Appc] and Estimote [Est]. According to their website, Appsee is a TPL that allows developers to record and upload the users' screen [Appb], including touch events [Pan+18]. If, by itself, recording the user's screen

does not constitute a privacy leak, recording and uploading this data could unintentionally leak private information such as account details. Estimote develops solutions for indoors geo-localization [Est]. Estimote's SDK allows an app to react to nearby wireless beacons to, for example, send personalized push notifications to the user upon entering a shop

Finally, we find TPLs provided by companies specialized in the Chinese market [Wan+18a] in 548 pre-installed apps. The most relevant ones are Tencent's SDK, AliPay (a payment service) and Baidu SDK [Baid] (for advertising and geolocation / geo-coding services), the last two possibly used as replacements for Google Pay and Maps in the Chinese market, respectively. Only one of the apps embedding these SDKs is signed by the actual third-party service provider, which indicates that their presence in pre-installed apps is likely due to the app developers' design decisions.

5.2.3 Public and Non-public Apps

We crawled the Google Play Store to identify how many of the pre-installed apps found by Firmware Scanner are available to the public. This analysis took place on the 19th of November, 2018 and we only used the package name of the pre-installed apps as a parameter. We found that only 9% of the package names in our dataset are indexed in the Google Play Store. For those indexed, few categories dominate the spectrum of pre-installed apps according to Google Play metadata, notably communication, entertainment, productivity, tools, and multimedia apps.

The low presence of pre-installed apps in the store suggests that this type of software might have escaped any scrutiny by the research community. In fact, we have found samples of pre-installed apps developed by prominent organizations that are not publicly available on Google Play. For instance, software developed and signed by Facebook (`com.facebook.appmanager` for instance), Amazon, and CleanMaster among others. Likewise, we found non-publicly available versions of popular web browsers (e.g., UME Browser, Opera).

Note that it is possible that these apps were not pre-loaded by a stakeholder of the supply chain, but might instead be installed on a system partition later on by a FOTA app. To verify this hypothesis, we performed the first large-scale and systematic analysis of the FOTA ecosystem through a dataset of 2,013 FOTA apps detected with a tool that we designed specifically for this purpose over 422,121 pre-installed apps. We classified the different stakeholders developing and deploying FOTA apps and showed that 43% of FOTA apps are in fact developed by third parties, and not by the phone vendor itself. Moreover, we reported that some devices can have as many as 5 apps implementing FOTA capabilities, from different categories of stakeholders.

By means of static analysis of the code of FOTA apps, we showed that some of these apps present behaviors that can be considered privacy intrusive, such as the collection of sensitive user data (e.g., geolocation linked to unique hardware identifiers), and a significant presence of third-party trackers. We also discovered implementation issues leading to critical vulnerabilities, such as the use of public AOSP test keys both for signing FOTA apps and for update verification, thus allowing any update signed with the same key to be installed. Finally, we used telemetry data collected from real devices by a NortonLifeLock and demonstrated that some FOTA apps are also responsible for the installation of non-system apps (e.g., entertainment apps and games), including malware and Potentially Unwanted Programs (PUP).

Looking at the last update information reported by Android's package manager for these apps, we found that pre-installed apps also present on Google Play are updated more often than the rest of pre-installed apps: 74% of the non-public apps do not seem to get updated and 41% of them remained unpatched for 5 years or more. If a vulnerability exists in one of these apps (see Section 5.4), the user may stay at risk for as long as they keep using the device.

5.3 Permission Analysis

Android permissions are not limited to those defined by AOSP: any app developer – including manufacturers – can define their own *custom permissions* to expose their functionality to other apps [Cusb]. We leverage Androguard [Anda] to extract the permissions, both defined and requested, by pre-installed apps. We primarily focus on the analysis of custom permissions as (i) pre-installed services have privileged access to system resources, and (ii) these services may (involuntarily) expose critical services and data, even bypassing Android's official permission set. In this section, we study in particular the actors of the supply chain that are behind these custom permissions. We will later complement this analysis in Chapter 7 by studying the prevalence overall of custom permissions and characterize their purpose by means of code analysis.

5.3.1 Defined Custom Permissions

We identify 1,795 unique Android package names across 108 Android vendors defining 4,845 custom permissions. We exclude AOSP–defined permissions, as well as those associated with Google Cloud Messaging (GCM) [Gcma]. The number of custom permissions defined per Android vendor varies across brands and models due to the actions of other stakeholders in the supply chain. We classify the organizations defining custom permissions in 8 groups as shown in Table 5.4: hardware vendors, MNOs (e.g., Verizon), third-party services (e.g., Facebook),

	Custom	Providers							
	permissions	Vendor	Third-party	MNO	Chipset	AV / Security	Ind. Alliance	Browser	Other
Total	4,845 (108)	3,760 (37)	192 (34)	195 (15)	67 (63)	46 (13)	29 (44)	7 (6)	549 (75)
Android Modules									
android	494 (21)	410 (9)	–	12 (2)	4 (13)	–	6 (7)	–	62 (17)
com.android.systemui	90 (15)	67 (11)	1 (2)	–	–	–	–	–	22 (8)
com.android.settings	87 (16)	63 (12)	–	1 (1)	–	–	–	–	23 (8)
com.android.phone	84 (14)	56 (9)	–	5 (2)	3 (5)	–	–	–	20 (10)
com.android.mms	59 (11)	35 (10)	–	1 (2)	–	–	1 (1)	–	22 (8)
com.android.contacts	40 (7)	32 (3)	–	–	–	–	–	–	8 (5)
com.android.calendar	33 (10)	24 (6)	–	–	–	–	–	–	9 (6)
com.android.email	33 (10)	18 (4)	–	–	–	–	–	–	15 (17)
com.android.gallery3d	29 (9)	27 (8)	–	–	–	–	2 (1)	–	–
com.android.nfc	28 (16)	21 (7)	–	–	–	–	3 (13)	–	4 (4)

Table 5.4: Summary of custom permissions per provider category and their presence in selected sensitive Android core modules. The value in brackets reports the number of Android vendors in which custom permissions were found.

Anti-Virus (AV) firms (e.g., Avast), industry alliances (e.g., GSM Association (GSMA)), chipset manufacturers (e.g., Qualcomm), and browsers (e.g., Mozilla). We could not confidently identify the organizations responsible for 9% of all the custom permissions.[2]

As shown in Table 5.4, 63% of all defined custom permissions are defined by 31 handset vendors according to our classification. Most of them are associated with proprietary services such as Mobile Device Management (MDM) solutions for enterprise customers. Yet three vendors account for over 68% of the total custom permissions; namely Samsung (41%), Huawei (20%), and Sony (formerly Sony-Ericsson, 7%). Most of the custom permissions added by hardware vendors–along with chipset manufacturers, and MNOs–are exposed by Android core services, including the default browser `com.android.browser`. Tables 5.6 and 5.6 show some examples of such custom permissions. Unfortunately, as demonstrated in the MediaTek case [Fel+11c], exposing such sensitive resources in critical services may potentially increase the attack surface if not implemented carefully.

[2]While Android's documentation recommends using reverse-domain-style naming for defining custom permissions to avoid collisions. [Cusb], 269 of them – many of which are defined by a single hardware vendor – start with AOSP prefixes such as `android.permission.*`. The absence of good development practices among developers complicated this classification, forcing us to follow a semi-manual process that involved analyzing multiple signals (the permission name, package name of the defining app, and signing certificate) to identify their possible purpose and for attribution.

MANUFACTURER PERMISSIONS			
Package name	**Developer Signature**	**Vendor(s)**	**Permission**
com.sec.android.app.sns3	Android SW 2 Group (KR)	Samsung	*.permission.SNS_LINKEDIN_API
com.htc.linkedin	Android (TW)	HTC	*.permission.useprovider
com.android.calendar	Sony Ericsson (SE)	Sony	*.linkedin.SYNC
com.sonyericsson.facebook.proxylogin	Sony Ericsson (SE)	Sony	com.sonyericsson.permission.FACEBOOK
com.sonymobile.twitter.account	Sony Ericsson (SE)	Sony	com.sonymobile.permission.TWITTER
android	Sony Ericsson (SE)	Sony	com.sonymobile.googleanalyticsproxy.permission.GOOGLE_ANALYTICS
com.htc.socialnetwork.facebook	Android (TW)	HTC	*.permission.SYSTEM_USE
com.sonymobile.gmailreaderservice	Sony Ericsson (SE)	Sony	com.sonymobile.permission.READ_GMAIL
com.sec.android.daemonapp	Samsung Corporation (KR)	Samsung	*.ap.accuweather.ACCUWEATHER_DAEMON_ACCESS_PROVIDER
com.sec.enterprise.knox.cloudmdm.smdms	Samsung (KR)	Samsung	*.permission.SAMSUNG_MDM_SERVICE
android	Lenovo (CN)	Lenovo	android.permission.LENOVO_MDM
com.asus.loguploaderproxy	AsusTek (TW)	Asus	asus.permission.MOVELOGS
com.miui.core	Xiaomi (CN)	Xiaomi	miui.permission.DUMP_CACHED_LOG
android	Samsung (KR)	Samsung	com.sec.enterprise.knox.KNOX_GENERIC_VPN
com.sec.enterprise.permissions	Samsung (KR)	Samsung	android.permission.sec.MDM_ENTERPRISE_VPN_SOLUTION
com.android.vpndialogs	Meizu (CN)	Meizu	com.meizu.permission.CONTROL_VPN
com.android.browser	Samsung (KR)	Samsung	com.sec.android.app.browser.permission.BOOKMARK

MNO PERMISSIONS			
Package name	**Developer Signature**	**MNO**	**Permission**
com.android.mms	ZTE	T-Mobile US	com.tmobile.comm.RECEIVE_METRICS
android	Motorola	T-Mobile US	com.tmobile.comm.RECEIVE_METRICS
com.lge.ipservice	LG	T-Mobile US	com.tmobile.comm.RECEIVE_METRICS
hr.infinum.mojvip	Infinum (HR) [Inf]	H1 Croatia	hr.infinum.mojvip.permission.RECEIVE_ADM_MESSAGE
com.locationlabs.cni.att	AT&T (US)	AT&T (US) [Loc]	com.locationlabs.cni.att.permission.BROADCAST
com.asurion.android.verizon.vms	Asurion (US) [Asua]	Verizon (US)	com.asurion.android.verizon.vms.permission.C2D_MESSAGE
com.smithmicro.netwise.director.cricket	Smith Micro (US) [Smi]	Cricket (US)	com.smithmicro.netwise.director.cricket.MND_AUTOMATION
jp.naver.line.android	Naver (JP)	South Korea Telekom	com.skt.aom.permission.AOM_RECEIVE

Table 5.5: Examples of custom permissions from manufacturers and MNOs. The wildcard * represents the package name whenever the permission prefix and the package name overlap.

THIRD-PARTY SERVICE PERMISSIONS			
Package name	**Developer Signature**	**Provider**	**Permission**
com.facebook.system	Facebook	Facebook	*.ACCESS
com.facebook.appmanager	Facebook	Facebook	*.ACCESS
com.amazon.mShop.android.shopping	Amazon	Amazon	com.amazon.client.metrics.nexus.permission.TRIGGER_UPLOAD
com.amazon.kindle	Amazon	Amazon	com.amazon.identity.auth.device.perm.AUTH_SDK
com.huawei.android.totemweather	Huawei (CN)	Baidu	android.permission.BAIDU_LOCATION_SERVICE
com.jrdcom.usercard	TCLMobile (CN)	Baidu	android.permission.BAIDU_LOCATION_SERVICE
com.oppo.findmyphone	Oppo (CN)	Baidu	android.permission.BAIDU_LOCATION_SERVICE
com.android.camera	YuLong (CN)	Baidu	android.permission.BAIDU_LOCATION_SERVICE
com.dti.sliide	Logia	Digital Turbine	com.digitalturbine.ignite.ACCESS_LOG
com.dti.att	Logia	Digital Turbine	com.dti.att.permission.APP_EVENTS
com.ironsource.appcloud.oobe.wiko	ironSource	ironSource	com.ironsource.aura.permission.C2D_MESSAGE
com.vcast.mediamanager	Verizon (US)	Synchronoss	com.synchronoss.android.sync.provider.FULL_PERMISSION
com.myvodafone.android	Vodafone (GR)	Exus	uk.co.exus.permission.C2D_MESSAGE
com.trendmicro.freetmms.gmobi	TrendMicro (TW)	GMobi	com.trendmicro.androidmup.ACCESS_TMMSMU_REMOTE_SERVICE
com.skype.rover	Skype (GB)	Skype	com.skype.android.permission.READ_CONTACTS
com.cleanmaster.sdk	Samsung (KR)	CleanMaster	com.cleanmaster.permission.sdk.clean
com.netflix.partner.activation	Netflix (US)	Netflix	*.permission.CHANNEL_ID

CHIPSET PERMISSIONS			
Package name	**Developer Signature**	**Provider**	**Permission**
com.qualcomm.location	ZTE (CN)	Qualcomm	com.qualcomm.permission.IZAT
com.qualcomm.location	Xiaomi (CN)	Qualcomm	com.qualcomm.permission.IZAT
com.qualcomm.location	Sony (SE)	Qualcomm	com.qualcomm.permission.IZAT
com.mediatek.mtklogger	TCL (CN)	MediaTek	com.permission.MTKLOGGER
com.android.bluetooth	Samsung (KR)	Broadcom	broadcom.permission.BLUETOOTH_MAP

Table 5.6: Examples of custom permissions from third-party services and chipsets manufacturers. The wildcard * represents the package name whenever the permission prefix and the package name overlap.

An exhaustive analysis of custom permissions and of the apps exposing them also suggests (and in some cases confirms) the presence of service integration and commercial partnerships between handset vendors, MNOs, analytics services (e.g., Baidu, ironSource, and Digital Turbine), and online services (e.g., Skype, LinkedIn, Spotify, CleanMaster, and Dropbox). We also found custom permissions associated with vulnerable modules (e.g., MediaTek) and potentially harmful services (e.g., Adups). We discuss cases of interest below.

MDM solutions: Samsung, Lenovo and LG incorporate Mobile Device Management (MDM) solutions for the remote administration and monitoring of mobile devices, mainly for enterprise customers. We have identified proprietary MDM-related permissions in Samsung [Samc], LG [Lgb], and Lenovo [Len] devices. MDM software is over-privileged by definition and allows collecting fine-grained telemetry about the device and user. Further, pre-installed packages in Xiaomi, Samsung, Asus and Sony handsets expose custom permissions to read and upload to the cloud system logs and telemetry, While there may be legitimate usages for these permissions, hence potentially opening new privacy risks to users if not implemented carefully.

We could also identify custom MDM permissions associated with companies such as AetherPal [Aet], and MediaTek's log reporting service (e.g., `com.permission.MTKLOGGER`) LogMeIn, and RSupport in several vendors. In some cases, the software exposing these permissions is signed by MNOs and hardware vendors. There are interesting differences across these solutions. The pre-installed app `net.aetherpal.device` and `com.rsupport.rsperm` are signed by the Android manufacturers directly. LogMeIn, instead, defines custom permissions for two MNOs (e.g., `com.lmi.vodafone.*`) and two hardware vendors (e.g., `com.lmi.htc.*`). but the packages are always signed by LogMeIn.

VPN solutions: Android provides native support to third-party VPN clients. This feature is considered as highly sensitive as it gives any app requesting access the capacity to break Android's sandboxing and monitor users' traffic [Andaj; Ikr+16]. The analysis of custom permissions reveals that Samsung and Meizu implement their own VPN service. It is unclear why these proprietary VPN implementations exist but it has been reported as problematic by VPN developers for whom their clients, designed for Android's default VPN service, do not run on such handsets [Adg; Raz+15; Ikr+16]. A complete analysis of these VPN packages is left for future work.

MNO permissions Mobile Network Operators (MNOs) may customize the firmware of the Android devices that they offer to their customers. We identify 195 permissions associated with 24 different MNOs in 15 vendors. Through these custom permissions, MNOs may add value and expose proprietary services to over-the-top developers, a and partners for control-

Package	Public	# Vendors	# Permissions
com.facebook.system	No	18	2
com.facebook.appmanager	No	15	4
com.facebook.katana (Facebook)	Yes	14	8
com.facebook.orca (Messenger)	Yes	5	5
com.facebook.lite (FB Lite)	Yes	1	1
com.facebook.pages.app	No	1	4
Total	3	24	18

Table 5.7: Facebook packages on pre-installed handsets.

ling and accessing users' data plan, or configuring system resources. These practices are common among European, North American and large Asian mobile carriers. However, The custom permissions defined by AT&T, South Korea Telekom, Verizon Wireless, Sprint and Vodafone account together for 68% of all the MNO-defined permissions. As opposed to vendor-defined permissions, MNO ones are exposed by over-the-top APKs as shown in Table 5.4. The majority of MNO permissions are exposed by apps developed by third-party developers but there are instances of packages signed by handset manufacturers. This suggest the presence of commercial agreements between manufacturers and MNOs as part of the supply chain. as in the case of T-Mobile's custom permission com.tmobile.comm.RECEIVE_METRICS which is defined in packages signed by Motorola, Samsung, LG, ZTE, and HTC. Interestingly, in these cases, the APK is signed by the hardware vendor. This suggests a close collaboration between hardware manufacturers and MNOs at the time of customizing their subsidized devices.

MNO-preinstalled modules may also give MNOs the capacity to harvest fine-grained user and system telemetry unavailable at the network-level. Several custom permissions are potentially the result of MNOs outsourcing software development to other companies such as Infinum.com [Inf] (H1 Croatia), LocationLabs [Loc][3] Asurion [Asua] (AT&T and Verizon), and SmithMicro (Cricket) [Smi].[4] Nevertheless, the majority of them are exposed by apps developed by third-party developers as shown in Table 5.5.

Facebook: We found 6 different Facebook packages, three of them unavailable on Google Play, defining 18 custom permissions as shown in Table 5.7. These permissions have been found in 24 Android vendors, including Samsung, Asus, Xiaomi, HTC, Sony, and LG. According to users' complaints, two of these packages (com.facebook.appmanager and com.facebook.system) seem to automatically download other Facebook software in users' phones [Andak; Xda], such as Instagram. We also found interactions between Facebook and MNOs such as Sprint.

Baidu: Baidu's geo-location permission is exposed by pre-installed apps, including core An-

[3] A subsidiary company of the security firm Avast which defines an AT&T-specific permission.

[4] According to their website, they have partnerships with cellular service providers such as Verizon Wireless, AT&T, and Sprint Nextel.

droid modules, in 7 different vendors, mainly Chinese ones. This permission seems to be associated with Baidu's geocoding API [Baic] and could allow app developers to circumvent Android's location permission.

Digital Turbine: We have identified 8 custom permissions in 8 vendors associated with Digital Turbine and its subsidiary LogiaGroup. Their privacy policy indicates that they collect personal data ranging from unique identifiers (UIDs) to traffic logs that could be shared with their business partners, which are undisclosed [Dig]. According to the SIM information of these devices, Digital Turbine modules are mainly found in North-American and Asian users. One package name, `com.dti.att` ("dti" stands for Digital Turbine Ignite), suggests the presence of a partnership with AT&T. A manual analysis confirms that this is the case. By inspecting their source-code, this package seems to implement comprehensive software management service. Installations and removals of apps by users are tracked and linked with PII, which only seem to be "masked" (i.e., hashed) discretionally.

ironSource: The advertising company ironSource exposes custom permissions related to its AURA Enterprise Solutions [Irob]. We have identified several vendor-specific packages exposing custom ironSource permissions, in devices made by vendors such as Asus, Wiko, and HTC (the package name and certificate signatures suggest that those modules are possibly introduced with vendor's collaboration). According to ironSource's material [Iroc], AURA has access to over 800 million users per month, while gaining access to advanced analytics services and to pre-load software on customers' devices. A superficial analysis of some of these packages (e.g., `com.ironsource.appcloud.oobe.htc` or `com.ironsource.appcloud.oobe.asus`) reveals that they provide vendor-specific out-of-the-box-experience apps (OOBE) to customize a given user's device when the users open their device for the first time and empower user engagement [Irob], while also monitoring users' activities.

Other Advertising and Tracking Services: Discussing every custom permission introduced by third-party services individually would require an analysis beyond the scope of this chapter. However, there are a couple of anecdotes of interest that we discuss next. One is the case of a pre-installed app signed by Vodafone (Greece) and present in a Samsung device that exposes a custom permission associated with Exus [Exu], a firm specialized in credit risk management and banking solutions. Another service defining custom permissions in Samsung and LG handsets (likely sold by Verizon) is the analytics and user engagement company Synchronoss. Its privacy policy acknowledges the collection, processing and sharing of personal data [Syn].

Chipset Manufacturers and Industry Alliances: Hardware manufacturers such as Qualcomm, Broadcom, Mediatek, and Wingtech also develop pre-installed software that defines

custom permissions. Due to the nature of these providers, this software is broadly distributed across Android handset vendors. Interestingly, in the majority of cases these permissions are not exposed by critical Android services as shown in Table 5.4. This suggests that Chipset manufacturers typically distribute their own APKs across Android vendors. Some of these permissions are associated with A-GPS and location services [VR+13] (e.g., `com.qualcomm.location`), wireless technologies such as Bluetooth, FOTA services, and network interface management. We also identified 29 permissions associated with Industry alliances such as GSMA, FIDO, Mirrorlink [Mir], ANT+ [Ant], or the SIM Alliance.

These efforts sometimes trigger innovation and standardization efforts in the form of SDKs and solutions available to its members. These initiatives can be also observed when studying custom permissions: we have identified 29 custom permissions associated with industry alliances in 44. As in the case of chipset and vendor-defined permissions, those falling in this category can be also defined by critical Android services.

Call protection services: We identify three external companies providing services for blocking undesired and spam phone calls and text messages: Hiya [Hiya], TrueCaller [Truc], and PrivacyStar [Prib]. Hiya's solution seems to be integrated by T-Mobile (US), Orange (Spain), and AT&T (US) in their subsidized Samsung and LG handsets according to the package signatures. Hiya and TrueCaller's privacy policies indicate that they collect personal data from end users, including contacts stored in the device, UIDs, and personal information [Hiyb].[5] PrivacyStar's privacy policy, instead, claims that any information collected from a given user's contacts is "NOT exported outside the App for any purpose" [Pric].

5.3.2 Requested Permissions

The use of permissions by pre-installed Android apps follows a power-law distribution: 4,736 of the package names request at least one permission and 55 apps request more that 100. The fact that pre-installed apps request many permissions to deliver their service does not necessarily imply a breach of privacy for the user. However, we identified a significant number of potentially over-privileged vendor- and MNO-specific packages with suspicious activities such as `com.jrdcom.Elabel` – a package signed by TCLMobile requesting 145 permissions and labeled as malicious by Hybrid Analysis (a free online malware analysis service) –and `com.cube26.coolstore` (144 permissions). Likewise, the calculator app found on a Xiaomi Mi 4c requests user's location and the phone state, which gives it access to UIDs such as the

[5]Note: the information rendered in their privacy policy differs when crawled from a machine in the EU or the US. As of January 2019, none of these companies mention the new European GDPR directive in their privacy policies.

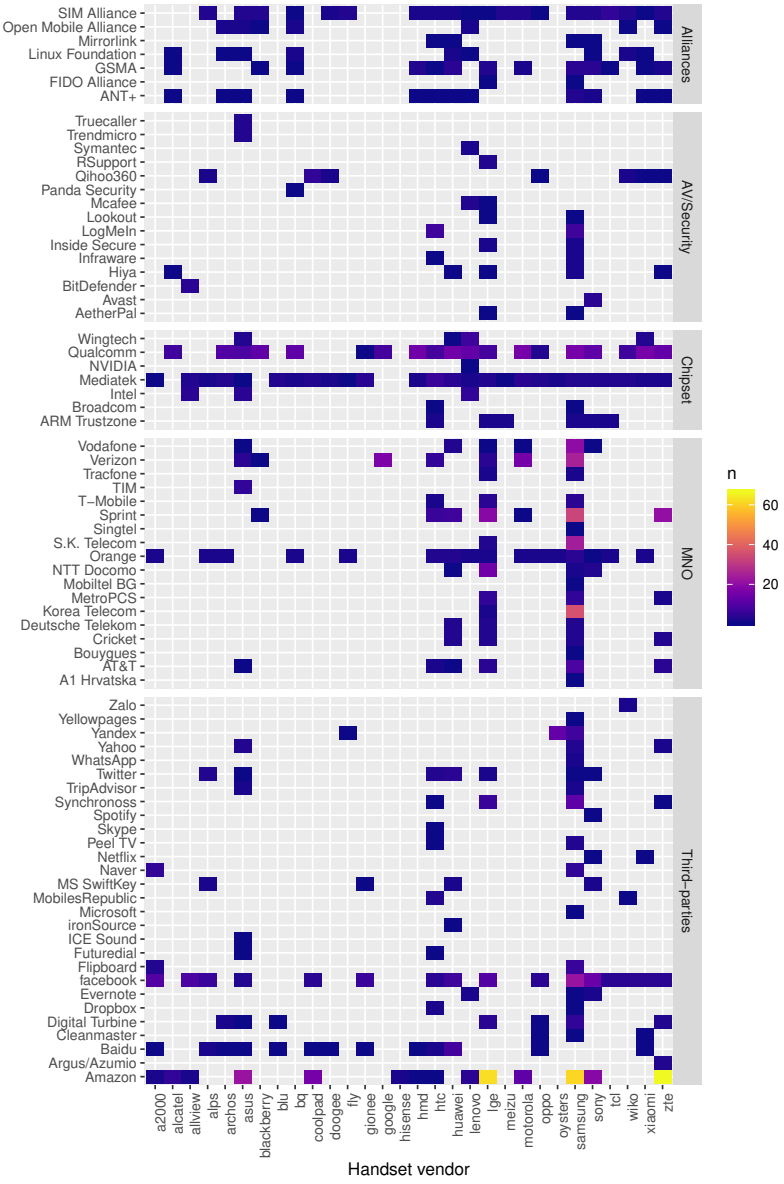


Figure 8: Permissions defined by anti-virus firms, mobile network operators, chipset vendors and third parties, requested by pre-installed apps.

International Mobile Equipment Identity (IMEI). We discuss more instances of over-privileged apps in Section 5.4.3.

Dangerous Android permissions. The median pre-installed Android app requests three dangerous AOSP permissions. When we look at the set of permissions requested by a given app (by its package name) across vendors, we can notice significant differences. We investigate such variations in a subset of 150 package names present at least in 20 different vendors. This list contains mainly core Android services as well as apps signed by independent companies (e.g., Adups) and chipset manufacturers (e.g., Qualcomm).

Then, we group together all the permissions requested by a given package name across all device models for each brand. As in the case of exposed custom permissions, we can see a

tendency towards over-privileging these modules in specific vendors. For instance, the number of permissions requested by the core `android` module can range from 9 permissions in a Google-branded Android device to over 100 in most Samsung devices. Likewise, while the median `com.android.contacts` service requests 35 permissions, this number goes over 100 for Samsung, Huawei, Advan, and LG devices.

Custom permissions. 2,910 pre-installed apps request at least one custom permission. The heatmap in Figure 8 shows the number of custom permissions requested by pre-installed packages in a hand-picked set of popular Android manufacturers (x-axis). As we can see, the use of custom permissions also varies across vendors, with those associated with large third-party analytics and tracking services (e.g., Facebook), MNOs (e.g., Vodafone), and AV/security services (e.g., Hiya) being the most requested ones.

This analysis uncovers possible partnerships beyond those revealed in the previous sections. We identify vendor-signed services accessing ironSource's, Hiya's, and AccuWeather's permissions. This state of affairs potentially allows third-party services and developers to gain access to protected permissions requested by other pre-installed packages signed with the same signature. Further, we found Sprint-signed packages resembling that of Facebook and Facebook's Messenger APKs (`com.facebook.orca.vpl` and `com.facebook.katana.vpl`) requesting Flurry-related permissions (a third-party tracking service owned by Verizon).

Commercial relationships between third-party services and vendors appear to be bidirectional as shown in Figure 9. This figure shows evidence of 87 apps accessing vendor permissions, including packages signed by Facebook, ironSource, Hiya, Digital Turbine, Amazon, Verizon, Spotify, various browser, and MNOs – grouped by developer signature for clarity purposes. As the heatmap indicates, Samsung, HTC and Sony are the vendors enabling most of the custom permissions requested by over-the-top apps. We found instances of apps listed on the Play Store also requesting such permissions. Unfortunately, custom permissions are not shown to users when shopping for mobile apps in the store–therefore they are apparently requested without consent–allowing them to cause serious damage to users' privacy when misused by apps.

5.3.3 Permission Usage by Third-Party Libraries

We look at the permissions used by apps embedding at least one TPL. We study the access to permissions with a protection level of either `signature` or `signature|privileged` as they can only be granted to system apps [Sys] or those signed with a system signature. The presence of TPLs in pre-installed apps requesting access to a signature or dangerous permission

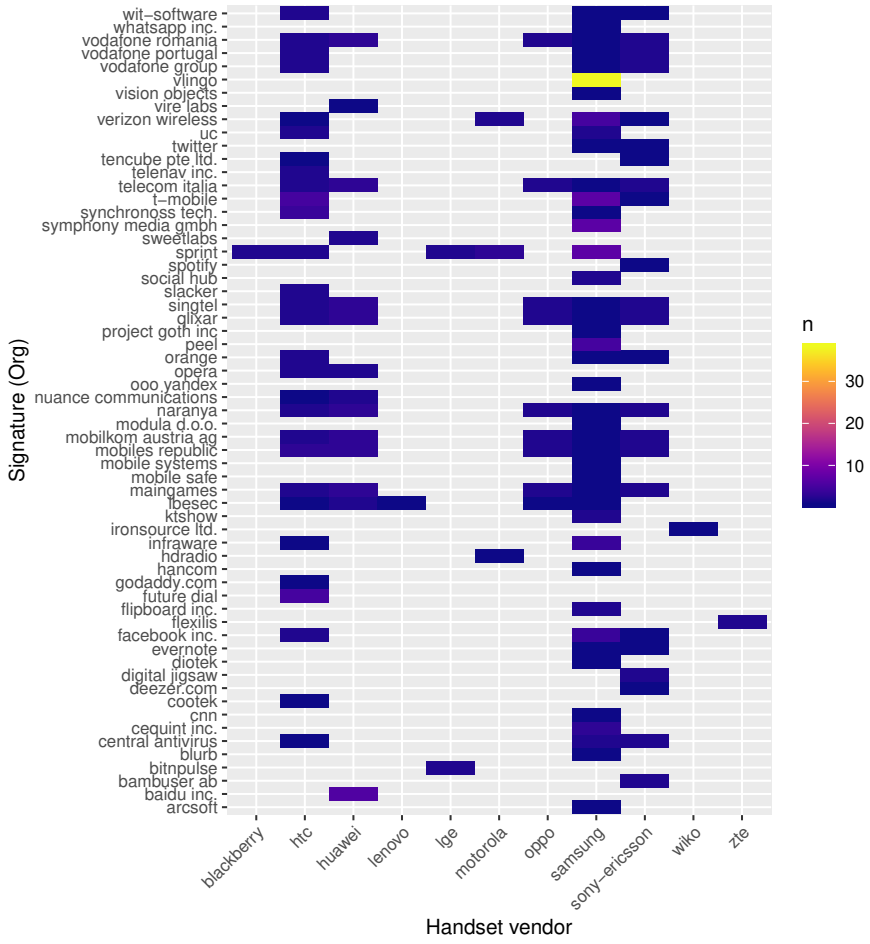


Figure 9: Apps accessing vendors' custom permissions.

can, therefore, give it access to very sensitive resources without user awareness and consent. Figure 10 shows the distribution of signature permissions requested across apps embedding TPLs. We find that the most used permissions—`READ_LOGS`—allows the app (and thus its embedded TPLs) to read system logs, mount and unmount file systems, or install extra packages. We find no significant differences between the three types of TPLs of interest. For completeness, we also find that 94 apps embedding TPLs of interest request custom permissions as well. Interestingly, 53% of the 88 custom permissions used by these apps are defined by Samsung.

5.3.4 Component Exposing

Custom permissions are not the only mechanism available for app developers to expose (or access) features and components to (or from) other apps. Android apps can also interact with each other using *intents*, a high-level communication abstraction [Andz]. An app may expose its component(s) to external apps by defining `android:exported=true` in the manifest without protecting the component with any additional measure, or by adding one or more

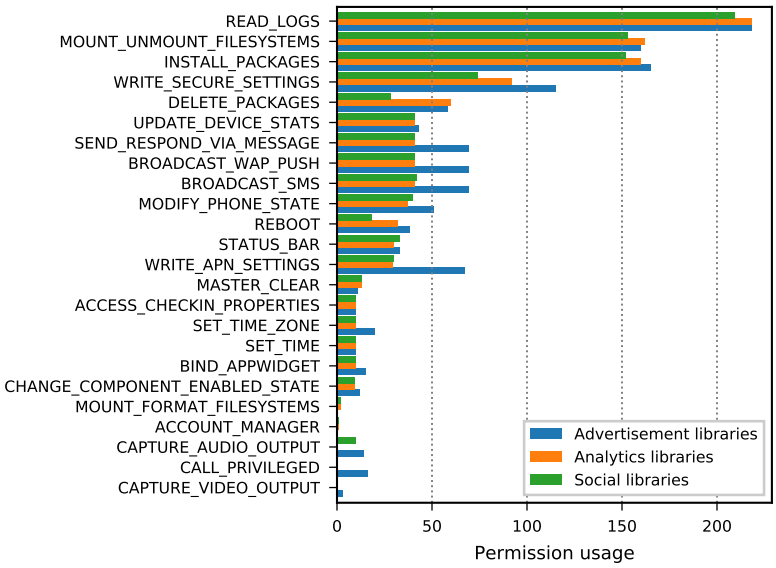


Figure 10: System permissions requested by pre-installed apps embedding third-party libraries.

intent-filters to its definition in the manifest; exposing it to a type of attack known in the literature as a confused deputy attack [Fel+11c]. If the `exported` attribute is used, it can be protected by adding a permission to the component, be it a custom permission or an AOSP one, through checking the caller app's permissions programmatically in the component's Java class.

We sought to identify potentially careless development practices that may lead to components getting exposed without any additional protection. Exporting components can lead to: *i*) harmful or malicious apps launching an exposed activity, tricking users into believing that they are interacting with the benign one; *ii*) initiating and binding to unprotected services; and *iii*) malicious apps gaining access to sensitive data or the ability to modify the app's internal state.

We found 6,849 pre-installed apps that potentially expose at least one activity in devices from 166 vendors and signed by 261 developer signatures with `exported=true`. For services, 4,591 apps (present in 157 vendors) signed by 183 developers including manufacturers, potentially exposed one or more of their services to external apps. The top-10 vendors in our dataset account for over 70% of the potentially exposed activities and services. Relevant examples include an app that potentially exposes several activities related to system configurations (device administration, networking, etc.), hence allowing a malicious developer could access or even tamper a users' device settings. The core package `com.android.mms` found in customized firmware versions across several vendors also expose services to read WAP messages to other apps. We also found 8 different instances of a third-party app, found in handsets built by two large Android manufacturers, whose intended purpose is to provide remote technical support

to customers. This particular service provides remote administration to MNOs, including the ability to record audio and video, browse files, access system settings, and upload/download files. The key service to do so is exposed and can be misused by other apps.

We leave the detailed study of apps vulnerable to confused deputy attacks and the study of the access to these resources by apps publicly available on Google Play for future work.

5.4 Behavioral Analysis

We analyze the apps in our dataset to identify potentially harmful and unwanted behaviors. To do this, we leverage both static and dynamic analysis tools to elicit behavior and characterize purpose and means. This section describes our analysis pipeline and evidence of potentially harmful and privacy-intrusive pre-installed packages.

5.4.1 Static Analysis

We triage all apps to determine the presence of potentially harmful behaviors. This step allows us to obtain a high-level overview of behaviors across the dataset and also provides us with the basis to score apps and flag those potentially more interesting. This step is critical since we could only afford to manually inspect a limited subset of all available apps.

Toolkit. Our analysis pipeline integrates various static analysis tools to elicit behavior in Android apps, including Androwarn [Andq], FlowDroid [Arz+14], and Amandroid [Wei+14], as well as a number of custom scripts based on the Apktool [Apkc] and Androguard [Anda] frameworks. In this stage we do not use dynamic analysis tools, which prevents us from identifying hidden behaviors that rely on dynamic code uploading (DEX loading) or reflection. This means that our results present a lower-bound estimation of all the possible potentially harmful behaviors. We search for apps using DEX loading and reflection to identify targets that deserve manual inspection.

Dataset. Because of scalability limitations —our dataset comprises 82,501 APK files with 6,496 unique package names— we randomly select one APK file for each package name and analyze the resulting set of apps, obtaining an analysis report for 48% of them. The majority of the remaining packages could not be analyzed due to the absence of a `classes.dex` for ODEXed files. Even though in some cases we had the corresponding `.odex` file, we generally could not de-ODEX it since the device's Android framework file was needed to complete this step but Firmware Scanner did not collect it. Moreover, we could not analyze a small subset of apps due to the limitations of our tools, including errors generated during analysis, file size limitations,

or analysis tools becoming unresponsive after hours of processing. Instead, we focused our analysis on the subset of apps for which we could generate reports.

Results. We processed the analysis reports and identified the presence of the 36 potentially privacy intrusive behaviors or potentially harmful behaviors listed in Table 5.8. The results suggest that a significant fraction of the analyzed apps could access and disseminate both user and device identifiers, user's location, and device current configuration. According to our flow analysis, these results give the impression that personal data collection and dissemination (regardless of the purpose or consent) is not only pervasive but also comes pre-installed. Other a priori concerning behaviors include the possible dissemination of contacts and SMS contents (164 and 74 apps, respectively), sending SMS (29 apps), and making phone calls (339 apps). Even though there are perfectly legitimate use cases for these behaviors, they are also prevalent in harmful and potentially unwanted software. The distribution of the number of potentially harmful behaviors per app follows a power-law distribution. Around 25% of the analyzed apps present at least 5 of these behaviors, with almost 1% of the apps showing 20 or more. The bulk of the distribution relates to the collection of telephony and network identifiers, interaction with the package manager, and logging activities. This provides a glimpse of how pervasive user and device fingerprinting is nowadays.

5.4.2 Traffic Analysis

While static analysis can be helpful to determine a lower bound of what an app is capable of, relying on this technique alone gives an incomplete picture of the real-world behavior of an app. This might be due to code paths that are not available at the time of analysis, including those that are within statically- and dynamically-linked libraries that are not provided with apps, behaviors determined by server-side logic (e.g., due to real-time ad-bidding), or code that is loaded at runtime using Java's reflection APIs. This limitation of static approaches is generally addressed by complementing static analysis with dynamic analysis tools. However, due to various limitations (including missing hardware features and software components) it was unfeasible for us to run all the pre-installed apps in our dataset in an analysis sandbox. Instead, we decided to use the crowd-sourced Lumen mobile traffic dataset to find evidence of dissemination of personal data from the pre-installed apps by examining packages that exist in both datasets.

Results. Of the 3,118 pre-installed apps with Internet access permissions, 1,055 have at least one flow in the Lumen dataset. At this point, our analysis of these apps focused on two main aspects: uncovering the ecosystem of organizations who own the domains that these apps

Accessed PII type / behaviors		Apps (#)	Apps (%)
Telephony identifiers	IMEI	687	21.8%
	IMSI	379	12%
	Phone number	303	9.6%
	MCC	552	17.5%
	MNC	552	17.5%
	Operator name	315	10%
	SIM Serial number	181	5.7%
	SIM State	383	12.1%
	Current country	194	6.2%
	SIM country	196	6.2%
	Voicemail number	29	0.9%
Device settings	Software version	25	0.8%
	Phone state	265	8.4%
	Installed apps	1,286	40.8%
	Phone type	375	11.9%
	Logs	2,568	81.4%
Location	GPS	54	1.7%
	Cell location	158	5%
	CID	162	5.1%
	LAC	137	4.3%
Network interfaces	Wi-Fi configuration	9	0.3%
	Current network	1,373	43.5%
	Data plan	699	22.2%
	Connection state	71	2.3%
	Network type	345	10.9%
Personal data	Contacts	164	11%
	SMS	73	2.3%
Phone service abuse	SMS sending	29	0.9%
	SMS interception	0	0%
	Disabling SMS notif.	0	0%
	Phone calls	339	10.7%
Audio/video interception	Audio recording	74	2.4%
	Video capture	21	0.7%
Arbitrary code execution	Native code	775	24.6%
	Linux commands	563	17.9%
Remote conn.	Remote connection	89	2.8%

Table 5.8: Volume of apps accessing / reading PIIs or showing potentially harmful behaviors. The percentage is referred to the subset of triaged packages ($N = 3,154$).

connect to, and analyzing the types of private information they could disseminate from user devices. To understand the ecosystem of data collection by pre-installed apps, we studied where the data that is collected by these apps makes its first stop. We use the fully qualified domain name (FQDN) of the servers that are contacted and use the web crawling and text min-

Organization	# of apps	# of domains
Alphabet	566	17052
Facebook	322	3325
Amazon	201	991
Samsung Electronics	187	571
Verizon Communications	171	320
Twitter	137	101
Microsoft	136	408
CloudFront	121	711
Adobe	116	302
AppsFlyer	98	10
Xiaomi	95	200
comScore	86	8
AccuWeather	86	15
MoatInc.	79	20
Appnexus	79	35
Baidu	72	69
Criteo	70	62
PerfectPrivacy	68	28
The Trade Desk	66	17
Other Advertising and Tracking Service (ATS)	221	362

Table 5.9: Top 20 parent ATS organizations by number of apps connecting to all their associated domains.

ing techniques described in our previous work [Raz+18] to determine the parent organization who own these domains.

The Big Players. Table 5.9 shows the parent organizations who own the most popular domains contacted by pre-installed apps in the Lumen dataset. Of the 54,614 domains contacted by apps, 7,629 belong to well-known Advertising and Tracking Services (ATS) [Raz+18]. These services are represented by organizations like Alphabet, Facebook, Verizon (now owner of Yahoo!, AOL, and Flurry), Twitter (MoPub's parent organization), AppsFlyer, comScore, and others. As expected, Alphabet, the entity that owns and maintains the Android platform and many of the largest advertising and tracking services (ATS) in the mobile ecosystem [Raz+18], also owns most of the domains to which pre-installed apps connect to. Moreover, vendors who ship their devices with the Google Play Store have to go through Google's certification program which, in part, entails pre-loading Google's services. Among these services is Google's own `com.google.backuptransport` package, which sends a variety of information about the user and the device on which it runs to Google's servers.

Traffic analysis also confirms that Facebook and Twitter services come pre-installed on many phones and are integrated into various apps. Many devices also pre-install weather apps like AccuWeather and The Weather Channel. As reported by previous research efforts,

these weather providers also gather information about the devices and their users [Ren+18; Raz+18].

5.4.3 Manual Analysis: Relevant Cases

We used the output provided by our static and dynamic analysis pipeline to score apps and thus flag a reduced subset of packages to inspect manually. Our goal here was to confidently identify potentially harmful and unwanted behavior in pre-installed apps. Other apps were added to this set based on the results of our third-party library and permission analysis performed in Sections 5.2 and 5.3, respectively. We manually analyzed 158 apps using standard tools that include DEX disassemblers (baksmali), dex-to-java decompilers (jadx, dex2jar), resource analysis tools (Apktool), instrumentation tools (Frida), and binary analysis frameworks (radare2 and IDA Pro) for native code analysis. Our main findings can be loosely grouped into three large categories: 1) known malware; 2) potential personal data access and dissemination; and 3) potentially harmful apps. Table 5.10 provides some examples of the type of behaviors that we found.

Known Malware

Triada. We found a variant of the infamous Android Triada banking trojan pre-installed in a Samsung SM-N9600 coming from users in Brazil (package name: `com.zotoo.factorymode`). The sample is modular and fairly complex. We only conducted a superficial analysis and observed behaviors that include leaking of PII and personal data (SMS, call logs, contact data, stored pictures and phone's recording data), persistence after reboot, and downloading of additional stages. The sample roots the device to install additional apps, though our library failed to detect it as rooted. This is not the first time that a member of the Triada family has been detected pre-installed, though the previous case we are aware of affected low-end Android phones [Dr ; Tri] Android phones.

Rootnik. We found various malware samples on a non-rooted Huawei Hol-U19 phone coming from Myanmar. The most eye-catching (because of its complexity) turned out to be a variant of the well-known *rootnik* family [Rooa] located inside two packages named `com.android.backup` and `com.android.newbackup`. It gains root access to the device (again, our library does not detect it as rooted) and then leaks PII and installs additional apps. It uses several anti-analysis and anti-debugging techniques to make code analysis more difficult. We found 6 more malware samples in the same phone that we mapped to recent families: Xinyin (PUP/Adware), Ztorg (SMS Trojan), Triada, and Iop (PUP/Adware). The relationship between all these samples is

Type	Family	Device(s) (Country)	Behavior
Malware	Triada	Samsung SM-N9600 (BR)	Banking Trojan. Leaks PII and personal data (SMS, call logs, contact data, stored pictures and videos). Downloads additional stages. Roots the device to install additional apps.
	Rootnik [Rooa]	Huawei Hol-U19 (MM)	Gains root access to the device. Leaks PII and installs additional apps. Uses anti-analysis and anti-debugging techniques.
	GMobi [Gmoa; Ups]	Micromax D303 (RU) Polytron R2452 (ID) Leagoo Z5 (VT) Asus Zenfone 2 Laser (IN) Fly IQ4416 (UA) Micromax A177 (IN)	Gmobi Trade Service. Leaks PII, including CPU serial number and MAC address, geolocation, installed packages and emails. Receives commands from servers to (1) send an SMS to a given number; (2) download and install an app; (3) visit a link; or (4) display a pop-up.
Potentially Dangerous	Rooting app	Samsung T8055 (VT)	Exposes an unprotected receiver that roots the device upon receiving a telephony secret code 9527 (via intent or dialing `*#*#9527#*#*`).
	Blocker	Sony Xperia X F5121 (AR) Sony Xperia X Compact (ES) Sony Xperia XZ2 Compact (DE) Sony Xperia XZ3 H8416 (UK)	If the device does not contain a signed file in a particular location, it loads and enforces 2 blacklists: one containing 103 packages associated with benchmarking apps, and another with 56 web domains related to phone reviews.
Spyware / Tracking	Truecaller	ASUS Zenfone 3 and 3 Max (ES, FR, SK)	Sends PII to its own servers and as third-party ATSes such as AppsFlyer, Twitter-owned MoPub, Crashlytics, inMobi, Facebook, and others. Uploads phone call data to at least one of its own domains.
	Metro Name ID	Metro PCS Phones (US)	Sends PII to domains owned by Metro PCS and its own analytics servers. Embeds Piano, a media audience and engagement analytics service that tracks user's installation of news apps and other partners including those made by CNBC, The Daily Beast, Bloomberg, TechCrunch, The Economist, among others, the presence of which it reports to its own domains.
	Adups [Adu]	Low-end phones from 55 vendors (worldwide)	FOTA app. Collects and shares private and PII with their own servers and those of third-party ATS domains, including Advmob and Nexage.
	Stats/Meteor	OPPO P9 Pro (TH)	Redstone's FOTA service. Uses dynamic code uploading and reflection to deploy components located in 2 encrypted DEX files. Leaks around 50 data items that fully characterize the hardware, the telephony service, the network, geolocation, and installed packages. Performs behavioral and performance profiling, including counts of SMS/MMS, call logs, bytes sent and transmitted, and usage stats and performance counters on a package-basis. Silently installs packages on the device and reports what packages are installed / removed by the user.

Table 5.10: Examples of relevant cases found after manual analysis of a subset of apps. When referring to leaks, the term PII encompasses items such as those enumerated in Table 5.8 in the categories "Telephony identifiers", "Device settings", and "Network interfaces."

unclear: they all might have come pre-installed or, more likely, one facilitated the installation of the remaining ones.

GMobi. We identified 6 cases of the GMobi malware family [Gmoa] pre-installed in various phones manufactured by Asus, Leagoo, Polytron, Micromax, and Fly located in India, Vietnam, Indonesia, Russia and Ukraine. The package names are `com.trendmicro.freetmms.gmobi`, `com.rock.gota`, and `com.redbend.dmClient`. The first two are FOTA updating apps, whereas

the latter is Trend Micro's Dr. Safety app. The packages deploy three key components, called: `com.gmobi.trade.ActionActivity`, `.ActionService`, and `.ActionMonitor`. These three modules jointly implement the so-called Gmobi Trade Service, which: (i) leaks a considerable amount of sensitive information from the device, including items often used for tracking (IMEI, IMSI, phone number, CPU serial number, MAC address, etc.), geolocation, all installed packages and emails; and (ii) receives commands from command and control servers and implements four actions: (1) send an SMS to a given number with the text content provided in the command message; (2) download and install an app; (3) visit a link; and (4) display a popup. A July 2018 Wall Street Journal article reported on how the GMobi SDK is being used in some emerging markets to perform systematic collection and transfer of PII and ad fraud. Technical details can be found in [Ups].

Dangerous and Potentially Unwanted Apps

The LogUploaderProxy case. We analyzed a package named `com.asus.loguploaderproxy` that we found pre-installed in 5 different Asus devices: a Zenfone ZC553KL, a ZenPad 3S 10, a Zenfone Max Pro M1, a Zenfone 3 ZE552KL, and a Zenfone 4 Max ZC554KL. The devices came from users in India, US, Spain and France. The app registers a broadcast receiver called `com.asus.loguploaderproxy.LogUploaderReceiver`. This component is exported and protected with the custom permission `asus.permission.MOVELOGS`, which means that only apps holding that permission can send intents to it.

It listens for 6 different types of messages (`SETPROP`, `EXECADB`, `CATSYSTEMFILE`, `COMBINEKEY`, `PERMISSIONS`, `INSTALL`), though it only handles the first five of them programmatically (i.e., it ignores `INSTALL` intents). The `SETPROP` intent requests the app to set a system property to a desired value, both provided as arguments, by invoking `android.os.SystemProperties.set()`. The `EXECADB` intent executes the command passed as argument and returns the output. The `CATSYSTEMFILE` intent executes the `cat` command with the file name passed as argument and returns the output. The intent `COMBINEKEY` uses `android.hardware.input.InputManager` to inject input events on other apps. Finally, the message `PERMISSIONS` grants to a package named `com.asus.loguploader` a list of runtime permissions passed as arguments. This is possible because `com.asus.loguploaderproxy` is granted the `GRANT_REVOKE_PERMISSIONS` permission. We analyzed this second package and found out it is a bug reporting tool used to upload different logs. The app grants itself the `WRITE_EXTERNAL_STORAGE`, `READ_EXTERNAL_STORAGE`, `READ_PHONE_STATE`, `ACCESS_FINE_LOCATION`, and `ACCESS_COARSE_LOCATION` permissions. These are needed to send, together with the logs, device-specific information (IMEI,

installed packages, firmware version, etc.) and its location.

The com.asus.loguploaderproxy can be leveraged by any attacker who tricks the user into installing an app with the asus.permission.MOVELOGS permission. Such an app will then be able to send intents and use it to execute any of the provided services.

Dial *#*#9527#*#* for rooting. We found an app named com.rd.user2root that came pre-installed in a Samsung T8055 device. The app exposes a broadcast receiver that is activated through an android.provider.Telephony.SECRET_CODE intent associated with the number "9527". This, in turn, runs an activity that modifies some key properties to effectively run in root mode (specifically, persist.service.adb.enable and ro.debuggable are set to 1, and ro.secure is set to 0.). The receiver is not protected through a permission, so any app can effectively use this capability to root the device. The app is signed by an organization named Jlink based on Shenzhen, China.

Blocker. We came across a package called com.sonymobile.pip that was found in 4 Sony Ericsson devices: an Xperia X, an Xperia X Compact, an Xperia XZ2 Compact, an Xperia XZ3. Their users came from Argentina, Spain, Germany and the UK. The app exposes a service and a receiver. The service disables itself if the user is "authorized", which is determined as true if the service correctly decrypts a file named SOMCPrototypeProtectionKey using a private RSA key located in the assets folder. The result after decryption is checked against a hardcoded string ("01234567890123"). If the user is not authorized, then the service loads and applies two blacklists for packages and domains, respectively. The first list contains 103 package names, most of which are performance benchmarking apps. The service disables any package found in the device that matches a name in the list. The second blacklist contains 56 domains related with technology and phone reviews. The service leverages com.sonymobile.packetfilter and adds one rule per domain to effectively block any traffic coming from or going to any of these domains. The package also registers a receiver that subscribes to the BOOT_COMPLETED and PACKAGE_ADDED events to run the blocking service. In all cases, the app certificate claims to belong to Sony Ericsson Mobile Communications.

Extreme User Profiling

The libcore case. We analyzed a package named com.redstone.ota.ui found pre-installed in an Alps phone coming from Bangkok. The app provides a FOTA service associated with Redstone Sunshine Technology Co., Ltd. [Redb], a Beijing-based FOTA provider that "supports 550 million phone users and IoT partners in 40 countries" [Reda]. The package also exposes four suspicious components called com.android.meteor.agent.library.AgentReceiver,

com.android.meteor.agent.library.AgentService, com.globe.android.stats.agent.AgentService, and com.globe.android.stats.agent.AgentReceiver with almost identical code components. The com.globe.android.stats.agent component loads a file called libcore.jar located in the assets folder and performs a rudimentary integrity checking. It then decrypts it using a custom encryption routine. The output is a DEX file that it is dynamically loaded and executed. It contains various components that implement an extraordinarily extensive fingerprinting of the device and user activities. It collects more than 60 parameters that characterize the hardware (including CPU, RAM, screen and sensors), the telephony service (SIM, IMEI, IMSI, line number, etc.), network information (network interfaces, SSID, base station ID, signal strength, etc.), geolocation, and installed packages. Another component implements what seems to be a fine-grained behavioral and performance profiler. This includes an hourly time series with counts of SMS, MMS, calls made and received, and bytes sent and transmitted. It also implements usage statistics and performance counters on a package-basis within a component called com.android.internal.os.PkgUsageStats. All in all, this amount to hundreds of data items which are encapsulated in a JSON and then zipped and shipped on a daily basis to two different hardcoded third-party server using a RESTful API: g.sinfoon.com:40081 and 42.62.125.197:40081. The domain is tagged as part of the web infrastructure of an operation by at least one anti-malware vendor (Forcepoint).

Overall, this seems to implement an analytics program that admits several monetization strategies: from optimized ad targeting to performance feedback for both developers and manufacturers. We emphasize that the data collected is not only remarkably extensive, but also very far away from being anonymous.

The com.android.meteor.agent component follows an identical loading strategy, though using a file called libcore64.jar located in the assets folder. The services contained in the decrypted DEX file implement an app promotion scheme that silently installs packages on the device under command. It also reports periodically what packages are installed and removed by the user. The report and pull servers are different, and different from those used previously: mad.dwphonetest.com:58801 and 114.215.238.53:58801. The former is tagged as "known infection source" in various malware lists. The IP was associated with the domain hwdl.redstone.net.cn in 2016. It currently belongs to a Chinese AS (37963) associated with Hangzhou Alibaba Advertising.

Paid PII Leaking Partners

GMobi. In addition to the larger companies, there is a long tail of third-party ATSes that are integrated into many pre-installed apps, or are pre-installed as an independent app that collects information about users without their knowledge or consent. One notable example of the case of the General Mobile Corporation (GMobi), a mobile ad tech company that also operates a "performance-based ad platform that enables content monetization and global users acquisition" with a reach of "150M installs base in more than 120 countries" [Gmob]. The GMobi components come pre-installed with lower-end mobile devices such as those made by Micromax, Intex, Lenovo, and some Samsung phones. GMobi services usually come in the form of a pre-installed app called `com.rock.gota`, that seems just like a Firmware Over The Air (FOTA) updater app. However, it actually connects to GMobi to leak sensitive device and subscriber-related identifiers such as the operator information, IMSI, IMEI, device serial, Android Advertising ID (AAID); MAC addresses of networking interfaces which can be used to geo-locate users [Ftc]; as well as a full list of all apps installed on the device to GMobi's advertising domains such as `api.ads.go2reach.com` and `api.reachads.com`. GMobi apps send all of this information in plain-text HTTP and without any encryption or hashing, enabling everyone in the path to be able to sniff this information with ease, risking further leakage of important identifying information.

Adups. Similarly, Adups FOTA apps collect and share a wealth of private and identifying information including phone number, IMSI, Android and device serial, and MAC addresses with their own servers and those of third-party ATS domains including Advmob and Nexage.

Truecaller. Another notable example of these services is Truecaller, a phone dialer app that comes pre-installed on some phones, including ones made by ASUS. This app sends the user's phone number, IMEI, IMSI, AAID, SMS address, operator information, device serial, and other device and subscriber-related identifiers to its own servers as well as third-party ATSes such as AppsFlyer, Twitter-owned MoPub, Crashlytics, inMobi, Facebook, and others. Moreover, it uploads phone call data to at least one of its own domains. A similar dialer app called Metro Name ID comes pre-installed on MetroPCS phones that is made by a company called PrivacyStar. In addition to sending the user's AAID, phone number, and IMSI to domains owned by MetroPCS, it also sends the same information to its own analytics servers. Moreover, it embeds another third-party service called Piano, a media audience and engagement analytics service that tracks user's installation of news apps and other partners including those made by CNBC, The Daily Beast, Bloomberg, TechCrunch, The Economist, among others, the presence of which it reports to its own domains.

5.5 A Case Study: Apps Accessing System Logs

Logging is a core part of software development.[6] In development, logging the right things can help find and fix bugs quickly. In a production environment, logging helps locate issues that may occur when no other data is available. It also can be used to make sure a system is running smoothly and monitor what it is doing. In the case of a crash, the system logs can be useful to locate the root cause of the issue. Apps and system components running on the Android platform have a shared system log. By default, apps can only read their own entries. Only apps that are granted the `android.permission.READ_LOGS` permission can access the full, unfiltered system logs. This permission has a `signature|privileged` protection level, which makes it only available to system apps. These systems logs can contain arbitrary data: apps are free to log at their own discretion and are given specific guidance by Google not to log private information [Gooc].

However, anecdotal evidence shows that Google itself may log personal information of users of the Google-Apple Exposure Notifications (GAEN), a framework developed jointly by Google and Apple to implement contact tracing during the COVID-19 pandemic. The GAEN framework broadcast anonymous rolling proximity identifiers (RPIs) that are later identified as being a risky encounter for the person who observed the identifier if the emitter tested positive for COVID-19. Third-party apps, specifically contact tracing apps developed by national health authorities, could then rely on with this framework for their apps. Reardon et al. observed that Google's implementation of GAEN was logging the RPIs emitted by the user, the received RPIs along with the nearby device's randomized Bluetooth MAC address, and the risk assessment warnings delivered to the user [Rea].

5.5.1 Logged PII in the Wild

We first organize a crowdsourcing campaign to examine at scale the presence of PII in real users logs. We rely on the WebUSB API built into recent versions of Google Chrome [Weba], and on an open source implementation of ADB in pure Javascript [Webb; Webc; Webd]. Using these tools, we develop a website that is able to access users' devices connected via USB and run ADB command directly from Javascript, and therefore access the system logs directly from the user's web browser. We also develop a purpose built app to gather unique identifiers directly from the users devices. This app is installed using the WebUSB interface when users agree to participate in the study. We then use the output of this app to look for the presence of unique

[6] This work was conducted after the publication of our paper on pre-installed apps. In this section only, our dataset of contains 89,147 unique devices and 1,247,447 unique apps.

identifiers in the system logs. Note that we do not collect the raw system logs nor the list of PII for privacy reasons. Rather, we report the numbers of times a given PII was spotted in the logs.

We deployed this website on the 14th of March, 2022 up until the 21st of the same month. During that time, we recruited volunteers through institutional mailing list and social networks, but also from specialized platforms such as Prolific [Pro] and Amazon Mechanical Turk [Amab]. In total, 1,400 participants provided reports from 866 unique devices (based on their build fingerprint) comprising 571 models from 46 manufacturers. As reported in Table 5.11, the presence of PII in the system logs is common, despite Google's recommendations. We find occurrences of every single PII that we collect from our app. While some are rather rare, such as the IMEI, present in only 1% of our participants' devices, other are more frequent. The Wi-Fi router MAC address for instance is present in 67% of the participants' devices. This state of affairs is concerning: should third-party apps be able to access the system logs, they might be able to access private information.

Table 5.11: Number of devices with PII in their system logs

PII type	Number	Percent
Android ID	92	8%
Bluetooth MAC Address	138	11%
Bluetooth Name	841	69%
Bluetooth Scan MAC	26	2%
Bluetooth Scan SSID	26	2%
Email Address	198	16%
IMEI	16	1%
Phone Number	14	1%
Coarse Location	293	24%
Location	272	22%
Wi-Fi MAC Address	544	45%
Wi-Fi Router MAC	821	67%
Wi-Fi Router SSID	834	68%
Wi-Fi Scan MAC	177	15%
Wi-Fi Scan SSID	480	39%
Serial Number	50	4%

5.5.2 System Logs Exfiltration

Because of this, it is important to know exactly how many different entities that are potentially able to access this data. We investigate whether there are some system apps that may access the system logs, and leak them to the cloud. We cross check the build fingerprints of the users' devices that participated in our crowdsourcing campaign with our database of pre-

installed apps collected using Firmware Scanner. Our of the 866 unique build fingerprints, 315 are present in our database, from 2,015 users. We find 1,319 apps that request the `READ_LOGS` permission on those devices. When grouping them by their package name and signing certificate, we find 237 groups. For each of these groups, we manually inspect the code of the latest version.

We analyze automatically our subset of apps and look for the presence of the string `logcat` in the code: we find 73 such apps. After manual inspection of the apps, we find 63 that run `logcat` as a shell command. We manually inspect the code of these apps and find that 7 of them filter the logs after retrieving them. Some apps look for a specific pattern (e.g., for a specific package name or PID), triggered upon specific event (e.g., an app crash). However, the rest of the apps do not filter anything and keep the full log. Among the apps that do not filter the logs, we find 15 that save these raw logs directly on the SD card. This essentially makes the logs available to any app that has the permission `READ_EXTERNAL_STORAGE`. More worrying, we find 9 apps that send the raw logs on the Internet. In one case, the logs are send to a Firebase instance, a third-party service operated by Google.

In total, we find 4,598 users of Firmware Scanner that have at least one of these apps that save the raw logs on the SD card or upload them, installed on a system partition. Our analysis reveal that this behavior is in some cases triggered upon a specific event, such as an app crash. This is the case of the Google Feedback app which allow the user to attach the system logs to the bug report sent to Google. We manually confirm that, if the user allows it, the full, unfiltered logs are sent to Google servers. In other cases, the behavior may be triggered by the reception of an intent with a specific action or extra, which hints at the presence of one or more other apps that have the logic to trigger said behavior. The majority of these apps are signed by the manufacturer of the device on which they were found pre-installed. However, some of these apps are signed by third-party companies, such as Vodafone. We find cases where the signing certificate does not give much useful information as to who is the company behind it, such as `C=IL, ST=il, L=TLV, O=Central antivirus, OU=antivirus, CN=Dror Shalev` or `O=voiceservice, OU=translator`.

Another example is the apps developed by Mobile Posse [Mob], a third-party advertising company that was bought by Digital Turbine in February 2020 [Dtb]. We found 8 such apps scattered across 68 unique devices. The app signing certificates of these apps indicate that they were not developed by the OEM of the devices. We manually inspected the code of these 8 apps. All of the samples contain the same code that accesses the logs. The code explicitly checks if the app has been granted the `READ_LOGS` permissions and, if so, runs `logcat` with

the `-d`[7] and saves the output. The logs are then converted into a JSON-formatted string and sent as an HTTP POST request. While we were not able to confirm with complete certainty the destination of this request, we found strong indications in the code that the logs are sent to an AWS domain.

The app also contains a JSON-formatted string called "schedule" that appears to contain data collection instructions, including which components are to be collected and at what frequency. The schedule contains, among other things, the `collect_system_log_schedule` operation which gives the `logcat` command to run: `logcat -v time -d *:e"`. The other operations seem to instruct the apps to collect very sensitive information, such as the list of installed apps, app usage, visited URLs, geographic and cell location, call history, signal strength, network info, connection speed (with links to test upload and download speed), boot time, SMS usage, battery status, and memory usage. This information would then be sent alongside the logs to the aforementioned domain.

Overall, these results highlight again the lack of control of the supply chain of Android devices. Google explicitly forbids apps from logging private data "unless it's absolutely necessary to provide the core functionality of the app" [Gooc], and has implemented tests as part of the CTS to verify that this rule is followed by devices that seek Google's certification. Despite these efforts, we still find a widespread presence of PII in logs, which can then be accessed and uploaded without any filtering whatsoever, in clear violation of the rules.

5.6 Study Limitations

Completeness and coverage. Our dataset is not complete in terms of Android vendors and models, and we might miss some brands or models. We, however, cover those with a larger market share, both in the high- and low-end parts of the spectrum. Our data collection process is also best-effort. The lack of background knowledge and documentation required performing a detailed case-by-case study and a significant amount of manual inspection. In terms of analyzed apps, determining the coverage of our study is difficult since we do not know the total number of pre-installed apps in all shipped handsets. Further, we cannot rely on the package name to reduce the number of samples as any actor can extend an open source app to include deceptive or even malicious behaviors. We have found evidence of such practices in our dataset.

Attribution. There is currently no reliable way to accurately find the legitimate developer of a given pre-installed app by its self-signed signature. We have found instances of certifi-

[7] The `-d` option makes `logcat` dump the whole log buffer and exit immediately.

cates with just a country code in the `Issuer` field, and others with strings suggesting major vendors (e.g., Google) signed the app, where the apps certainly were not signed by them. The same applies to package and permission names, many of which are opaque and not named following best-practices. Likewise, the lack of documentation regarding custom permissions prevented us from automatizing our analysis. Moreover, a deeper study of this issue would require checking whether those permissions are granted in runtime, tracing the code to fully identify their purpose, and finding whether they are actually used by other apps in the wild, and at scale.

Package Manager. We do not collect the `packages.xml` file from our users' devices as it contains information about all installed packages, and not just pre-installed ones. We consider that collecting this file would be invasive. This, however, limits our ability to see if user-installed apps are using services exposed by pre-installed apps via intents or custom permissions. We tried to compensate for that with a manual search for public apps that use pre-installed custom permissions, as discussed in Section 5.3.4.

Behavioral coverage. Our study mainly relies on static analysis of the samples harvested through Firmware Scanner, and we only applied dynamic analysis to a selected subset of 1,055 packages. This prevents us from eliciting behaviors that are only available at runtime because of the use of code loading and reflection, and also code downloading from third-party servers. Despite this, our analysis pipeline served to identify a considerable amount of potentially harmful behaviors. A deeper and broader analysis would possibly uncover more cases.

Identifying rooted devices. There is no sure way of knowing whether a device is rooted or not. While our conservative approach limits the number of false negatives, we have found occurrences of devices with well-known custom ROMs that were not flagged as rooted by RootBeer. Moreover, we have found some apps that allow a third party to root the device on-the-fly to, for example, install new apps on the system partition as discussed in Section 5.4.3. Some of these apps can then un-root the phone to avoid detection. Under the presence of such an app on a device, we cannot know for sure if a given package— particularly a potentially malicious app— was pre-installed by an actor in the supply chain, or was installed afterwards.

5.7 Takeaways

In this chapter, we studied, at scale, the vast and unexplored ecosystem of pre-installed Android software and its potential impact on consumers. This chapter has made clear that, thanks in large part to the open-source nature of the Android platform and the complexity of its supply chain, organizations of various kinds and sizes have the ability to embed their software in

custom Android firmware versions. As we demonstrated, this situation has become a peril to users' privacy and even security due to an abuse of privilege such as in the case of pre-installed malware, or as a result of poor software engineering practices that introduce vulnerabilities and dangerous backdoors.

Chapter 6

Evolution of the Permission System

"A change is as good as a rest."

— Stephen King, *Hearts in Atlantis* (1999)

The Android permission system has been the focus of several studies over time, as we highlight in Chapter 3.2 (page 30). However, most of these studies focus on its limitations, or specific attacks, but not its evolution over time. One notable exception is the work by Zhauniarovich et al. [ZG16], but this paper was published in 2016, and the Android permission system has gone through significant changes since then. In reality, the Android permission system evolves, either to add new features for device manufacturers or developers or to improve the security and privacy guarantees of the system. This translated into additional permissions, and new *flags*, which allow refining the protection level of a permission. In this chapter, we present an analysis of the temporal evolution of both the overall number of Android permissions (§ 6.1), of protection level and permission flags (§ 6.2), and highlight their impact on the permission granting algorithm in Android (§ 6.3). Finally, we investigate the use of protection level flags by pre-installed apps in the wild (§ 6.4).

6.1 Temporal Analysis of AOSP Permissions

We extract the number of built-in permissions by parsing the manifest file of the open-source framework app [Aosd], which defines those permissions for the system, for each major Android release. The number of permissions kept increasing over time, going from 114 in Android 1.6 (API level 4, released in 2009) to 689 in Android 12 (API level 31, released in 2021). Figure 11 shows the number of permissions per Android version, including the breakdown per annota-

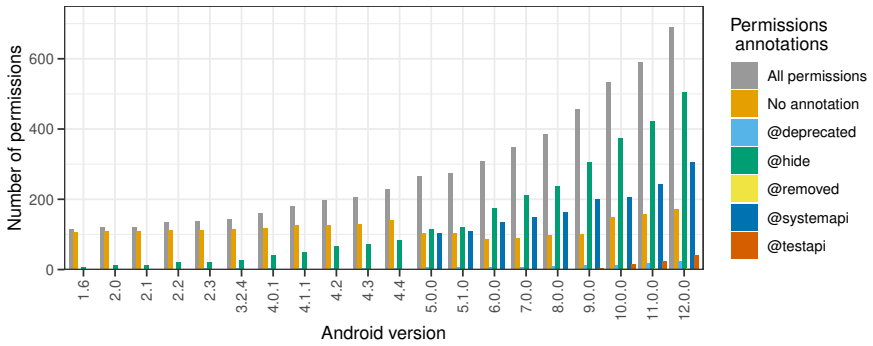


Figure 11: Evolution of the number of AOSP permissions per Android release

tion. Not all of these permissions are available to all developers though: some are marked as "Not for use by third-party apps" in the AOSP source code (e.g., the `READ_LOGS` permission which allows an app to get access to the system log files). In fact, out of the 689 official permissions defined at API level 31, 305 permissions (44% of the total) have the `@SystemAPI` annotation, which indicates that they are reserved for system processes. Furthermore, 506 permissions (73% of the total) have the `@hide` annotation which removes them from the publicly available documentation. Among the other annotations is `@removed` which indicates permissions that do not grant any special privilege anymore, and are kept for backward compatibility only.

The increase in the number of permission is mainly due to the new features provided by the Android OS, to simplify user interactions, or to fix privacy or security vulnerabilities that have been uncovered. For instance, Android 8 introduced a new permission, `ANSWER_PHONE_CALLS`, which allows an app to answer incoming phone calls programmatically [Andd]. Another example is tristate location permissions, introduced in Android 10 [Andc; Andah]. With tristate permissions, users can choose to deny a location permission (either `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION`), to grant it only once, or all the time, instead of the previous two choices, grant or deny. This behavior was further changed in Android 11: it is not possible anymore to grant a location permission "all the time". Instead, users can choose to grant the permission only when the app is running in the background, to prevent apps from abusing the permission and continuously tracking the user's location [Andac; Andaa].

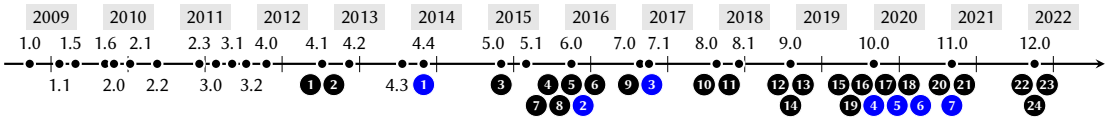


Figure 12: Protection level flags per major Android release. Each black number represents a protection level flag, and each blue number a permission flag.

6.2 Permission Definition Flags

6.2.1 Protection Level Flags

The protection level of a given permission can be further adjusted using *protection level flags*. There are 24 flags defined in the documentation [Andad] (this number includes the `system` flag which is a deprecated synonym of the `privileged` flag). For each flag, we manually inspected the source code of the Android package manager (which is open source) to understand their role. Figure 12 shows the evolution of the number of flags over time (in black):

1. `system` [Dev]: old synonym for `privileged`. It allows a permission to be automatically granted to a system app.
2. `development` [Dev]: `development` permissions can only be granted from the command line using `adb`. The app still has to request the permission in its manifest.
3. `appop` [Appd]: it supports the `AppOp` service, introduced in 2012 [Appe]. This service can be used to monitor "app operations", i.e., what a given app does, and which resources it uses (e.g., for battery usage statistics).
4. `pre23` [Preb]: a permission with this flag will be granted at install time instead of at runtime if the app targets an API level lower than 23 when runtime permissions were introduced [Runb].
5. `installer` [Ver]: a permission with the `installer` flag will be automatically granted to the installer app (i.e., the app responsible for installing packages).
6. `verifier` [Ver]: a permission with the `verifier` flag will be automatically granted to the verifier app (i.e., the app responsible for checking the validity of a package, such as its certificate).
7. `preinstalled` [Prea]: this flag allows any app installed on the system image to be granted the permission automatically.
8. `privileged` [Prea]: this flag replaces the old `system` flag [Dev]. If an app is on a system permission and requests a permission defined with this flag, then it is automatically granted, even if it has a signature base protection type.
9. `setup` [Set]: a permission with this flag will be automatically granted to the setup wizard

app.

10. `ephemeral` [Eph], renamed `instant` in 2017 [Insa]. Instant apps [Insb] can only be granted permissions that have the `instant` flag.
11. `runtime` [Runa]: This flag forces the app to treat the permission as a runtime one, i.e., the app must display a popup to the user to ask for consent. If the app does not support runtime permission, then the permission is not granted.
12. `oem` [Oem]: this flag was introduced to allow original equipment manufacturers (OEM) to define their own permissions for their apps. The permissions must also be present in an allowlist in `/oem/permission/`. To be granted such permissions, the requesting app must be in the OEM partition (in `/oem`) and be signed with the same certificate as the app defining the permission.
13. `vendorPrivileged` [Ven]: these permissions must be explicitly allowlisted by the vendor, by adding them to XML files located in the `/vendor/permission/` folder. In addition, the app requesting the permission must be signed with the same certificate as the app defining the permission.
14. `textClassifier` [Texa]: the text classification is a feature introduced in Android 8.1 to allow developers to use machine learning techniques to sort out text [Texb]. Permissions with a `textClassifier` flag can be automatically granted to the system text classifier app.
15. `wellBeing` [Comi]: permissions with this flag can be granted automatically to the well-being app, as defined by the device manufacturer.
16. `documenter` [Comb]: permissions with this flag can be automatically granted to the document manager app.
17. `configurator` [Comg]: permissions with this flag can be automatically granted to the device configurator app.
18. `incidentReportApprover` [Comh]: permissions with this flag can be automatically granted to the app responsible for sharing incidents and bug reports.
19. `appPredictor` [Comm]: permissions with this flag can be automatically granted to the system's app predictor, a feature introduced in Android 10.
20. `companion` [Comc]: permission with this flag can be automatically granted to the system companion device manager service.
21. `retailDemo` [Comj]: permissions with this flag will be granted to the retail demo app which is defined by the OEM.
22. `recents` [Coma]: permissions with this flag will be granted to the recent apps, i.e., apps

the user used most recently [Rec].

23. `role` [Comk]: permissions with this flag will be managed as `development` permissions are, via `grantRuntimePermission()` or `revokeRuntimePermission()`, but will only be manageable by role.
24. `knownSigner` [Comd]: permissions with this flag can also be granted to any app signed by any of the certificates in the `knownCerts` array [Andae]. This feature was introduced in Android 12 to grant signature level permissions regardless of their actual signature.

6.2.2 Permission Flags

In addition to protection level flags, permissions can use one or more *permission flags*. These permission flags appear in blue in Figure 12. Here too, we manually inspected the source code of the Android package manager to understand the role of permission flags. These are additional flags that are not related to the protection level. They are intended for OEMs to define their own policy to allowlist apps that could be granted such permissions [Perf], or display additional information to the user (e.g., the `costsMoney` flag for a permission that allows an app to send SMS). These flags are not set when defining a permission but are used internally by the package manager. Nevertheless, they can still have an influence on the decision to grant a permission to an app or not, as we detail below.

1. `costsMoney` [Comf]: this indicates that granting this permission may cost the user money (e.g., a permission to allow an app to send SMS).
2. `installed` [Comn]: this flag indicates that the app has been installed into the system's globally defined permissions.
3. `removed` [Come]: this flag indicates that the permissions has been removed from the system, and is not enforced anymore. These permissions are kept for backward compatibility as some apps could be checking if the permission was granted before calling an API.
4. `softRestricted` [Comq]: this flag means that the permission should only be granted to a permission that meets certain conditions, as defined by the system's policy. If these conditions are not met, then only a weaker form of the permission is granted.
5. `hardRestricted` [Comq]: this flag is similar to the `softRestricted` one, but if the app does not meet the system's criteria, then no permission is granted.
6. `immutablyRestricted` [Como]: this flag indicates that the allowlisting state of the permission is evaluated only once at the installation time of the apps.
7. `installerExemptIgnored` [Comp]: this flag prevents the installer app from making an

exception on the restriction of the permission.

6.3 Evolution of the Permission Granting Algorithm

In Section 2.2.2, we explained that the permission granting algorithm mainly depends on the protection level of the permission. In reality, the many protection levels and permission flags that exist thoroughly complicate the algorithm to decide if a permission should be granted to an app or not. From a user perspective, the algorithm is simple: an app that requests access to a permission should do so by asking the user to grant it, or not. However, as we will now see, it is vastly more complicated, and the algorithm keeps gaining in complexity with new Android releases.

Figure 13 summarizes this algorithm, taking into account protect levels and flags, from the perspective of the package manager. Note that this figure shows a *simplified* version of the granting algorithm, as we do not include flags that only apply to a specific category of apps (e.g., `installer` or `configurator`), to avoid overcrowding the figure. The various existing flags make the process much more complicated. Moreover, some of the protection level flags allow for bypassing some of the restrictions put in place by the protection level of a permission. For instance, a pre-installed app could request and be granted any permission that has the `preinstalled` flag, regardless of the protection level of the permission. There are other similar examples, such as the `privileged` flag for system apps installed in a privileged folder of a system partition. Overall, not only is the permission granting process significantly more complex, but it also opens the door to potential security and privacy issues: system apps could take advantage of these flags to get access to permissions they otherwise could not, as we will explore in the next section.

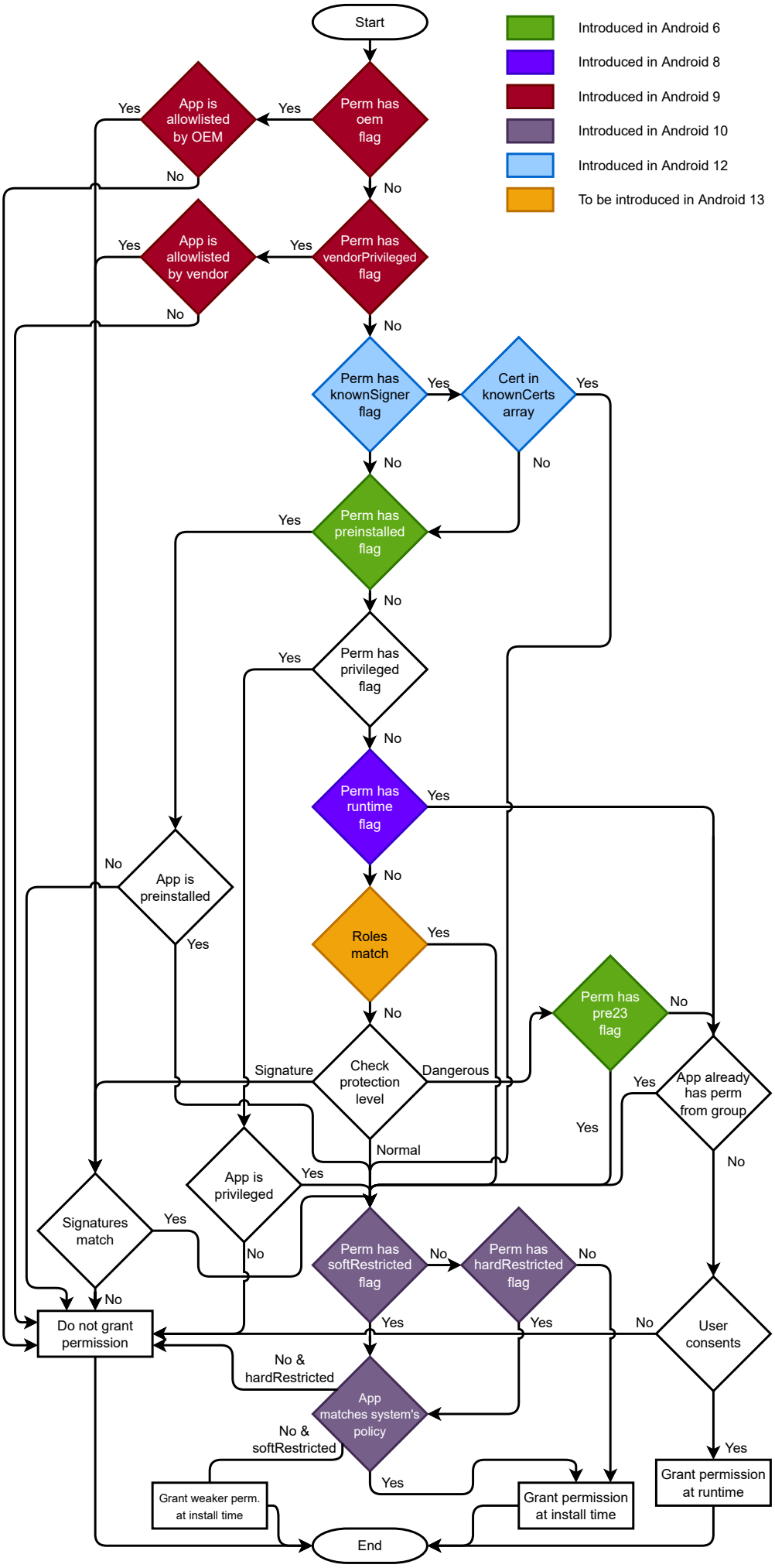


Figure 13: Flow chart of the permission granting algorithm

6.4 Protection Level Flags Usage in the Wild

Table 6.1: Number of unique permission definitions that use either protection level or permission flags, broken down by the vendor of the device on which the defining apps were found (according to the build fingerprint). For readability, we only display the top 10 vendors, and group all the others into the "Others" column.

Flag	Asus	Huawei	Lenovo	Motorola	Oppo	Samsung	Sony	Vivo	Xiaomi	ZTE	Others
❶ system	—	—	—	—	—	—	—	—	—	—	—
❷ development	2	59	—	—	—	628	48	1	1	—	77
❸ appop	8	—	—	74	16	388	—	16	42	6	256
❹ pre23	—	—	—	—	—	—	—	—	—	—	1,230
❺ installer	6	—	—	74	16	366	—	16	42	6	256
❻ verifier	2	—	—	37	8	172	—	8	21	3	128
❼ preinstalled	106	98	11	356	39	1,103	7	287	73	26	1,273
❽ privileged	1,318	30,334	512	6,496	1,603	83,469	675	575	1,847	604	36,049
❾ setup	4	—	—	74	16	344	—	16	42	6	256
❿ instant	94	94	40	94	54	128	44	40	74	54	367
⓫ runtime	—	—	—	—	—	—	—	—	—	—	—
⓬ oem	1	—	—	—	—	11	—	—	—	—	3
⓭ vendorPrivileged	7	—	—	74	16	377	—	16	42	6	259
⓮ textClassifier	—	—	—	—	—	—	—	—	—	—	—
⓯ wellBeing	—	—	—	—	—	—	—	—	—	—	—
⓰ documenter	—	—	—	—	—	—	—	—	—	—	—
⓱ configurator	—	—	—	—	—	—	—	—	—	—	—
⓲ incidentReportApprover	—	—	—	—	—	—	—	—	—	—	—
⓳ appPredictor	1	—	—	37	8	161	—	8	21	3	128
⓴ companion	—	—	—	—	—	—	—	—	—	—	—
㉑ retailDemo	—	—	—	—	—	—	—	—	—	—	—
㉒ recents	—	—	—	—	—	—	—	—	—	—	—
㉓ role	—	—	—	—	—	—	—	—	—	—	—
㉔ knownSigner	—	—	—	—	—	—	—	—	—	—	—

We now investigate the use of those protection level flags in pre-installed apps, by relying on the data collected by Firmware Scanner (Chapter 4, page 41). Specifically, we use the dataset of 983,875 apps found pre-installed on non-rooted devices. For each of those apps, we parse their manifest and extract the protection level of each of the custom permissions they define, by looking for the `android:protectionLevel` attribute of the `<permission>` tag used to define a permission.

Table 6.1 shows the results for the top 10 most common manufacturers in our dataset of pre-installed apps, based on their build fingerprint. We group the other manufacturers in the "Others" column and exclude potentially rooted devices. For each of those vendors, we show the number of permission definitions (i.e., the number of times any pre-installed app defines a custom permission) that make use of at least one protection level flag. Interestingly, we find that half of the flags are in fact never used by any app in our dataset. This includes the most recent ones, namely `recents`, `role`, and `knownSigner` which were introduced in Android 12,

which can explain the current lack of adoption of those flags. Other unused flags include flags reserved for specific categories of apps, such as `incidentReportApprover`, which restrict a permission to only the app responsible for sending bug reports, as defined by the OEM.

We find that the most used protection level flag is by far the `privileged` one. This flag allows any privileged app (i.e., any app installed in the `priv-app/` folder of any system partition) to automatically be granted this permission, regardless of the protection level. Note that this applies to any privileged app, including third-party apps that have been installed there either during the manufacturing of the device or later by a FOTA component. This could potentially create serious security and privacy issue, as such permission would be granted without any user interaction or even awareness.

6.4.1 Custom Permissions Usage by Privileged Apps

Indeed, we find that apps signed by third parties request such custom permissions with the `privileged` flag. Our method is the following. First, we get the set of privileged pre-installed apps of a device on which we find at least one permission using the `privileged` flag. These apps would be granted the permission, should they request it.[1] Then, we use the `Subject` field of the apps' signing certificate to determine whether they are signed by a third party. Specifically, we consider any app that has the vendor name in its `Subject` field to be a first-party app.

Overall, we find that 13% of the pre-installed apps that would be granted custom permissions with the `privileged` are *not* signed by the manufacturer of the device, but rather by a third-party company. This percentage varies significantly within the top 10 manufacturers of our dataset, going from less than 2% for Samsung (0.6%) or Huawei (1%), up to 20% or more for Lenovo (20%), Asus (22%) or ZTE (27%). Interestingly, we find that in the overwhelming majority of the cases (99%), third-party apps request permissions defined by other third-party apps on the device. This could indicate the presence of actors of the supply chain taking advantage of their privileged position on the device to make available features to other apps they installed, or even partnerships between stakeholders.

Some of those cases can easily be explained: the app `com.google.android.setupwizard`, signed by Google, requests the `com.google.android.setupwizard.PARTNER_SETUP` permission with a `signature|privileged` protection level. This particular case would appear as a third-party app requesting a privileged permission. However, we find cases that cannot be

[1] Note that it is theoretically possible for the protection level of the permission to have more than one flag, which could prevent the requesting app from being granted the permission. However, we do not find any such case in our dataset.

explained this easily. For instance, we find a launcher app signed by the Sharp corporation (`jp.co.sharp.android.launcher3.simple`) which requests a permission defined by the app `jp.softbank.mobileid.installer`: `jp.softbank.mobileid.permission.PACK_ACTIVITY` SoftBank Group is a Japanese finance company based in Tokyo which bought Vodafone Japan in 2006 [Sof]. The `com.verizon.settings.permission.RECEIVE_UPDATED_SETTING` permission defined by the framework app of some TCL devices, is another example. Interestingly, in these cases, the defining and requesting apps seem to belong to two different companies.

We could find several other similar examples, which highlight again not only the complexity of the supply chain of Android devices but also the existence of partnerships between the stakeholders. Note that the mere fact that third-party apps are relying on privileged custom permissions does not necessarily imply malicious intent. It does however create a risk for privacy and security issues which warrants further investigation. We will study in detail custom permissions, their usage, and the potential privacy and security risks that they create in Chapter 7.

6.5 Takeaways

In this chapter, we presented an updated view of the evolution of the Android permission system, both in terms of number of permissions but, crucially, in terms of its complexity. The several new features which were introduced in the recent versions of Android vastly complicate the process of granting a permission or not, and potentially allow system apps to share data and resources more easily, including with third-party apps. Overall, this evolution opens the door to more security and privacy abuses which, combined with the lack of control of the supply chain we highlighted in Chapter 5, paints a worrying picture for end users.

Chapter 7

Analyzing Custom Permissions Behaviour

"*HAL: I'm sorry, Dave. I'm afraid I can't do that.*"

— Stanley Kubrick, *2001: A Space Odyssey* (1968)

The Android permission system possess an interesting feature: its extensibility. Developers are allowed to create their own custom permissions, which they can then use to protect the components their apps expose. The majority of previous studies on the permission system overlooked custom permissions and their potential security and privacy risks (see Chapter 3, page 27), with some notable exceptions [Bag+15; Bag+18; Tun+18; Li+21]. In this chapter, we expend and complement the high-level analysis presented in Chapter 5. Specifically, we empirically study the usage of custom permissions at large scale, using a dataset of 2.2M pre-installed and publicly available apps (§7.1). We first study the prevalence of custom permissions in the wild (§7.2), and show that violation of the naming conventions for custom permissions are routinely broken (§7.3). Finally, we seek to understand the purpose of these permissions. We develop tools, `PermissionTracer` and `PermissionTainter`, dedicated to the analysis of custom permissions, and present several cases of potentially harmful behaviors in the wild (§7.4).

7.1 Data Collection

We now describe our data collection and processing methodology.

Table 7.1: Number of unique apps (based on their MD5 hash) and custom permissions per data source, with and without permissions associated with push notification services. We merge the apps downloaded from AndroZoo with their market of origin if we consider said market in our study (e.g., we merge AndroZoo apps downloaded from Google's Play Store into a Google Play set). Otherwise, we group them together as "AndroZoo."

Origin	Number of apps	Number of permissions requested	defined	all
Google Play	638,758	19,464	13,626	22,010
Tencent	94,443	11,610	7,013	12,591
APKMonk	23,774	1,037	402	1,108
Xiaomi Mi	21,613	6,381	3,852	6,838
Baidu	11,522	3,172	1,810	3,358
APK Mirror	9,696	2,106	852	2,246
Huawei	6,655	3,613	2,227	3,895
Qihoo 360	4,321	3,092	1,524	3,251
AndroZoo (other stores)	217,639	9,814	6,195	10,660
Pre-installed	1,247,447	16,886	14,912	19,312
Total	**2,234,506**	**46,556**	**37,743**	**52,468**

7.1.1 Data Sources

We rely on two main data sources to gather a large-scale and representative dataset of Android apps, including ones that came pre-installed, and ones that are published to app markets. These two data sources, actively gathered from 2019 to 2022, are complementary and bring a holistic perspective of apps exposing and requesting custom permissions.

Public app stores. We implemented a purpose-built crawler to download apps and their associated metadata from several public app stores at scale: Google's Play Store [Gplc], Tencent [Tenb], APKMonk [Apkb], Xiaomi's Mi Store [Mis], Baidu [Baib], APK Mirror [Apka], Huawei [Huac], and Qihoo 360 [Qih]. We chose these app stores for their popularity, thus giving us access to a representative picture of the Android ecosystem including and beyond the Play Store [Wan+18b]. We complement this corpus with apps collected by the AndroZoo project [Andr].[1] We gather a total of 987,059 apps (see Table 7.1) through a multi-year crawling campaign started in March 2019 until March 2022.

Pre-installed apps. In this chapter, we use a subset of the data collected by Firmware Scanner (see Chapter 4, page 41 for details). This subset contains 1,247,447 pre-installed apps collected from 58,540 users, representing 17,973 unique device models associated with 783 Original Equipment Manufacturers (OEMs).

[1] We only collect apps from AndroZoo that we do not already have from our other sources to increase our overall coverage.

7.1.2 Methodology for Extracting Custom Permissions

We consider any permission to be custom if it never was in the official list of AOSP permission for any Android release. We therefore start by listing the official AOSP permissions across Android releases, by parsing the manifest of the open-source AOSP framework app [Aosc]. Then, to extract custom permissions, we parse the apps' manifests and extract `<permission>` tags for defined permissions, and both `<uses-permission>` tags and the `android:permission` attributes of permissions protecting apps' components for requested custom permissions. Using this methodology, we obtain 257,710 custom permissions in total, including both defined and requested ones. Alongside the permission name, we also extract metadata related to the app that defined or requests it (e.g., app's package name and signing certificate), and information about the permission itself (e.g., description field and protection level) to further study the adoption of naming conventions, and developers' willingness to document their custom permissions.

Attribution. We leverage Google's naming recommendation as a proxy to identify the party responsible for the definition of custom permissions. For example, `com.foo.PERMISSION` has the second-level domain `foo.com`, which should identify the author of the custom permission. However, it is important to note that developers do not necessarily abide by this convention. There is currently nothing preventing a developer from defining custom permissions with a different package name than its app. Relying on extra signals such as the app's signing certificate does not solve this issue, as apps in Android use self-signed certificates, and previous work showed the existence of apps purposefully using false information in their certificate to impersonate other companies [Gam+20]. Due to the lack of robust mechanisms to do sound attribution of custom permissions, in this section, we rely on the naming convention as the only way to potentially understand who defined a given permission. When available, we rely on online documentation as a reliable source for (1) attributing permissions to app developers or SDKs; and (2) inferring what service or data the permission is protecting. In some cases, we manually inspect the package name of the app and the signing certificate to enhance our attribution process.

Push notification services. Push notifications are messages displayed to the user, either from a local app, or from a remote server even when their app is not running on the device. Developers include a receiver in their app to receive the notifications, which they protect with a custom permission, in order to prevent other apps from intercepting the messages from the remote service. We identified several push notification services from companies such as Xiaomi [Pusg], Amazon [Pusa], and others [Pusf; Pusd; Gcmb; Puse; Pusb; Pusc]. Due to their

widespread use and its well-known and supposedly harmless purposes, we exclude 205,242 permissions associated with such services for the rest of the chapter.[2] After applying this filter, we are left with 52,468 custom permission names, both requested and defined, that we consider in the rest of this paper.

7.1.3 Ethical Considerations

In Section 7.4, we survey app developers making use of custom permissions to better understand their rationale and the reasons why they include them using the developer contact details available on Google Play. We treat this as sensitive data since it might have unexpected consequences, e.g., for their current and future employment. We therefore only report statistical and anonymized data, and do not store any information that could be used to identify a particular developer or company. In both cases, we consulted our data collection protocols with IMDEA Networks' Data Protection Officer (DPO) and received approval from our institutional ethical review board to conduct this survey.

7.2 Prevalence of Custom Permissions

The preliminary results reported in Table 7.1 suggest that there is a significant usage [3] of custom permissions, both requested and defined, regardless of the type of app or its origin. However, these numbers by themselves do not completely convey the scale and complexity of the custom permissions ecosystem, especially the number of actors involved. This section measures how widely defined and requested custom permissions are at the app- and market-level.

7.2.1 Definition of Custom Permissions

Figure 14 shows the increasing number of defined custom permission per API level targeted by the app, i.e., as new Android versions get released. We do not plot the result for apps that target an API level lower than 15, as less than 0.1% of current devices run such an old Android version as of April, 2020 [Andp]. We note that the low number of permissions for API level 30 and up is due to the fact that our dataset only contains 85 apps targeting such API levels, all origins included.

This figure reveals that the usage of custom permission seems to grow over time: between API levels 15 and 25 (both included) it is of 3,463.5 permissions, and 5,938 permissions for API

[2] We note that it is technically possible for an app to create a permission which follows the syntax of a push notification service permission for unrelated and potentially harmful purposes.

[3] For the sake of clarity, we say that an app *uses* a custom permission if it either requests or defines it, and clarify the specific case when needed.

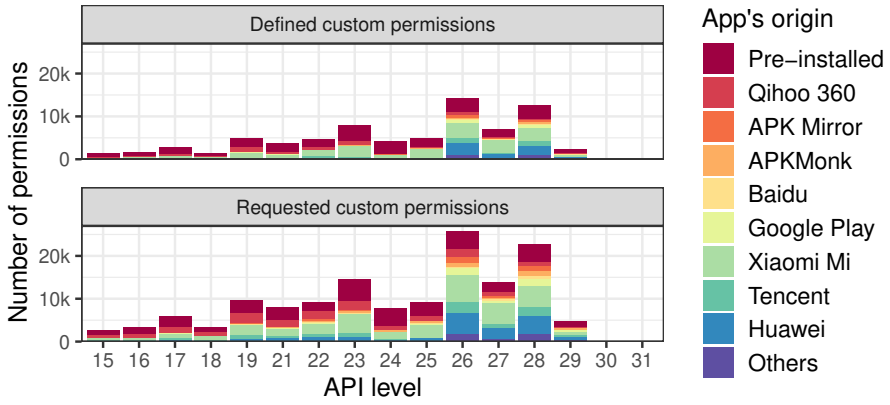


Figure 14: Number of custom permissions requested or defined per target API level, broken down by the origin of the app

levels 26 to 31. We find that the proportion of apps defining custom permissions is much lower than the proportion of apps requesting them: only 4% of apps available on app stores define at least one custom permission versus 26% of pre-installed apps. In fact, the Android Open Source Project allows manufacturers to expose their own features and services to other apps using custom permissions to control the access.

By comparing the device fingerprint reported by Firmware Scanner with the prefixes of the custom permissions exposed by pre-installed apps, we could label 63% as OEM-defined. While all OEMs expose custom permissions in their handsets, Samsung, Huawei and Amazon devices tend to define more than the average. In fact, Samsung is the OEM that exposes the largest number of features with 4,822 permissions. Just Samsung's Knox framework–a pre-installed security framework that offers features like access control, mobile device management, and VPN capabilities [Samb; Knoa; Knob]—is responsible for 109 permissions.

Yet, the reasons why custom permissions exist are diverse as already reported by Gamba et al.[Gam+20]. For example, there are 34% custom permissions related to companies offering third-party SDKs [Ma+16; Raz+18; Fea+21] offering analytics services (e.g., Baidu, AppsFlyer) or social network integration (e.g., Facebook, Twitter), amongst others. For example, the permission `com.twitter.android.permission.AUTH_APP` is used for allowing users of a given app to log in through Twitter, and `com.baidu.permission.BAIDU_LOCATION_SERVICE` is related to Baidu's map services, according to their official documentation. Another interesting app of custom permissions is enabling IoT platform integration. We find 6 custom permissions defined by Amazon to allow app developers to communicate with Amazon devices: e.g., `amazon.speech.permission.SEND_DATA_TO_ALEXA` for Alexa devices [Amad] and `com.amazon.device.permission.COMRADE_CAPABILITIES` for Fire TV [Amae]. We also find 51 custom permissions related to Google's Android for cars [Ands], which allow accessing car-

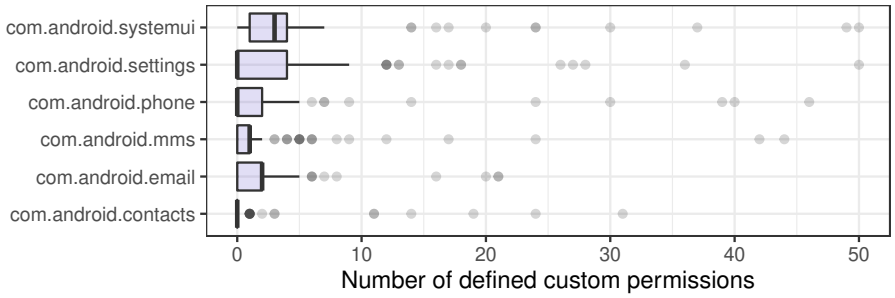


Figure 15: Number of custom permissions defined by core Android components across 783 OEMs and 17,973 device models

specific information such as the speed of the vehicle (`android.car.permission.CAR_SPEED`) or control of the lights (`android.car.permission.CONTROL_CAR_INTERIOR_LIGHTS`).

We also find that core Android components, such as the default dialer app (of package name `com.android.phone`), the system UI app (`com.android.systemui`), or the main framework app (`android`). Such apps are open-source, and can therefore easily be modified by phone manufacturers, to add new features for instance. We show in Figure 15 a boxplot of the number of custom permissions defined by such apps. In general, we find custom permissions defined by all of the core Android components. Note that we do not include the `android` app in this plot for readability reasons: while the median number of custom permissions by the `android` app is only of 1 permission, we find a version on a Samsung device that defines as many as 664 custom permissions. This highlights the level of customization some vendors add to their version of Android, and indicate a potentially high number of features made available by pre-installed apps to other apps, which could include publicly available apps and other third-party apps.

Protection level analysis

As with regular AOSP permissions, custom permissions can set different protections to regulate its access. Figure 16 shows the protection level of defined custom permissions per app type.[4] This figure shows that the most popular protection level used by app developers when defining custom permissions is `signature`, with 39% of the exposed custom permissions on average. When considering exposed custom permissions with a `signatureOrSystem` protection level, this proportion rises up to 86%. This means that the majority of custom permissions will only be granted to apps that share a signing certificate with the defining app as we will study at the

[4]We note that the protection level `signatureOrSystem` is deprecated since API level 23 (Android 6.0) [Andad] and it is semantically equivalent to the `signature` base protection type with the `privileged` flag, which allows an app installed on a system partition to be automatically granted the permission when requested [Prea].Starting in Android 8, a pre-installed app must also be allow-listed by the manufacturer to be granted a permission with the `privileged` flag [Perf].

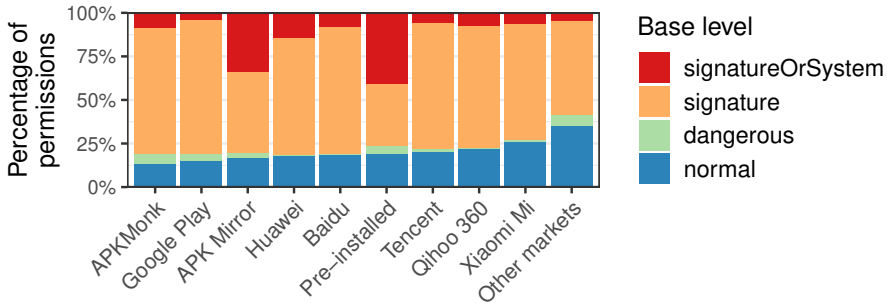


Figure 16: Base protection level usage per origin of the app for defined custom permissions.

end of this section.

More concerning is the fact that 11% of the permissions are defined with the `normal` protection level. In fact, all OEMs expose at least one custom permissions with a `normal` protection level: Motorola, HTC and Xiaomi define a total of 170, 193, and 269 custom permissions with the `normal` protection level. For Samsung, this number goes as high as 867 custom permissions. Therefore, any app installed on the same device will automatically get granted these permissions at installation time and, consequently, to any resource protected by said permissions, unless the developer exposing the permission implements other access control mechanisms programmatically (e.g., by checking the package name of the calling app). Unfortunately, the lack of public information about the actual purpose of these custom permissions (or the type of data or service that they protect) prevents us from automatically assessing their potential risks. Hypothetically, `normal` custom permissions might leave completely unprotected sensitive data as we will further study in Section 7.4.

7.2.2 Requests of Custom Permissions

Figure 14 shows the number of requested custom permissions per target API level and origin of the app. In general, 30% of apps published in public app markets request at least one custom permission but this number is significantly higher (62%) for pre-installed apps. When ranking them by their popularity, we can observe clusters of custom permissions that are significantly more requested than the rest. For example, GMS permissions are requested by more than 10,000 apps and they enable functionalities related to Google Sign-In [Plaa] (`com.google.android.gms.auth.api.signin.permission.REVOCATION_NOTIFICATION`), in-app purchases [Bil] (`com.android.vending.BILLING`), and the Play Install Referrer Library [Fin; Plab] (`com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE`).

Other popular permissions, such as `com.oppo.launcher.permission.READ_SETTINGS` or `com.anddoes.launcher.permission.UPDATE_COUNT` are associated with launcher apps. These last two permissions seem to allow developers to interact with the launchers to display noti-

Table 7.2: Top 20 most requested custom permissions in our dataset, in order. We infer the creator of those permissions using the `Subject` field of the signing certificate of the APKs

Permission name	Creator
com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY	Samsung
com.wssnps.permission.COM_WSSNPS	Samsung
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE	Android
com.sec.android.diagmonagent.permission.DIAGMON	Samsung
com.sec.android.settings.permission.SOFT_RESET	Samsung
com.sec.android.diagmonagent.permission.PROVIDER	Samsung
com.samsung.android.permission.SSRM_NOTIFICATION_PERMISSION	Samsung
com.sec.phone.permission.SEC_FACTORY_PHONE	Samsung
com.google.android.providers.gsf.permission.READ_GSERVICES	Android
com.samsung.android.bixby.agent.permission.APP_SERVICE	Samsung
com.google.android.gms.auth.api.signin.permission.REVOCATION_NOTIFICATION	Android
com.android.vending.BILLING	Android
com.sec.android.app.twdvfs.DVFS_BOOSTER_PERMISSION	Samsung
com.sec.imsservice.PERMISSION	Samsung
com.sec.imsservice.READ_IMS_PERMISSION	Samsung
com.sec.android.provider.logsprovider.permission.READ_LOGS	Samsung
com.sec.android.provider.badge.permission.READ	Samsung
com.samsung.cmh.data.READ	Samsung
com.sec.enterprise.knox.MDM_CONTENT_PROVIDER	Samsung
com.samsung.android.launcher.permission.READ_SETTINGS	Samsung

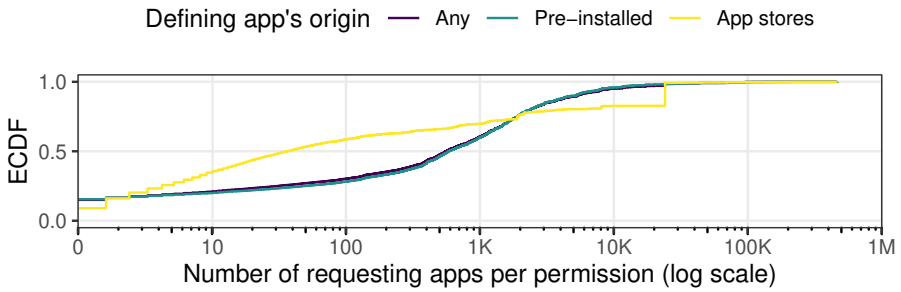


Figure 17: Number of apps requesting custom permissions in our dataset, broken down by the origin of the defining app

fication counts above the icon for their app (e.g., a messenger app could display the number of unread messages in the launcher). Table 7.2 shows the top 20 most requested custom permissions, along to their potential creator (we infer that information from the `Subject` field of the app signing certificate). 4 of these custom permissions seems to be associated with Google (including the 3 we previously presented); all of the others with Samsung. Other less popular cases are Amazon's and Android's Cars permissions, requested by 15,622 and 4,136 apps, respectively. We note, however that while 71% of the apps requesting Amazon's permissions are available on the Google Play Store, 100% of the apps requesting Android Cars permissions are pre-installed.

OEM-specific custom permissions

Figure 17 shows that custom permissions defined by pre-installed apps are likely to be requested by more apps than those defined by publicly available apps. Specifically, the median number of requesting apps per permission is of 36 and 587 for publicly available and pre-installed apps, respectively. The relative popularity of custom permissions defined by privileged pre-installed apps confirms the importance of inspecting potential vulnerabilities on pre-installed apps, as we will further discuss in Section 7.4.

Figure 18 provides a more detailed perspective on how the OEM-specific permissions for the top-10 Android OEMs are requested by publicly available apps.[5] For completeness, we include Google Mobile Services permissions exposed by pre-installed apps on 87% of the devices in our dataset. We group the remaining vendors under the "Others" label. We can infer two things from this figure: (i) a large number of permissions exposed by pre-installed apps are, in fact, requested mainly by pre-installed apps, which could indicate the existence of partnerships between actors of the supply chain of Android devices; and (ii) apps from all app stores do request permissions exposed by device vendors. For example, a total of 43,517 apps in our dataset request Samsung Knox permissions, but 98% of them are other apps pre-loaded on Samsung devices. Those distributed through Google Play and requesting Knox services are mostly professional apps like Cisco Webex, and MDM solutions. This confirms that pre-installed apps are responsible for the exposure of the most widely-demanded permissions, and that this could potentially expose sensitive data and system resources to publicly available apps, either by mistake or intentionally.

Market-level differences

At the market-level, we see that apps published in Xiaomi Mi, Tencent and Huawei app markets are more likely to request custom permissions than apps published in Google Play. We note that some markets are more recent than others,[6] and this might explain why the usage of custom permissions by apps published in Huawei's market is higher in recent API levels. Nevertheless, the definition of custom permissions in Android apps grows with new Android releases: the median number of requested custom permissions between API levels 15 and 25 (both included) is 5,303.5 per API level, while for API levels 26 to 31 (478,244 of all apps in our dataset), the median rises up to 9,550 requested custom permissions per API level.

Table 7.3 shows the most requested permission (grouped by their SLDs) for apps publicly

[5] To measure OEM popularity, we rank them by the number of users with devices of a given OEM in our dataset. We find that the top-10 vendors in our dataset is correlated to publicly available market shares [Appf].

[6] For instance, the Huawei app store was only launched globally in 2018 [Huad]

Table 7.3: Most popular second level domains for custom permissions defined or requested by apps on public app stores

Origin	Most popular SLD	
	requested perms	defined perms
Google Play	google.com	google.com
Tencent	google.com	tencent.com
APKMonk	google.com	sina.com
Xiaomi Mi	permission.android	tencent.com
Baidu	permission.android	lechuan.com
APK Mirror	google.com	google.com
Huawei	permission.android	huawei.com
Qihoo 360	permission.android	tencent.com
Others	permission.android	permission.android

available on these public app stores. This table stresses the popularity of Google permissions, which seems to be the most popular requested permissions in half of the markets we cover. In particular, Google apps present on Google Play (including very popular ones such as YouTube, GMail or the Google Play Services app which is also pre-installed on any Google-certified device) define as many as 183 custom permissions with the `com.google` prefix, which makes the `google.com` SLD group the most popular in that app store. However, we find that the `android.permission` prefix is the most requested permission in app stores from China. For instance, the `android.permission.DOWNLOAD_WITHOUT_NOTIFICATION` permission (which is not part of AOSP) is requested by 5,757 apps on the Baidu app store alone. Permissions seemingly from Google (i.e., in the `google.com` SLD group) are still in the most popular ones in Chinese markets, but followed by well-known Chinese technology companies such as Tencent or Sina Corporation. This table shows a clear geographical divide between app stores located in China in terms of naming conventions and permissions' popularity.

Signature permissions

One final aspect to consider is the link between exposed and requested custom permissions with signature level, as they can be automatically granted during installation time. When a custom permission has a `signature` or `signatureOrSystem` protection level, we check the certificate(s) of both the defining and requesting app, and identify cases where both apps have at least one certificate in common, to reproduce the behavior of the Android OS. In particular, we focus on custom permissions that are defined by pre-installed apps, as those apps are inherently more trusted by the operating system [Gam+20]. Again, we find that custom permissions defined by pre-installed apps are mostly requested (and, in this case, granted) to other pre-installed apps: out of the 586,354 apps that would be granted `signature`

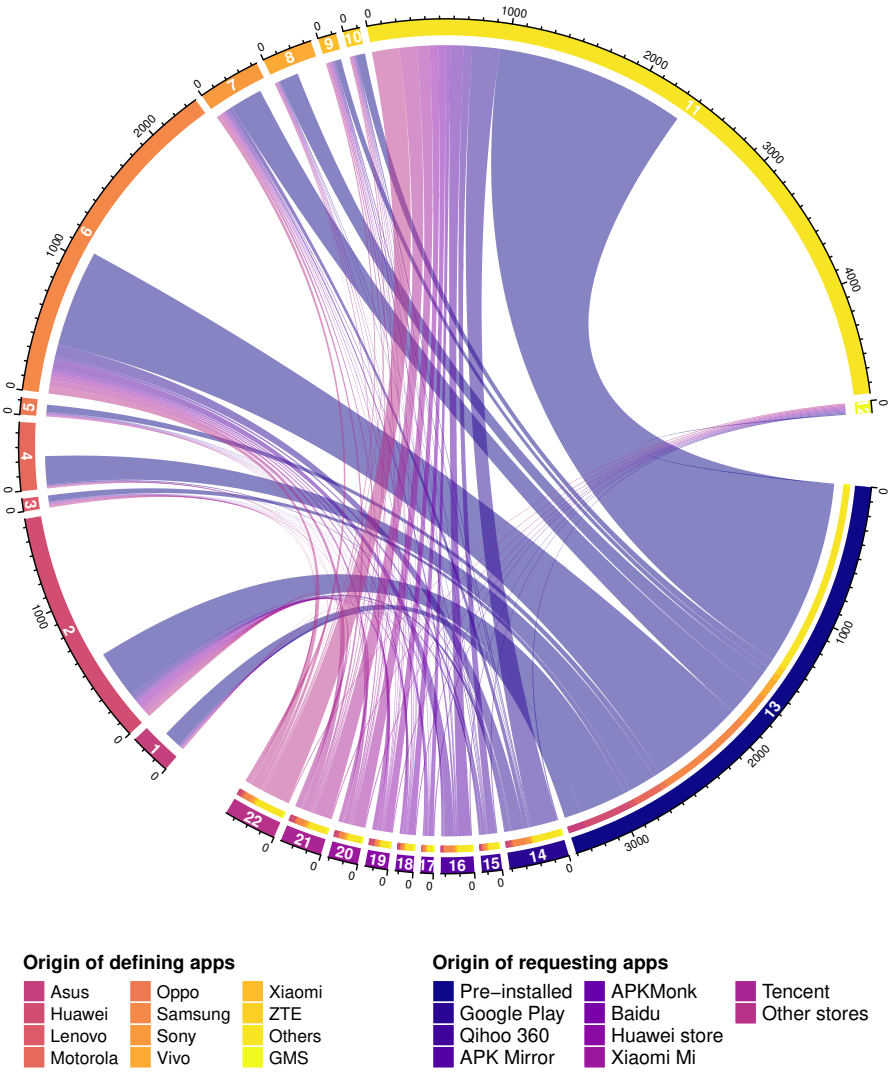


Figure 18: Number of requested permissions defined by pre-installed apps, broken down by the origin of the requesting app (left part) and the vendor for pre-installed apps (right part).

or `signatureOrSystem` permissions, 99.9% of them are pre-installed. We find 13,717 apps (2.3% of the total) on public markets that would also be granted such permissions (note that some apps can be both pre-installed and available on public app stores, Google Chrome for instance). In particular, we find that some Facebook apps, among others, the official Facebook app (`com.facebook.katana`) and Facebook Messenger (`com.facebook.orca`) request custom permissions defined by pre-installed apps sign by the same certificate (hence most likely Facebook apps too). Such permissions include `com.facebook.appmanager.ACCESS` or `com.facebook.receiver.permission.ACCESS`. These permissions are not documented publicly and do not include a description when defined by the Facebook apps. However, such permissions could be related to the partnerships between Facebook and major mobile manufacturers revealed in 2018 by the New York Times [New].

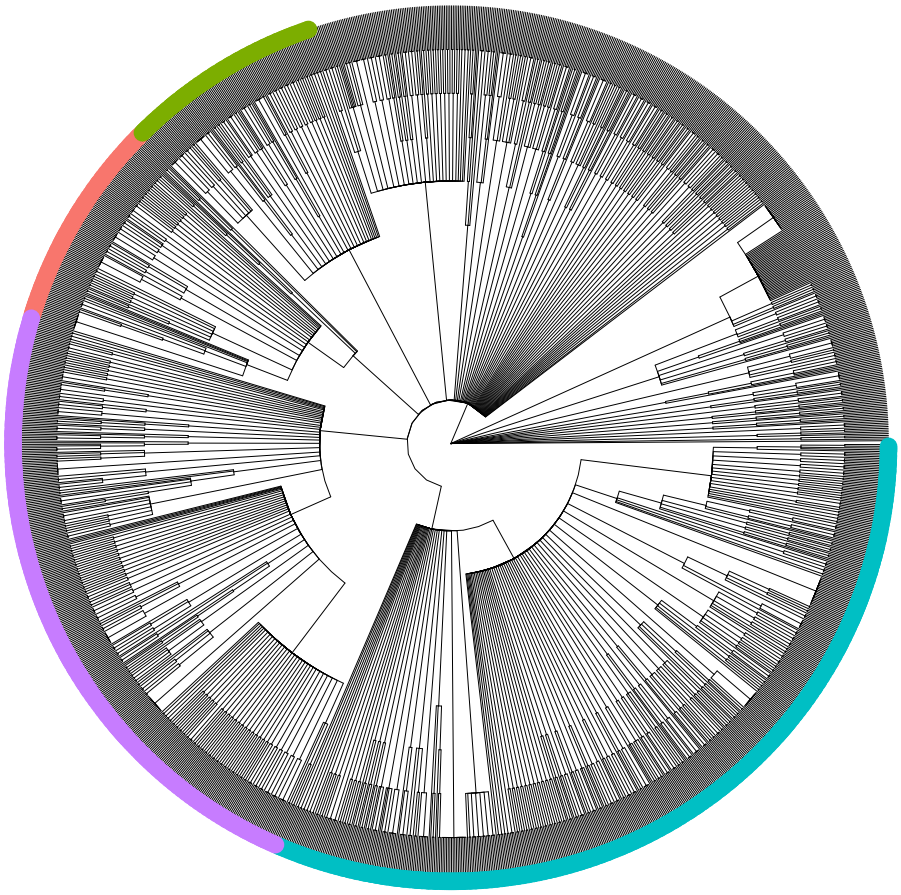


Figure 19: Phylogenetic tree of custom permissions requested by at least 2,000 apps each, grouped by their second level domain. The colors represent the most common SLDs: com.google, com.huawei, com.sec, com.samsung Note that the com.sec prefix might in fact be related to Samsung's Knox API [Knoa]

7.3 Naming and Definition Conventions

While Google recommends app developers defining custom permissions to follow a clear naming convention and adding a description of the purpose of the custom permission, there is no enforcement. Figures 19 and 20 show the scale and complexity of the problem for a subset of custom permissions that are requested by at least 2,000 apps in our dataset. Using the attribution methodology described in Section 7.1.2, we cluster these popular custom permissions into 67 second-level domains (SLDs). However, when considering the totality of our dataset of custom permissions, we find a total of 11,209 SLDs groups, the majority of which (65%) only contain one custom permission, and 94% five or less. Without proper and verifiable naming conventions, nor a clear description of the services and data protected by custom permissions, users have no ability to take informed decisions when granting custom permissions to apps. In fact, any app developer could confuse users by (intentionally) impersonating well-known prefix, such as com.google or com.samsung. In this section, we empirically measure whether

Figure 20: Treemap of custom permissions requested by at least 2,000 apps each, grouped by their second level domain. For readability, we do not include the top 10 most common SLDs. The excluded prefixes seems to be associated with Samsung (`com.samsung`, `com.sec`, `.sec`), Google (`com.google`), Huawei (`com.huawei`, `.huawei`), HTC (`com.htc`), and other entities which we could not identify (`org.adw`, `android.permission`, `com.android`)

app developers exposing permissions follow recommended practices.

7.3.1 Naming Convention Violations

We find naming convention violations to be widespread. Table 7.4 lists the percentage of definitions that fail to adhere to the naming convention, broken down per origin. The percentage of permission definitions that fail to adhere to the naming convention varies from 8% to 33% on public app stores. For pre-installed apps, almost half (47%) of custom permission definitions break the naming convention.

An example of such a violation is the `com.qualcomm.permission.QCOM_AUDIO` permission, defined by the `com.verizon.obdm_permissions` app. Not only are the SLDs of the package name (`qualcomm.com`, a chipset manufacturer) and of the custom permission (`verizon.com`, a network operator) different, but the `Subject` field of the signing certificate of the app mention a third entity, Google. In that case, it is impossible to attribute with certainty the custom

Table 7.4: Number of custom permissions *definitions* that do not follow the naming convention. Note that an app defining multiple custom permissions will be counted multiple times in this table.

Origin	# of definitions	# of bad definitions	Percentage
Google Play	63,193	7,087	11.2%
Tencent	9,902	1,629	16.5%
APKMonk	3,060	298	9.7%
Xiaomi Mi	5,898	1,219	20.7%
Baidu	4,703	612	13%
APK Mirror	19,543	1,654	8.5%
Huawei	3,392	464	13.7%
Qihoo 360	1,999	297	14.9%
AndroZoo (other stores)	28,636	9,478	33.1%
Pre-installed	2,237,585	1,045,815	46.7%
Total	**2,373,124**	**1,067,421**	**45%**

permission to any of these entities.

Some violations are due to developers choosing to use the same prefix as AOSP permissions, which can also confuse the end user into granting a permission, thinking it was created by the operating system, such as `android.permission.DOWNLOAD_WITHOUT_NOTIFICATION`, or `android.permission.RECORD_VIDEO`. In total, we find 722 custom permissions that use the `android.permission` prefix. The high number of permissions using AOSP prefixes is surprising as OEMs are explicitly forbidden from adding permissions in the `android.*` namespace as part as their customization of the OS [Andh]. Yet, we find that 87% of the apps defining at least one of the 722 custom permissions that we identified are pre-installed apps, which could be a breach of the CDD. This issue is still present in recent versions of Android: we find that 226 of these permissions (31% of the total) are defined by apps pre-installed on devices running Android 11 or 12. Anecdotally, we observe instances of apps requesting custom permissions with names that are similar to those of AOSP permissions, but with typos. We find, for instance, custom permissions that include the string `andorid` instead of `android`, `CORSE_LOCATION` instead of `COARSE_LOCATION`, or `RUN_TIME` instead of `RUNTIME`.

We also find evidence that suggests that some naming violations might be due to embedded third-party SDKs or components integrated in the app: if an app embeds an SDK that defines a custom permission, that permission will be in the manifest of the host app (as explained in Chapter 2.2, page 20), and most likely result in a violation of the naming convention (unless both the app and the SDK share the same package name). For instance, the app `com.iugome.lilknights` (a RPG game available on Google Play) defines the permis-

Table 7.5: Percentage of custom permissions definitions (grouped by their SLD or not) without description per app origin

Origin	% of definitions without description	% of SLDs without description
Google Play	82%	75.5%
Tencent	93.9%	91.5%
APKMonk	75.8%	66.2%
Xiaomi Mi	91.5%	87.7%
Baidu	97.8%	96.8%
APK Mirror	73.8%	48.3%
Huawei	96.6%	94.3%
Qihoo 360	95.9%	93.4%
AndroZoo (other stores)	66.5%	59.7%
Pre-installed	69.8%	44.8%
All	75%	47.4%

sion `com.facebook.orca.provider.ACCESS`, which seems to be associated with the Facebook Messenger app. Another more complex example is the `com.mediatek.op12.phone` app which defines the `com.verizon.permission.ACCESS_REMOTE_SIMLOCK` permission. Not only are the SLDs of the package name (`mediatek.com`, a chipset manufacturer) and of the custom permission (`verizon.com`, a network operator) are different, but the signing certificate of the app mention a third entity: TCL, a phone manufacturer. Unfortunately, the lack of developers' compliance and third-party control by app markets defeats any automatic effort to perform accurate attribution of custom permissions to the responsible party.

7.3.2 (Lack of) Documentation for Custom Permissions

One option for trying to better understand custom permissions would be to look at their descriptions on the Android Manifest file. While this is a practice recommended by Google's official documentation[Gpla], it is not mandatory for developers and we find that in 75% of the cases this field is just empty. In table 7.5, we break down the percentage of custom permissions definitions without description by the origin of the apps. We also give the percentage when grouping custom permissions by their SLD. While there is some variation between the origins, we find that apps from all origins tend to lack the description when defining custom permissions.

Even when developers provide a description when defining custom permissions, it is often vague. For example, "Quick connect" or "Dolby Tuning permission description". However, these descriptions lack any actual indication as to what the purpose of the permission is. Yet,

there is no reason to believe that these "descriptions" truthfully and completely describe the behavior and purpose of the custom permissions, especially in the case of those used for nefarious purposes.

In some cases, the suffix of a permission can render useful for inferring their purpose. We find custom permissions that use the exact same suffix as official AOSP permissions, such as `com.oppo.permission.safe.CAMERA` or `thinkyeah.permission.READ_SMS`. In total, we find 142 unique custom permissions with a normal protection level that use the same suffix as a `dangerous` AOSP permission, and 1,334 with a `signature` or `signatureOrSystem` suffix. It is unclear to us why these developers might try to replicate AOSP permissions, and this might suggest that they could provide covert access to AOSP-protected system resources and data. However, this string-based analysis is not conclusive in itself, and requires further investigation.

Finally, we find that online documentation explaining which company is behind a given permission and what is the functionality or data protected is very scarce. In fact, we manually looked for public documentation for the permissions in our dataset using online search engines and do not find publicly available documentation for most of them (94%). This is a highly manual and time-consuming task, and thus we could not realistically manually search for 257,710 permissions. Instead, we rank the permissions by their prevalence and focus our manual efforts on those that are most highly used. For the lesser known permissions, we implement an automatic crawler that relies on the DuckDuckGo API to search for documentation relevant to the permission. Furthermore, we also crawl the StackOverflow forum to find discussions revolving around the permission. Regardless of combining automatic and manual analysis of different resources, we are barely ever able to find any information relevant to a given permissions functionality, showcasing the need for an approach to infer this from the app itself.

7.4 Detecting Leaky Custom Permissions

The main goal of the Android permission system is to protect a set of system APIs from unwanted access without explicit user consent. However, custom permissions also make the Android permission model vulnerable to an elevation of privilege attack, as highlighted by Tuncay et al. [Tun+18] and Bagheri et al. [Bag+15; Bag+18]. In this scenario, we hypothesize that a custom permission can enable access to sensitive data, or to perform an action that is protected by an AOSP permission, and makes it available to other apps via a custom permission that has a lower protection level than the original AOSP permission.

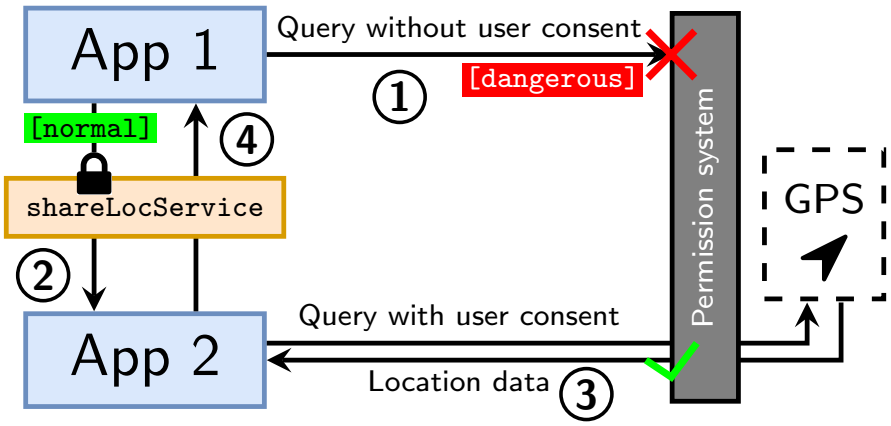


Figure 21: Scenario where an attacker bypasses the permission model using a service protected by a custom permission. The circled numbers indicate the order of each step.

Figure 21 illustrates this situation and involves two apps. app1 first tries to get the user's location through the official API but either lacks the necessary AOSP permission, or the user rejects the request, so it is denied. Then, app1 sends an `Intent` [Int] to the `shareLocService` service exposed by app2. This component is protected by a custom permission that app1 does hold. app2 also holds the AOSP location permission, so it is able to successfully obtain the user's location. app2 then sends back the location to app1 as the response to its `Intent`.

In this particular scenario, app1 and app2 do not necessarily need to cooperate. The result is identical if app2 fails to correctly protect its service, e.g., by giving access to it with a permission that has a `normal` protection level. This creates a vulnerability that an attacker could exploit simply by sending an `Intent` to the service to retrieve the location. In the attack above, the only user interaction that will occur would be at step three, where the OS will display a popup window to ask the user if they wish to allow app2 to access the location. If the user had already granted such a permission to app2, then the attack will play out without any user interaction.

7.4.1 Tooling

Android's custom permissions are asynchronous software artifacts that are difficult to monitor, model, and study. While there is a vast arsenal of highly useful static and dynamic analysis tools to study many harmful and privacy-intrusive behaviors on Android, none of them are fit to effectively infer the purpose of custom permissions and whether they expose sensitive data or system resources. For example, Flowdroid [Arz+14] allows tracking data flows within a given component, but it is unable to handle neither inter-component nor inter-app communication–both of which are essential in the analysis of custom permissions. Similar problems arise from Amandroid (since renamed Argus-SAF) [WRO18] which is able to detect inter-component leaks but it does not detect information leaks between apps through compo-

nents protected by a custom permission. Finally, PScout [Au+12a] focuses on the analysis of permissions by mapping AOSP APIs to AOSP permissions. It is not intended for understanding what these permissions are protecting and cannot be used to determine the purpose of a custom permission. Furthermore, typical analysis challenges such as software obfuscation, dynamic code loading, or deodexing of compiled pre-installed software further complicate the analysis of custom permissions.

To overcome these technical limitations and challenges, we create `PermissionTracer` and `PermissionTainter`, two complementary tools tailored to the analysis of custom permissions:

Tool 1. `PermissionTracer`. We create `PermissionTracer`, a triage tool to extract information about the type of data (or features) protected by custom permissions. Given an app defining custom permissions, the tool analyzes all component protected by it and its methods to report: (i) the data types of return values, and method prototypes that an app can access when interacting with said component; and (ii) the list of APIs protected by AOSP permissions accessed within the component's methods. The ability to extract this knowledge allows determining whether components protected by custom permissions could potentially allow access—by mistake or by design–to restricted data to an app that does not hold the required AOSP permission and which ones might require manual verification. The way `PermissionTracer` analyzes a protected component depends on its type:

- For activities and broadcast receivers, it looks for the `setResult` method and extracts its return data type.
- For content providers (which work as a database for other apps), it obtains the type of the `getType` method.
- For services where no data is returned, it extracts and parses the method prototypes (i.e., method name, return type, and parameter types) from all the interfaces that are returned by the `onBind` method. The type of data (e.g., `String` or Android objects such as `Android.location.Location`) allows understanding the kind of information (e.g., contacts or location) it might expose.

`PermissionTracer` follows a tree search of method calls, parsing the Smali code of a method looking for API calls. This process involves multiple steps. First, `PermissionTracer` extracts and classifies all methods as either *external*, i.e., not defined by the app being analyzed like AOSP calls, or *internal*. For the external calls, `PermissionTracer` looks at whether an AOSP permission is needed to invoke the method using our permission mappings.[7] For inter-

[7]To extract the list of protected AOSP APIs, we update Axplorer's mappings [Bac+16] with (1) mappings provided by Android Studio IDE [Ando]; This includes lint scripts to warn developers if they use certain API calls

nal methods, it adds them to a stack and traverses them recursively once the current method has been analyzed. We limit the stack size to an arbitrary limit of 7 such method calls. To that end, we modify Androguard [Anda] to load our AOSP permission mappings, and to obtain the list of permission protected APIs accessed in a given class. We evaluate `PermissionTracer` by manually inspect the Android components protected by custom permissions of 400 APKs. From those, we manually extract the objects and value types that the components return, and compare this to the output of `PermissionTracer` in the dataset. We do not find any false positive or false negative in the output of our tool. We make our modifications publicly available along with `PermissionTracer`'s code and AOSP permission mappings.

Tool 2. `PermissionTainter`. `PermissionTracer` does not allow discovering potential instances of leaks of data protected by AOSP permissions. In addition to `PermissionTracer`, we build `PermissionTainter`, a static taint analyzer developed to study custom permissions on top of our modified version of Androguard. `PermissionTainter` starts by looking for intent filters that are registered by the app that are not already defined in the app's manifest. Then, it parses the DEX code to look for intents and handlers, and try to associate them with their target. For intents, the target can either be explicitly set by the app, or implicit in the case of broadcasted intents. In the latter case, we use the list of intent filters to determine the classes that would receive such an intent.

After this step, `PermissionTainter` enriches the analysis object created by Androguard (which contains, among other things, all the classes and methods and the cross-references between them) to add extra cross-references to account for asynchronous communications, such as intents. Essentially, `PermissionTainter` is creating a graph representing the whole DEX code, where vertices are methods, and edges are methods calls, which now include asynchronous communications as well.

Finally, `PermissionTainter` relies on an external list of sources and sinks. It also consider any AOSP API protected by an AOSP permission as a source. `PermissionTainter` first locates all call to sinks methods, and, for each occurrence, builds a call graph rooted at that method. It then looks for any call to a source method in that call graph, and extract all paths from the sources to the sink. A path in the call graph would indicates that the value returned by the source method could make its way to the sink. `PermissionTainter` will then follow each path in the call graph and create the corresponding control flow graph (CFG). Again,

without requesting the associated permission. and (2) knowledge extracted from the AOSP source code to see the prototypes of methods that use the `@RequiresPermission` annotation [Req], which indicate the permission(s) that need to be granted to an app in order to invoke a given AOSP method. To the best of our knowledge, we are the first to follow this easy-to-update approach to obtain a more complete and fresh mapping of API calls to AOSP permissions.

`PermissionTainter` looks for all paths from the source to the sink, this time in the CFG, and apply tainting rules to detect potential misuses.

Limitations. Both of our tools suffer from a number of limitations which are common to other static analysis approaches for Android apps. It can not detect calls to protected APIs that are called by other components loaded dynamically during runtime (e.g., using Java's reflection [Jav] or JNI APIs); and while it can tell if a component uses permission-protected APIs, it can not guarantee that the component will ever actually be used in runtime. Moreover, pre-installed apps can use ODEX instead of DEX files, which are stored alongside the APK file. Because of limitations in our data collection strategy, we may miss the ODEX file associated with an APK, preventing us from doing any code analysis. Lastly, our tools cannot detect if an app manually implement access control mechanisms (e.g., by checking the package name of the calling app upon receiving an intent). Such analysis must therefore be conducted manually, after detection of a potential case of abuse.

7.4.2 Results

We run both tools on our dataset of 96,748 unique apps exposing custom permissions to protect 214,943 components.[8] Using `PermissionTracer`, we find that 24,648 of those components (11%) access at least one API protected by an AOSP permission, and 16% of those components access at least one API protected by an AOSP permission with a `dangerous` protection level. This tool allows us to identify the following behaviors:

Sensitive components

When taking into account the protection level of the custom permission protecting these components, we find 1,209 components (over 2,192 apps) use a custom permission with a `normal` protection level. These components are essentially unprotected, as the `normal` protection level allows any app on the device to request and be granted the permission. 55% of these apps are pre-installed. For example:

- 950 of those components access APIs protected by the `READ_PHONE_STATE` permission, which grants access to non resettable device identifiers such as the IMEI until Android 10, which can be used for user tracking [Gooa].
- 497 components access location data protected by the `ACCESS_COARSE_LOCATION` permission, while 422 do access `ACCESS_FINE_LOCATION`.

[8] Note that given the scale of our dataset, we only analyse the latest version of each package and, in the case of pre-loaded apps, we define as unique apps those unique combinations of package names and signing certificates.

- We find 134 components accessing APIs protected by `READ_PRIVILEGED_PHONE_STATE`, which also gives access to unique identifiers, and 58 components accessing APIs protected by `WRITE_SECURE_SETTINGS` which allow for the modification of the system preferences of the device.

Such findings do not necessarily indicate a malicious intent from the developer, but potentially insecure development practices that could be exploited by malicious actors to access AOSP-protected data without user awareness. This is particularly concerning with `normal` custom permissions, which are granted automatically at install time. An example of such a permission is `melons.dialer.permission.CALL_LOG`, defined by a dialer app that was published on Google Play. This permission has a `normal` protection level and protects a content provider that allows other apps to read and delete entries from the call log. The app implements access control simply by checking the package name of the caller app, and only allows queries from package names in a hard-coded list of messenger apps, including some from the same developer. Thus, an attacker just needs to use one of these packages names for their app, and then query the dialer app to read or delete call log entries without requesting the AOSP permission. We tested and verified this vulnerability dynamically with a proof-of-concept app.

We also study in detail the return types of the 3,780 methods that `PermissionTracer` detected. Unsurprisingly, we find that most methods return void, boolean, or integer values (36%, 29% and 16% of the cases, respectively). However, the method returns Android objects in 123 cases.

For instance, the `mobi.maptrek.lite.permission.RECEIVE_LOCATION` permission (with a `normal` protection level) defined by the `mobi.maptrek.lite` app protects a service that defines a `getLocation()` method, which returns a `Landroid/Location/Location` object. Further analysis of the code of the app shows that the service makes the location of the user available to any colluding app that requires the custom permission. The app defining this permission is an offline map app, intended to be used during outdoor activities when the user has no Internet connection. The app is available on Google's Play Store and has been downloaded over 10k times. We verified this attack with a proof-of-concept app, showing that any app can access user location without requesting the official AOSP permission and without the need to interact with the developers of the other app. In 19% of the cases, the methods return a custom object defined by the app itself.

PII leaks

`PermissionTainter` detects 5 potential PII leaks in pre-installed apps. All these apps implement a similar pattern: upon receiving an intent with a specific action (which can be discovered simply by analyzing the source code of the protected component), an attacker can make the component broadcast an intent which contains the Wi-Fi and Bluetooth MAC addresses as extras. We find these apps even in recent Samsung, Asus and LGE devices running Android version 11. We have not found similar behaviors in apps published in app stores. Any colluding app that has the correct intent filter (which can also be simply discovered by analyzing the component's source code) can then receive that intent and get access to the MAC addresses. The MAC addresses can then be used to uniquely identify a user, or can be used to infer their location [Ftc].

Placeholder permissions

We identify 212,277 apps defining custom permissions that are potentially unused, i.e., the permissions is defined but it is never used in the manifest to protect any of the app's components. We name those as *"placeholder permissions"*. The reasons why they are defined remain unknown to us but it might be the result of poor development practices, such as including code obtained from online forums or legacy code from older versions of the app. Yet, it is possible that such apps do not rely on the system's package manager to enforce their permission and chose to do so internally using either `checkPermission`, `enforcePermission`, or one of their variants [Andt].

To detect such cases, we analyze the binaries of these apps to look for calls to these methods. We find stark differences between pre-installed apps, where 51,793 of the apps call one of the methods, and publicly-available apps, where only 149 of the apps do so. Overall, only 51,942 of the apps seem to do dynamic enforcement of custom permissions. Table 7.6 shows the number of apps for which we detected at least one call to `checkPermission`, `enforcePermission` or one of their variants [Andt] in the DEX or ODEX code of apps that are defined but do not protect any component. We grouped together all apps collected from public app stores or from AndroZoo under the "Public apps" category.

To gain a better understanding of why so many app developers define custom permissions but do not protect any component with it (nor enforce them dynamically), we contacted 529 developers using the contact email address listed in the public profile of their apps. We discuss the ethical considerations and IRB approval in §7.1. Our survey received 53 responses. Surprisingly, 28% of the developers that responded to us either did not know that their app defined

Table 7.6: Number of apps defining placeholder permissions and apps dynamically enforcing custom permissions broken down by dataset of origin.

Origin	# placeholder apps	# calling `check*`	# calling `enforce*`	# calling any
Pre-installed	189,177	45,889	5,771	51,793
Public apps	23,143	149	8	149
Total	**212,277**	**5,779**	**46,038**	**51,942**

a custom permission or they did not know why it was there. In 17% of the cases, an SDK used by the developer added the permission. In 9% of the cases, the permission was associated with an old feature that had been already removed.

Although the scale of our survey is small, it provides some intriguing perspectives on some of the reasons behind the widespread usage of custom permissions. The responses suggest a poor understanding of the (custom) permission system by a fraction of the developers, which could negatively impact users by inadvertently exposing sensitive data or resources.

7.5 Takeaways

In this chapter, we presented a holistic view of the prevalence of custom permissions in the Android ecosystem and their inherent transparency, security and privacy problems. Our findings suggest that, despite this being a widely used feature in both pre-installed and publicly available apps, custom permissions lack transparency, accountability, and it is the source of potential security and privacy harm for end users. In an effort to foster more research efforts in this area, we make available our dataset of custom permissions [Datb; Data], as well as the source code of our tools, `PermissionTracer` [Perd] and `PermissionTainter` [Pere], to the research community, platform operators, and regulators.

III

Conclusions and Open Issues

Chapter 8

Discussion

"*That's mortals for you, Death continued. They've only got a few years in this world and they spend them all in making things complicated for themselves. Fascinating.*"

— Terry Pratchett, *Mort* (1987)

In this thesis, we systematically studied, at scale, the vast and unexplored ecosystem of pre-installed Android software and its potential impact on consumers. In Chapter 5, we have made clear that, thanks in large part to the open-source nature of the Android platform and the complexity of its supply chain, organizations of various kinds and sizes have the ability to embed their software in custom Android firmware versions. The myriad of actors involved in the development of pre-installed software and the supply chain range from hardware manufacturers to MNOs and third-party advertising and tracking services. These actors have privileged access to system resources through their presence in pre-installed apps but also as third-party libraries embedded in them. Potential partnerships made behind closed doors between stakeholders may have made user data a commodity before users purchase their devices or decide to install software of their own. Then, we have shown in Chapter 6 how the Android permission system has evolved over time, how it gained in complexity, and how pre-installed apps make use of some of its feature and the potential security and privacy issues that stem from this usage. Finally, we have uncovered in Chapter 7 several problems inherent to custom permissions. As they are, custom permissions open various avenues for abuse, an issue which is compounded by a severe lack of transparency in the app ecosystem of Android.

As we demonstrated in this thesis, this situation has become a peril to users' privacy and even security due to an abuse of privilege such as in the case of pre-installed malware, or as a result of poor software engineering practices that introduce vulnerabilities and dangerous

backdoors. Google, both as the platform operator and the main driving force behind the Android open source project, is in a privileged position to mitigate the issues we reported in this thesis. In this chapter, we first discuss strategies to address issues related to attribution in Android (§8.1) and the privilege escalations made possible by exposed components and custom permissions (§8.2). Then, we discuss transparency and user control mechanisms (§8.3), and the various consumer protection regulations in place that could have an impact on privacy in the Android ecosystem (§8.4). We conclude this chapter with recommendations to address the issues presented in this thesis (§8.5).

8.1 Attribution and Accountability

The attribution problem in Android is rooted in the absence of a reliable way of tracing an app back to its developer. Due to a lack of central authority or trust system to allow verification and attribution of the self-signed certificates that are used to sign apps, and due to a lack of any mechanism to identify the purpose and legitimacy of many of these apps and custom permissions, it is difficult to attribute unwanted and harmful app behaviors to the party or parties responsible. This has broader negative implications for accountability and liability in this ecosystem as a whole.

Signing certificates are a logical candidate to use for attribution, but their rather poor usage in the wild currently makes them unreliable as an attribution signal due to the widespread use of self-signed certificates, which often lack any valid or useful information, such as the case of apps using debug signing certificates as we highlight in Chapter 5. One potential solution to this problem would be for Google to require app developers to take ownership of their apps through a centralized public key infrastructure. This, in turn, allows users to know the true developer of the apps, as well as the entity that exposes the custom permission to other apps (which itself could be an embedded third-party component). Additionally, custom permissions could have a *definer* tag to their definitions so that a user would always know who is the actor behind a given custom permission as in the case of permissions defined by third-party components embedded in the app.

8.2 Privilege Escalation

Fixing privilege escalation issues arising from features exposed by system apps or custom permissions is complex, as it exploits the customization process which is a central part of the Android ecosystem. One approach to tackle them would be to determine the AOSP permis-

sions being used in the exposed component to perform a risk assessment using a tool such as `PermissionTracer`. Note that a dangerous permission might, nonetheless, be used within a component without exposing the data protected by it. Such false positives could be weeded out by using taint analysis (as we do using `PermissionTainter`) to automatically track protected data and figure out whether or not such data is being exfiltrated. We believe that the ability to automatically prevent potential attacks justifies instances in which the platform enforces a higher permission level (e.g., dangerous) for a custom permission than the originally necessary (e.g., normal). This enforcement can be done automatically by analyzing the app's code and it could be introduced as part of the analysis processes implemented in Google Play Protect [Plac], the built-in security mechanism present on Android devices and in the Google Play Store.

8.3 Transparency and User Control

In the meantime regular Android users are, by and large, unaware of the presence of most of the software that comes pre-installed on their Android devices and their associated privacy risks. Users are clueless about the various data-sharing relationships and partnerships that exist between companies that have a hand in deciding what comes pre-installed on their phones. Custom permissions, for instance, are only shown to the user if the developer creates them with a `dangerous` protection level, despite their potential usage for data dissemination and their risks for end users privacy and security. Even when such permissions are displayed, the developer-provided description usually does not contain any meaningful information as to the actual role of a given custom permission, as we highlight in Chapter 7. Users' activities, personal data, and habits may be constantly monitored by stakeholders that many users may have never heard of, let alone consented to collect their data. We have demonstrated instances of devices being backdoored by companies with the ability to root and remotely control devices without user awareness, and install apps through targeted monetization and user-acquisition campaigns. Even if users decide to stop or delete some of these apps, they will not be able to do so since many of them are core Android services, and others cannot be permanently removed by the user without root privileges. It is unclear if the users have actually consented to these practices, or if they were informed about them before using the devices (i.e., on the first boot) in the first place. To clarify this, we acquired (in 2017) 6 popular brand-new Android devices from vendors including Nokia, Sony, LG, and Huawei from a large Spanish retailer. When booting them for the first time, 3 devices did not present a privacy policy at all, only the Android terms of service. The rest rendered a privacy policy that only mentions that they collect data about

the user, including PII such as the IMEI for added value services. Only one mentioned that the device should not be used by minors due to aggressive data collection practices. Note that users have no choice but to accept Android's terms of service, as well as the manufacturer's one if presented at all to the user. Otherwise Android will simply stop booting, which will effectively make the device unusable.

8.4 Consumer Protection Regulations

While some jurisdictions have very few regulations governing online tracking and data collection, there have been several movements to regulate and control these practices, such as the GDPR in the EU [Gdp], and California's CCPA [Ccp] in the US. While these efforts are certainly helpful in regulating the rampant invasion of users' privacy in the mobile world, they have a long way to go. Most mobile devices still lack a clear and meaningful mechanism to obtain informed consent, which is a potential violation of the GDPR. In fact, many of the pre-installed ATSes may not be COPPA-compliant [Rey+18],[1] even though many minors in the US use mobile devices with pre-installed software that engage in data collection. This indicates that even in jurisdictions with strict privacy and consumer protection laws, there still remains a large gap between what is done in practice and the enforcement capabilities of the agencies appointed to uphold the law.

8.5 Recommendations

To address the issues mentioned above and to make the ecosystem more transparent we propose a number of recommendations. These suggestions are made under the assumption that stakeholders are willing to self-regulate and to enhance the status quo. We are aware that some of these suggestions may inevitably not align with corporate interests of every organization in the supply chain, and that an independent third party may be needed to audit the process. Google might be a prime candidate for it given its capacity for licensing vendors and its certification programs. Alternatively, in absence of self-regulation, governments and regulatory bodies could step in and enact regulations and execute enforcement actions that wrest back some of the control from the various actors in the supply chain. We also propose a number of actions that would help independent investigators to detect deceptive and potentially harmful behaviors.

[1] The Children's Online Privacy Protection Act of 1998 (COPPA) is a US federal law to protect minors under 13 years of age from online tracking without "verifiable consent" from a parent or legal guardian [Cop]

8.5.1 Attribution and Accountability

To combat the difficulty in attribution and the resulting lack of accountability, we propose the introduction and use of certificates that are signed by globally-trusted certificate authorities. Alternatively, it may be possible to build a certificate transparency repository dedicated to providing details and attribution for self-signed certificates used to sign various Android apps, including pre-installed ones.

In addition, requiring app developers to include a list of their defined custom permissions along with a better description of the purpose and the potential associated risks is an important and much-needed first step to improve transparency, promote user awareness, and empower user control. We understand that developers can still be obscure or deceitful in describing the purpose of a permission, but we argue that if all developers were forced to add a more informative description to their permissions, users would be more likely to grant access to well-explained permissions over those that are unclear about the functionality that they expose. Moreover, developers with a legitimate reason to create a custom permission or expose a specific feature will most likely comply with such a rule by providing a clear description.

To make this more effective, we suggest extending the description with a mandatory risk self-assessment done by the developer. Such assessment might consist of a few key questions with a set of predefined answers regarding the data and features accessed or shared by the permission. Software distribution channels can verify and enforce permission description sanity, at least at a basic level, to ensure the system is not cheated. Furthermore, this could be a way to ensure that developers do not define custom permissions that are unnecessary, reinforcing the practices already implemented by Google to encourage developers to minimize the access to sensitive permissions via permission nudges [Ped+19].

8.5.2 Accessible Documentation and Consent Forms

Android devices can be required to document the specific set of apps that are pre-installed, along with their purpose and the entity responsible for each piece of software, in a manner that is accessible and understandable to the users. This will ensure that at least a reference point exists for users (and regulators) to find accurate information about pre-installed apps and their practices. Moreover, the results of our small-scale survey of consent forms of some Android vendors leave a lot to be desired from a transparency perspective: users are not clearly informed about third-party software that is installed on their devices, including embedded third-party tracking and advertising services, the types of data they collect from them by default, and the partnerships that allow personal data to be shared over the Internet. This necessitates

a new form of privacy policy suitable for pre-installed apps to be defined (and enforced) to ensure that such practices are at least communicated to the user in a clear and accessible way. This should be accompanied by mechanisms to enable users to make informed decisions about how or whether to use such devices without having to root them.

Another step in the right direction would be to inform users about the custom permissions requested and defined by an app. At the time of this writing, a custom permission is only shown to the user when requested, and only if the developer decides to give it a dangerous protection level, which essentially allows nefarious developers to stay hidden if they so choose. The risk self-assessment discussed above should be the basis to convey the information effectively. The replies to the set of questions could be leveraged to automatically decide the protection level of the custom permission, instead of leaving this decision to the developer. Finally, the platform should offer users a mechanism to revoke previously granted custom permissions, both individually for a particular app or globally within the system through a blocklist.

Chapter 9

Conclusion

"It is the nature of science that answers automatically pose new and more subtle questions"

— Isaac Asimov, *The Wellsprings of Life* (1960)

In this thesis, I demonstrated the opacity and complexity of the supply chain of modern Android devices, and the level of customization that each stakeholder brings to each device. While some of this customization is due to product differentiation, the opacity and absence of control over the whole process clearly endangers the privacy and security of the Android operating system. This thesis present the first large-scale systematic analysis of Android customization. Our work was recognized by some data protection agencies (DPA), the CNIL (French DPA) and the AEPD (Spanish DPA) in particular. Upon publication of our paper on pre-installed apps [Gam+20], the AEPD wrote a press release to help disseminate our results towards regulators [Aep]. Thanks to this, we were invited to present our results to the European Data Protection Board (EDPB). In this chapter, I highlight the key contributions from this thesis (§9.1), and outline possible future research directions in this area (§9.2).

9.1 Contributions

9.1.1 The Android Pre-installed Apps Ecosystem

In Chapter 5, I presented the first large-scale study of pre-installed software on Android devices. Our work relies on a large dataset of real-world Android firmware that I acquired worldwide using crowd-sourcing methods. This allows us to answer questions related to the stakeholders involved in the supply chain, from device manufacturers and mobile network oper-

ators to third-party organizations like advertising and tracking services, and social network platforms. I also uncovered relationships between these actors, which seem to revolve primarily around advertising and data-driven services. Overall, I demonstrate that the supply chain that results of Android's open source model lacks transparency and has facilitated potentially harmful behaviors and backdoored access to sensitive data and services without user consent or awareness.

9.1.2 Evolution of the Android Permission System

The permission system has long been the focus of the research community, especially in the use, enforcement, and usability of AOSP permissions, which revealed severe privacy and security shortcomings inherent to the Android permission model (see Chapter 3.2, page 30). In Chapter 6, I presented the temporal evolution of the Android permission system. I showed how additional features of AOSP translated into new permissions, but also new permission flags and protection level flags. Notably, I studied the impact of these additions over time over the process of granting a permission to a given app. I also showed how pre-installed apps, specifically from third-party sources, make use of these features of the permission system and the potential impact on the privacy and security of users.

9.1.3 Android Custom Permissions

In Chapter 7, I presented a holistic view of the prevalence of custom permissions in the Android permission system, and the problems that they bring from the point of view of transparency, security and privacy. Our findings show that, despite being a widely used feature in both pre-installed and publicly available apps, custom permissions lack transparency, accountability, and could be the source of potential security and privacy harm for end users. To detect such privacy and security issues, I created two custom-made tools: (1) `PermissionTracer`, a tool that reports potentially-dangerous custom permissions and detects potential cases of a privilege escalation attack in which an attacker can access permission-protected information using custom permissions; and (2) `PermissionTainter`, a static taint analysis tool that inspects the DEX code of apps that define custom permissions to identify potential privacy leaks due to those permissions. I make both these tools freely available to the community, along with our dataset of custom permissions.

9.2 Open Issues and Future Work

In an effort to encourage more research efforts in the area of Android customization and the supply chain issues, I make the tools I developed and some data freely available to the community on our website, the Android Observatory [Andn]. The dataset of pre-installed apps is also available on demand. In this section, I highlight some possible research directions that steam from our results.

9.2.1 Android Framework Customization

In our work, I only considered customization of the OS implemented by pre-installing extra apps on the system partitions on a device. However, there is nothing preventing a phone manufacturer to modify core Android components, or the OS directly, as long as they respect the limits set by the CDD. I have shown examples of such customization in Chapter 7 (page 93) in the case of custom permissions, or the addition of extra root certificates in the system in Chapter 5 (page 47), which was also highlighted in previous work [VR+14]. Some recent studies already present evidence of privacy invasive behavior in core components [Lei21; LPL21; Lei22]. This studies are focused on major vendors though, and do not give a global overview of the issue. Such a study could also potentially help shed light on the supply chain, by looking at the recipient of network flows.

9.2.2 Native Libraries

Every Android device includes native libraries that provide features for the OS (e.g., hardware drivers). Native libraries can have the same role as pre-installed apps, and be a used as a customization vector for stakeholders of the supply chain. The study of such libraries poses another set of challenges, as they require new tools to be analyzed: native libraries are (usually) ELF objects, and cannot be analyzed using state-of-the-art Android analysis tools.

9.2.3 Dynamic Analysis

The dynamic analysis of Android apps remains challenging, especially at scale, as this requires realistic emulation of user input, and more time than for static analysis. For pre-installed apps, there are additional challenges. Pre-installed apps developers know in advance the environment in which their app will run, and can therefore rely on other apps or specific hardware, which would not be present in an emulated environment. Recent studies used dynamic analysis to conduct privacy analysis of core components and devices as a whole [Lei21; LPL21;

Lei22], but such studies rely on the actual hardware which has obvious scaling limitations. However, Pustogarov et al. has already shown that it is possible to emulate the behavior of hardware dependencies without having access to the physical device [PWL20b]. It should therefore be possible to build and expand this approach to emulate the complete environment an app expects, and build a dynamic analysis environment for pre-installed apps.

Bibliography

[Aaf+15] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. "Hare Hunting In The Wild Android: A Study On The Threat Of Hanging Attribute References". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2015.

[AHIN14] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. "A New Sensors-based Covert Channel on Android". In: *The Scientific World Journal* (2014).

[All+16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. "AndroZoo: Collecting Millions of Android Apps for the Research Community". In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2016.

[Ama+15] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M Memon. "Exploiting the Saturation Effect in Automatic Random Testing of Android Applications". In: *ACM International Conference on Mobile Software Engineering and Systems*. 2015.

[Arz+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps". In: *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN)* (2014).

[Au+12a] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. "PScout: Analyzing the Android Permission Specification". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2012.

[Au+12b] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. "PScout: Analyzing The Android Permission Specification". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2012.

[Aut+21] Marco Autili, Ivano Malavolta, Alexander Perucci, Gian Luca Scoccia, and Roberto Verdecchia. "Software Engineering Techniques for Statically Analyzing Mobile Apps: Research Trends, Characteristics, and Potential For Industrial Adoption". In: *Journal of Internet Services and Applications* (2021).

[Avd+15] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. "Mining Apps for Abnormal Usage of Sensitive Data". In: *Proceedings of the International Conference on Software Engineering*. 2015.

[Bac+16] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. "On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis". In: *Proceedings of the USENIX Security Symposium*. 2016.

[Bag+15] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. "Detection of design flaws in the android permission protocol through bounded verification". In: *Proceedings of the International Symposium on Formal Methods*. 2015.

[Bag+18] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. "A formal approach for detection of security flaws in the android permission system". In: *Formal Aspects of Computing* (2018).

[Bar+12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot". In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. 2012.

[Bha+17] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. "Android Inter-App Communication Threats and Detection Techniques". In: *Computers & Security* (2017).

[Blá+21] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. "Trouble Over-The-Air: An Analysis of FOTA Apps in the Android Ecosystem". In: *IEEE Symposium on Security and Privacy (SP)*. 2021.

[BNN17] Kenneth Block, Sashank Narain, and Guevara Noubir. "An Autonomic and Permissionless Android Covert Channel". In: *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. 2017.

[Bug+11] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. "Xmandroid: A New Android Evolution to Mitigate Privilege Escalation Attacks". In: *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[Bug+12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2012.

[Cal+20] Paolo Calciati, Konstantin Kuznetsov, Alessandra Gorla, and Andreas Zeller. "Automatically Granted Permissions in Android Apps: An Empirical Study on Their Prevalence and on the Potential Threats for Privacy". In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2020.

[Chi+17] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. "Does this app really need my location? Context-aware privacy management for smartphones". In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* (2017).

[Des14] Luke Deshotels. "Inaudible Sound as a Covert Channel in Mobile Devices". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2014.

[Die+11] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. "Quire: Lightweight Provenance for Smart Phone Operating Systems". In: *Proceedings of the USENIX Security Symposium*. 2011.

[DK12] David Dittrich and Erin Kenneally. "The Menlo Report: Ethical principles guiding information and communication technology research".
In: *US Department of Homeland Security* (2012).

[Els+20] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin.
"{FIRMSCOPE}: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware".
In: *Proceedings of the USENIX Security Symposium*. 2020.

[Enc+14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones".
In: *ACM Transactions on Computer Systems (TOCS)* (2014).

[FCF09] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster.
"ScanDroid: Automated Security Certification of Android Applications".
In: *Manuscript, University of Maryland* (2009).

[Fea+21] Álvaro Feal, Julien Gamba, Juan Tapiador, Primal Wijesekera, Joel Reardon, Serge Egelman, and Narseo Vallina-Rodriguez.
"Don't Accept Candy from Strangers: An Analysis of Third-Party Mobile SDKs".
In: *Data Protection and Privacy: Data Protection and Artificial Intelligence* (2021).

[Fel+11a] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android Permissions Demystified". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2011.

[Fel+11b] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android Permissions Demystified". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2011.

[Fel+11c] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin.
"Permission Re-Delegation: Attacks and Defenses".
In: *Proceedings of the USENIX Security Symposium*. 2011.

[Fel+12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner.

"Android Permissions: User Attention, Comprehension, and Behavior".
In: *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. 2012.

[FGW11] Adrienne Porter Felt, Kate Greenwood, and David Wagner.
"The effectiveness of application permissions".
In: *Proceedings of the USENIX conference on Web application development*. 2011.

[Gam+20] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez.
"An Analysis of Pre-installed Android Software".
In: *IEEE Symposium on Security and Privacy (SP)* (2020).

[Gib+12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen.
"AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale".
In: *International Conference on Trust and Trustworthy Computing*. 2012.

[GK14] David Sounthiraraj Justin Sahs Garret Greenwood and Zhiqiang Lin Latifur Khan. "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2014.

[Gor+15] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard.
"Information Flow Analysis of Android Applications in DroidSafe". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2015.

[Gra+12] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang.
"Systematic Detection of Capability Leaks in Stock Android Smartphones". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2012.

[He+19] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang.
"Dynamic Privacy Leakage Analysis of Android Third-Party Libraries".
In: *Journal of Information Security and Applications* (2019).

[Ikr+16] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. "An Analysis of the Privacy and Security Risks of Android VPN Permission-Enabled Apps".
In: *Proceedings of the Internet Measurement Conference (IMC)*. 2016.

[Jeo+12] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein.
"Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications".
In: *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. 2012.

[Ji+21] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou.
"DEFInit: An Analysis of Exposed Android Init Routines".
In: *Proceedings of the USENIX Security Symposium*. 2021.

[Jin+16] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. "Checking Intent-Based Communication in Android with Intent Space Analysis".
In: *Proceedings of the ACM symposium on Information, computer and communications security (ASIA CCS)*. 2016.

[JMF12] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster.
SymDroid: Symbolic Execution for Dalvik Bytecode. Tech. rep. 2012.

[Joh+12] Ryan Johnson, Zhaohui Wang, Corey Gagnon, and Angelos Stavrou.
"Analysis of Android Applications' Permissions". In: *International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*. 2012.

[KGC13] Kristen Kennedy, Eric Gustafson, and Hao Chen.
"Quantifying the effects of removing permissions from android applications".
In: *Mobile Security Technologies (MoST)*. 2013.

[Kim+12] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center.
"ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications".
In: *MoST* (2012).

[Kli+14] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer.
"Android Taint Flow Analysis for App Sets".
In: *Proceedings of the ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. 2014.

[KZM17] Babu Khadiranaikar, Pavol Zavarsky, and Yasir Malik.
"Improving Android Application Security for Intent Based Attacks".
In: *IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 2017.

[Lau+20] Billy Lau, Jiexin Zhang, Alastair R Bereford, Daniel Thomas, and René Mayrhofer. "Uraniborg's Device Preloaded App Risks Scoring Metrics". In: *Institute of Networks and Security: Linz, Austria* (2020).

[Lee+21] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. "PolyScope:Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems". In: *Proceedings of the USENIX Security Symposium*. 2021.

[Lei21] Douglas J Leith. "Mobile Handset Privacy: Measuring The Data iOS and Android Send to Apple And Google". In: *International Conference on Security and Privacy in Communication Systems*. 2021.

[Lei22] Douglas J Leith. *What Data Do The Google Dialer and Messages Apps On Android Send to Google?* Tech. rep. Trinity College Dublin, 2022.

[Leo+12] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. "Don't kill my ads! balancing privacy in an ad-supported mobile application market". In: *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*. 2012, pp. 1–6.

[Li+14] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. "Automatically Exploiting Potential Component Leaks in Android Applications". In: *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. 2014.

[Li+15] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps". In: *Proceedings of the International Conference on Software Engineering*. 2015.

[Li+16] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. "An Investigation into the Use of Common Libraries in Android Apps". In: *The IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. 2016.

[Li+17] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. "Static Analysis of Android Apps: A Systematic Literature Review". In: *Information and Software Technology* (2017).

[Li+21] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. "Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings". In: (2021).

[Liu+15] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. "Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps". In: *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2015.

[LPL21] Haoyu Liu, Paul Patras, and Douglas J Leith. "Android Mobile OS Snooping By Samsung, Xiaomi, Huawei and Realme Handsets". 2021.

[Lu+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2012.

[Ma+16] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. "LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps". In: *Proceedings of the International Conference on Software Engineering*. 2016.

[Mar+12] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. "Analysis of the Communication between Colluding Applications on Modern Smartphones". In: *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. 2012.

[MBN14] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. "Gyrophone: Recognizing Speech from Gyroscope Signals". In: *Proceedings of the USENIX Security Symposium*. 2014.

[Mic+15] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. "PowerSpy: Location Tracking Using Mobile Device Power Analysis". In: *Proceedings of the USENIX Security Symposium*. 2015.

[MN21] Mehran Mahmoudi and Sarah Nadi.
"The Android Update Problem: An Empirical Study". In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2021.

[MTN13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik.
"DynoDroid: An Input Generation System for Android Apps".
In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2013.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang.
"Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints". In: *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. 2010.

[Pan+18] Elleen Pan, Jingjing Ren, Martina Lindorfer, Christo Wilson, and David Choffnes. "Panoptispy: Characterizing Audio and Video Exfiltration from Android Applications".
In: *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)* (2018).

[Ped+19] Sai Teja Peddinti, Igor Bilogrevic, Nina Taft, Martin Pelikan, Úlfar Erlingsson, Pauline Anthonysamy, and Giles Hogben.
"Reducing Permission Requests in Mobile Apps".
In: *Proceedings of the Internet Measurement Conference (IMC)*. 2019.

[Pos+21] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. "Trust, But Verify: A Longitudinal Analysis of Android OEM Compliance and Customization".
In: *IEEE Symposium on Security and Privacy (SP)*. 2021.

[PWL20a] Ivan Pustogarov, Qian Wu, and David Lie.
"Ex-Vivo Dynamic Analysis Framework for Android Device Drivers".
In: *IEEE Symposium on Security and Privacy (SP)*. 2020.

[PWL20b] Ivan Pustogarov, Qian Wu, and David Lie.
"Ex-Vivo Dynamic Analysis Framework for Android Device Drivers". In: (2020).

[Qia+14] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan.
"On Tracking Information Flows through JNI In Android Applications". In: *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014.

[Qia+15] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu.
"Vulhunter: Toward Discovering Vulnerabilities in Android Applications".
In: *IEEE Micro* (2015).

[Raz+15] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson.
"Haystack: In Situ Mobile Traffic Analysis in User Space".
In: *arXiv preprint arXiv:1510.01419* (2015).

[Raz+18] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. "Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem". In: (2018).

[Rea+19] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman.
"50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions Systems". In: (2019).

[Ren+18] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez.
"Bug Fixes, Improvements,... and Privacy Leaks". In: (2018).

[Rey+18] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman.
""Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale".
In: *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)* (2018).

[Sad+18] Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek.
"A temporal permission analysis and enforcement framework for android".
In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 846–857.

[Sar+12] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy.
"Android Permissions: A Perspective Combining Risks and Benefits".
In: *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 2012.

[SBM15] Alireza Sadeghi, Hamid Bagheri, and Sam Malek.
"Analysis of android inter-app security vulnerabilities using covert".
In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 725–728.

[SC13] James Sellwood and Jason Crampton.
"Sleeping Android: The danger of Dormant Permissions". In: *Proceedings of the ACM workshop on Security and Privacy in Smartphones & Mobile Devices*. 2013.

[Spr+17] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. "Systematic Classification of Side-channel Attacks: A Case Study for Mobile Devices". In: *IEEE Communications Surveys & Tutorials* (2017).

[SQH17] Wei Song, Xiangxing Qian, and Jeff Huang.
"EHBDroid: Beyond GUI Testing for Android Applications". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017.

[SR14] Raimondas Sasnauskas and John Regehr.
"Intent Fuzzer: Crafting Intents of Death". In: *Proceedings of the Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. 2014.

[Sto18] Maddie Stone. "Unpacking the Packed Unpacker: Reversing an Android Anti-Analysis Native Library". In: (2018).

[Su+17] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su.
"Guided, Stochastic Model-Based GUI Testing of Android Apps".
In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017.

[SXA16] Laurent Simon, Wenduan Xu, and Ross Anderson.
"Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards".
In: *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)* (2016).

[Tam+15] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro.
"CopperDroid: Automatic Reconstruction of Android Malware Behaviors". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2015.

[Tia+18] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. "ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem". In: *Proceedings of the USENIX Security Symposium*. 2018.

[Tun+18] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl Gunter. "Resolving the Predicament of Android Custom Permissions". In: (2018).

[VCC11] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. "Curbing Android Permission Creep". In: *Proceedings of the International Conference on World Wide Web (WWW)*. 2011.

[VR+12] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Yan Grunenberger, Konstantina Papagiannaki, Hamed Haddadi, and Jon Crowcroft. "Breaking for Commercials: Characterizing Mobile Advertising". In: *Proceedings of the Internet Measurement Conference (IMC)*. 2012.

[VR+13] Narseo Vallina-Rodriguez, Jon Crowcroft, Alessandro Finamore, Yan Grunenberger, and Konstantina Papagiannaki. "When Assistance Becomes Dependence: Characterizing the Costs and Inefficiencies Of A-GPS". In: *Proceedings of the SIGMOBILE Mobile Computing and Communications Review* (2013).

[VR+14] Narseo Vallina-Rodriguez, Johanna Amann, Christian Kreibich, Nicholas Weaver, and Vern Paxson. "A Tangled Mass: The Android Root Certificate Stores". In: *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 2014.

[VRH98] Raja Vallee-Rai and Laurie J Hendren. "Jimple: Simplifying Java bytecode for Analyses and Transformations". In: (1998).

[Wan+18a] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. "Beyond Google

Play: A Large-Scale Comparative Study of Chinese Android App Markets".
In: *Proceedings of the Internet Measurement Conference (IMC)*. 2018.

[Wan+18b] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. "Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets". In: *Proceedings of the Internet Measurement Conference (IMC)*. 2018.

[Wei+14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2014.

[Wei+18] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. "JN-SAF: Precise and Efficient NDK/JNI-aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2018.

[Wij+15] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. "Android Permissions Remystified: A Field Study on Contextual Integrity". In: *Proceedings of the USENIX Security Symposium*. 2015.

[Wog+14] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr Olesen, and René Rydhof Hansen. "Formalisation and Analysis of Dalvik Bytecode". In: *Science of Computer Programming* (2014).

[WRO18] Fengguo Wei, Sankardas Roy, and Xinming Ou. "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps". In: *ACM Transactions on Privacy and Security (TOPS)* (2018).

[Wu+13] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. "The Impact of Vendor Customizations on Android Security". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2013.

[Wu+19] Daoyuan Wu, Debin Gao, Rocky K. C. Chang, En He, Eric K. T. Cheng, and Robert H. Deng. "Understanding Open Ports In Android Applications: Discovery, Diagnosis, And Security Assessment". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2019).

[Xue+18] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. "NDroid: Toward Tracking Information Flows across Multiple Android Contexts".
In: *IEEE Transactions on Information Forensics and Security* (2018).

[You+15] Wei You, Bin Liang, Jingzhe Li, Wenchang Shi, and Xiangyu Zhang. "Android Implicit Information Flow Demystified". In: *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 2015.

[YY12] Zhemin Yang and Min Yang. "Leakminer: Detect Information Leakage on Android with Static Taint Analysis".
In: *Third World Congress on Software Engineering*. 2012.

[ZG16] Yuri Zhauniarovich and Olga Gadyatskaya. "Small Changes, Big Changes: An Updated View On The Android Permission System". In: *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 2016.

[Zho+14] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. "The Peril Of Fragmentation: Security Hazards In Android Device Driver Customizations".
In: *IEEE Symposium on Security and Privacy (SP)*. 2014.

[ZSL14] Min Zheng, Mingshen Sun, and John CS Lui.
"DroidRay: a Security Evaluation System for Customized Android Firmwares".
In: *Proceedings of the ACM symposium on Information, computer and communications security (ASIA CCS)*. 2014.

[Adg] *AdGuard - Meizu Incompatibilities.*
https://github.com/AdguardTeam/AdguardForAndroid/issues/800.
Accessed on 29th of April, 2022.

[Ado] *Commit b858dfda50: Implement system data migration support.*
https://android.googlesource.com/platform/frameworks/base/+/b858dfda5012a1040927ed62c3bb856c3294d882.
Accessed on 29th of April, 2022.

[Adu] *Kryptowire - KRYPTOWIRE DISCOVERS MOBILE PHONE FIRMWARE THAT TRANSMITTED PERSONALLY IDENTIFIABLE INFORMATION (PII) WITHOUT USER CONSENT OR DISCLOSURE.*
http://www.kryptowire.com/adups_security_analysis.html.

Accessed on 29th of April, 2022.

[Aep] *Análisis del software preinstalado en dispositivos Android y riesgos para la privacidad de los usuarios.*
https://www.aepd.es/es/prensa-y-comunicacion/notas-de-prensa/analisis-del-software-preinstalado-en-dispositivos-android-y.
Accessed on 29th of April, 2022.

[Aet] *Aetherpal.*
https://aetherpal.com/.
Accessed on 29th of April, 2022.

[Amaa] *Amazon Appstore For Android.*
https://www.amazon.com/gp/feature.html?docId=1002999431.
Accessed on 29th of April, 2022.

[Amab] *Amazon Mechanical Turk.*
https://www.mturk.com/.
Accessed on 29th of April, 2022.

[Amac] *Amazon suspends sales of Blu phones for including preloaded spyware, again.*
https://www.theverge.com/2017/7/31/16072786/amazon-blu-suspended-android-spyware-user-data-theft.
Accessed on 29th of April, 2022.

[Amad] *Integrate Amazon Device Messaging (ADM).*
https://developer.amazon.com/docs/video-skills-fire-tv-apps/integrate-adm.html.

[Amae] *Integrate Your App with the Fire TV Launcher.*
https://developer.amazon.com/docs/catalog/integrate-with-launcher.html.

[Anda] *Androguard.*
https://github.com/androguard/androguard/.
Accessed on 29th of April, 2022.

[Andb] *Android — Certified.*
https://www.android.com/certified/.
Accessed on 29th of April, 2022.

[Andc] *Android 10 Release Notes | Android Open Source Project.*
https://source.android.com/setup/start/android-10-release#permissions.
Accessed on 29th of April, 2022.

[Andd] *Android 8 Release Notes | Android Open Source Project.*
https://developer.android.com/about/versions/oreo/android-8.0#perms.
Accessed on 29th of April, 2022.

[Ande] *Android Certified Partners — brands.*
https://www.android.com/certified/partners/.
Accessed on 29th of April, 2022.

[Andf] *Android Certified Partners — ODMs.*
https://www.android.com/certified/partners/#tab-panel-odms.
Accessed on 29th of April, 2022.

[Andg] *Android Compatibility Program Overview.*
https://source.android.com/compatibility/overview.
Accessed on 29th of April, 2022.

[Andh] *Android Comptability Document — Permissions.*
https://source.android.com/compatibility/android-cdd#9_1_permissions.
Accessed on 29th of April, 2022.

[Andi] *Android Developer Documentation.*
https://developer.android.com/.
Accessed on 29th of April, 2022.

[Andj] *Android Developers.*
https://developer.android.com/guide/topics/manifest/permission-element.html#plevel.
Accessed on 29th of April, 2022.

[Andk] *Android Developers | Announcing the Android 1.0 SDK, release 1.*
https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html.
Accessed on 29th of April, 2022.

[Andl] *Android founder: We aimed to make a camera OS.*

https://www.pcworld.com/article/451350/android-founder-we-aimed-to-make-a-camera-os.html.
Accessed on 29th of April, 2022.

[Andm] *Android NDK | Android Developers.*
https://developer.android.com/ndk/.
Accessed on 29th of April, 2022.

[Andn] *Android Observatory.*
https://androidobservatory.com/home.
Accessed on 29th of April, 2022.

[Ando] *Android Studio code annotations.*
https://android.googlesource.com/platform/tools/adt/idea/+/refs/heads/mirror-goog-studio-master-dev/android/annotations/android/.
Accessed on 29th of April, 2022.

[Andp] *Android Version Distribution statistics.*
https://www.xda-developers.com/android-version-distribution-statistics-android-studio/.
Accessed on 29th of April, 2022.

[Andq] *Androwarn–Yet another static code analyzer for malicious Android applications.*
https://github.com/maaaaz/androwarn.
Accessed on 29th of April, 2022.

[Andr] *AndroZoo.*
https://androzoo.uni.lu/.
Accessed on 29th of April, 2022.

[Ands] *Cars | Android Developers.*
https://developer.android.com/reference/android/car/Car.

[Andt] *Context | Android Developers.*
https://developer.android.com/reference/android/content/Context.

[Andu] *Design an Android Device.*
https://source.android.com/compatibility.
Accessed on 29th of April, 2022.

[Andv] *Fresher OS with Projects Treble and Mainline.*
https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html.

Accessed on 29th of April, 2022.

[Andw] *Google's Android OS: Past, Present, and Future.* https://www.phonearena.com/news/Googles-Android-OS-Past-Present-and-Future_id21273.
Accessed on 29th of April, 2022.

[Andx] *Here comes Treble: A modular base for Android.* https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.
Accessed on 29th of April, 2022.

[Andy] *Industry Leaders Announce Open Platform for Mobile Devices.* http://www.openhandsetalliance.com/press_110507.html.
Accessed on 29th of April, 2022.

[Andz] *Intents and Intent Filters - Android Developers.* https://developer.android.com/guide/components/intents-filters.
Accessed on 29th of April, 2022.

[Andaa] *One-time Permissions | Android Developers.* https://developer.android.com/training/permissions/requesting#one-time.
Accessed on 29th of April, 2022.

[Andab] *permission | Android Developers.* https://developer.android.com/guide/topics/manifest/permission-element#desc.

[Andac] *Permissions updates in Android 11 | Android Open Source Project.* https://developer.android.com/about/versions/11/privacy/permissions.
Accessed on 29th of April, 2022.

[Andad] *R.attr.* https://developer.android.com/reference/android/R.attr.html#protectionLevel.
Accessed on 29th of April, 2022.

[Andae] *R.attr | Android developers | knownCerts.* https://developer.android.com/reference/android/R.attr#knownCerts.

Accessed on 29th of April, 2022.

[Andaf] *There are over 3 billion active Android devices.*
https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021.
Accessed on 29th of April, 2022.

[Andag] *This is the droid you're looking for.*
https://android-developers.googleblog.com/2007/11/posted-by-jason-chen-android-advocate.html.
Accessed on 29th of April, 2022.

[Andah] *Tristate Location Permissions | Android Open Source Project.*
https://source.android.com/devices/tech/config/tristate-perms.
Accessed on 29th of April, 2022.

[Andai] *UI/Application Exerciser Monkey.*
https://developer.android.com/studio/test/other-testing-tools/monkey.
Accessed on 29th of April, 2022.

[Andaj] *VPN Service.*
https://developer.android.com/reference/android/net/VpnService.
Accessed on 29th of April, 2022.

[Andak] *What is "com,facebook,app manager" and why is it trying to download Instagram, Facebook, and Messenger.*
https://forums.androidcentral.com/android-apps/547447-what-com-facebook-app-manager-why-trying-download-instagram-facebook-messenge.html.
Accessed on 29th of April, 2022.

[Ant] *Ant+.*
https://www.thisisant.com.
Accessed on 29th of April, 2022.

[Aosa] *Android Developers | PackageManager.*
https://developer.android.com/reference/android/content/pm/PackageManager.
Accessed on 29th of April, 2022.

[Aosb] *Android Open Source Project.*

https://source.android.com/.

Accessed on 29th of April, 2022.

[Aosc] *AndroidManfiest.xml.*

https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/AndroidManifest.xml.

Accessed on 29th of April, 2022.

[Aosd] *AndroidManifest.xml | Android Open Source Project.*

https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/AndroidManifest.xml.

Accessed on 29th of April, 2022.

[Apea] *APEX File Format.*

https://source.android.com/devices/tech/ota/apex.

Accessed on 29th of April, 2022.

[Apeb] *APEX Manager.*

https://source.android.com/devices/tech/ota/apex#apex_manager.

Accessed on 29th of April, 2022.

[Apka] *APK Mirror App Store.*

https://www.apkmirror.com/.

Accessed on 29th of April, 2022.

[Apkb] *APK Monk App Store.*

https://www.apkmonk.com/.

Accessed on 29th of April, 2022.

[Apkc] *Apktool–A tool for reverse engineering Android apk files.*

https://ibotpeaches.github.io/Apktool/.

Accessed on 29th of April, 2022.

[Appa] *Application Manifest Overview.*

https://developer.android.com/guide/topics/manifest/manifest-intro.html.

Accessed on 29th of April, 2022.

[Appb] *Appsee — Features.*

https://www.appsee.com/features.

Accessed on 29th of April, 2022.

[Appc] *Appsee Mobile App Analytics.*
https://www.appsee.com/.
Accessed on 29th of April, 2022.

[Appd] *Commit 33f5ddd: Add permissions associated with app ops.*
https://android.googlesource.com/platform/frameworks/base/+/33f5ddd1bea21296938f2cba196f95d223aa247c.
Accessed on 29th of April, 2022.

[Appe] *Commit a06de0f: New "app ops" service.*
https://android.googlesource.com/platform/frameworks/base/+/a06de0f29b58df9246779cc4bfd8f06f7205ddb6.
Accessed on 29th of April, 2022.

[Appf] *Market share development per Android phone manufacturer.*
https://www.appbrain.com/stats/top-manufacturers.

[Appg] *Monetize, advertise and analyze Android apps.*
www.appbrain.com/.
Accessed on 29th of April, 2022.

[Asua] *Asurion.*
https://www.asurion.com/.
Accessed on 29th of April, 2022.

[Asub] *Hackers Hijacked ASUS Software Updates to Install Backdoors on Thousands of Computers.*
https://motherboard.vice.com/en_us/article/pan9wn/hackers-hijacked-asus-software-updates-to-install-backdoors-on-thousands-of-computers.
Accessed on 29th of April, 2022.

[Baia] *Baidu Android Appstore.*
https://shouji.baidu.com/.
Accessed on 29th of April, 2022.

[Baib] *Baidu App Store.*
https://shouji.baidu.com/.
Accessed on 29th of April, 2022.

[Baic] *Baidu Geocoding API.*
http://api.map.baidu.com/lbsapi/geocoding-api.htm.

Accessed on 29th of April, 2022.

[Baid] *Baidu SDK*.
https://developer.baidu.com/.
Accessed on 29th of April, 2022.

[Bgc] *Compatibility Test Suite | Android Developers*.
https://source.android.com/compatibility/cts.
Accessed on 29th of April, 2022.

[Bgv] *Vendor Test Suite (VTS) and Infrastructure | Android Developers*.
https://source.android.com/compatibility/vts.
Accessed on 29th of April, 2022.

[Bil] *Android Developers - AIDL to Google Play Billing Library migration guide*.
https://developer.android.com/google/play/billing/migrate.
Accessed on 29th of April, 2022.

[Ccp] *California Consumer Privacy Act*.
https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.
Accessed on 29th of April, 2022.

[Cer] *Android Certified Partners*.
https://www.android.com/certified/partners/.
Accessed on 29th of April, 2022.

[Coma] *Commit 0dec6c042cf: Create recents protectionLevel*.
https://android.googlesource.com/platform/frameworks/base/+/0dec6c042cf3752cd853eab4a0909c7afc6c23f5.
Accessed on 29th of April, 2022.

[Comb] *Commit 15707b3f4df: Define protection level for document manager*.
https://android.googlesource.com/platform/frameworks/base/+/15707b3f4df8f44881643adfc369b3cd50bc5598.
Accessed on 29th of April, 2022.

[Comc] *Commit 1fa23ed08ac: [CDM] Bypass location setting when scanning for devices*.
https://android.googlesource.com/platform/frameworks/base/+/1fa23ed08ac4c2113319097d03e30b558dd37698.
Accessed on 29th of April, 2022.

[Comd] *Commit 27eef5cfcdd: Add support for knownSigner permission protection flag.* https://android.googlesource.com/platform/frameworks/base/+/27eef5cfcdd21bdf549f59fc7a150a9811e1835a.
Accessed on 29th of April, 2022.

[Come] *Commit 2a1376d9dfb: Expose removed permissions flag as system API.* https://android.googlesource.com/platform/frameworks/base/+/2a1376d9dfb362a18ba110d8e172f96021f1d879.
Accessed on 29th of April, 2022.

[Comf] *Commit 2ca2c878713: More adjustments to permissions.* https://android.googlesource.com/platform/frameworks/base/+/2ca2c8787130506d350d997c18bbc274faf88e37.
Accessed on 29th of April, 2022.

[Comg] *Commit 596437fd4e0: Added a new set of permissions for DeviceConfig API.* https://android.googlesource.com/platform/frameworks/base/+/596437fd4e0941df378558a374c172148bb37b7c.
Accessed on 29th of April, 2022.

[Comh] *Commit 5a15b551076: Added a new "incidentReportApprover" permission protection flag.* https://android.googlesource.com/platform/frameworks/base/+/5a15b55107651968312f39a830ddb26909b9d362.
Accessed on 29th of April, 2022.

[Comi] *Commit 5f303659c23: Added a new "wellbeing" protection flag.* https://android.googlesource.com/platform/frameworks/base/+/5f303659c23d7d0a944aa51edb9b3353da1d497d.
Accessed on 29th of April, 2022.

[Comj] *Commit 8126b1fe0f0: Added a new "retailDemo" protection level.* https://android.googlesource.com/platform/frameworks/base/+/8126b1fe0f09f359f3ec3b80bc5717e101e295b8.
Accessed on 29th of April, 2022.

[Comk] *Commit a3968751099: Add role as a new permission protection flag.* https://android.googlesource.com/platform/frameworks/base/+/a3968751099bd85f6a20673a8556b033f82357a3.
Accessed on 29th of April, 2022.

[Coml] *Commit b9893a600ea: Add internal as a new permission protection level.*
https://android.googlesource.com/platform/frameworks/base/+/b9893a600ea8c047cebb6a4a352322916ba8eaca.
Accessed on 29th of April, 2022.

[Comm] *Commit cd7695dda05: Add a new "appPredictor" protection flag.*
https://android.googlesource.com/platform/frameworks/base/+/cd7695dda0576a954745a59d3feb579bcb644795.
Accessed on 29th of April, 2022.

[Comn] *Commit cfbfafe1b9c: Additional permissions aren't properly disabled after toggling them off.*
https://android.googlesource.com/platform/frameworks/base/+/cfbfafe1b9ca2fd135a4fb6b528b3829830803bf.
Accessed on 29th of April, 2022.

[Como] *Commit d563e937f2d: Make storage a restricted permission - framework.*
https://android.googlesource.com/platform/frameworks/base/+/d563e937f2d2a6d256b1284c3119c8787faf156d.
Accessed on 29th of April, 2022.

[Comp] *Commit d7087b25ce3: Introduce new permissionFlag.*
https://android.googlesource.com/platform/frameworks/base/+/d7087b25ce394ec54cc6ec8e2852aee0a12c0e8a.
Accessed on 29th of April, 2022.

[Comq] *Commit d8eb8b2690d: Restricted permission mechanism - framework.*
https://android.googlesource.com/platform/frameworks/base/+/d8eb8b2690dd27d5ffe6262dd8ce8594ec8028a6.
Accessed on 29th of April, 2022.

[Cop] *COPPA - Children's Online Privacy Protection Act.*
http://coppa.org/.
Accessed on 29th of April, 2022.

[Cusa] *Android Developers - Define a Custom App Permission.*
https://developer.android.com/guide/topics/permissions/defining.
Accessed on 29th of April, 2022.

[Cusb] *Define a Custom Permission.*
https://developer.android.com/guide/topics/permissions/defining.

Accessed on 29th of April, 2022.

[Data] *Dataset of defined custom permissions.* https://androidobservatory.com/files/defined_perms_all_release.json.xz.

[Datb] *Dataset of requested custom permissions.* https://androidobservatory.com/files/requested_perms_all_release.json.xz.

[Dev] *Commit e639da7: New development permissions.* https://android.googlesource.com/platform/frameworks/base/+/e639da7baa23121e35aa06d6e182558e0e755696.
Accessed on 29th of April, 2022.

[Dex] *Dextripador | A tool to extract the DEX file from ODEX compiled ahead of time version.* https://github.com/Android-Observatory/DEXtripador.
Accessed on 29th of April, 2022.

[Dig] *Digital Turbine - Privacy Policy.* https://www.digitalturbine.com/privacy-policy/.
Accessed on 29th of April, 2022.

[Dr] Dr Web. *Trojan preinstalled on Android devices infects applications' processes and downloads malicious modules.* http://news.drweb.com/news/?i=11390&lng=en.
Accessed on 29th of April, 2022.

[Dtb] *Mobile Posse acquired by Digital Turbine.* https://www.crunchbase.com/acquisition/mandalay-digital-group-acquires-mobile-posse--1e380e32.
Accessed on 29th of April, 2022.

[Dum] *Manifest.permission — Android Developpers.* https://developer.android.com/reference/android/Manifest.permission.html#DUMP.
Accessed on 29th of April, 2022.

[Eph] *Commit c19706a: Add ephemeral protection level.* https://android.googlesource.com/platform/frameworks/base/+/c19706a937abc5d025a59b354b3a0d89e7d62805.

Accessed on 29th of April, 2022.

[Est] *Estimote — indoor location with bluetooth beacons and mesh.*
https://estimote.com/.
Accessed on 29th of April, 2022.

[Exu] *EXUS.*
https://www.exus.co.uk.
Accessed on 29th of April, 2022.

[Fac] *Facebook Gave Device Makers Deep Access to Data on Users and Friends.*
https://www.nytimes.com/interactive/2018/06/03/technology/facebook-device-partners-users-friends-data.html.
Accessed on 29th of April, 2022.

[Fin] *Google Issue Tracker - Why Google play services dependency automatically added com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE permission.*
https://issuetracker.google.com/issues/78380811#comment22.
Accessed on 29th of April, 2022.

[Ftc] *Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission.*
https://www.ftc.gov/news-events/news/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked-hundreds-millions-consumers.
Accessed on 29th of April, 2022.

[Gcma] *Google Cloud Messaging.*
https://developers.google.com/cloud-messaging/android/android-migrate-fcm.
Accessed on 29th of April, 2022.

[Gcmb] *Migrate a GCM Client App for Android to Firebase Cloud Messaging.*
https://developers.google.com/cloud-messaging/android/android-migrate-fcm.

[Gdp] *EU General Data Protection Regulation (GDPR).*
https://eugdpr.org/.
Accessed on 29th of April, 2022.

[Gmoa] *Android.Gmobi.1.*
https://vms.drweb.com/virus/?_is=1&i=7999623&lng=en.
Accessed on 29th of April, 2022.

[Gmob] *GMobi — General Mobile Corporation.*
http://www.generalmobi.com/en/.
Accessed on 29th of April, 2022.

[Gooa] *Best practices for unique identifiers.*
https://developer.android.com/training/articles/user-data-ids.
Accessed on 29th of April, 2022.

[Goob] *Google Buys Android for Its Mobile Arsenal.*
https://web.archive.org/web/20110205190729/http://www.businessweek.com/technology/content/aug2005/tc20050817_0949_tc024.htm.
Accessed on 29th of April, 2022.

[Gooc] *Privacy Security Best Practices | Android Open Source Project.*
https://source.android.com/security/best-practices/privacy#logging-data.
Accessed on 29th of April, 2022.

[Gpla] *Android Developers - Define a Custom App Permission.*
https://developer.android.com/guide/topics/permissions/defining.
Accessed on 29th of April, 2022.

[Gplb] *Android Developers - Permissions Overview.*
https://developer.android.com/guide/topics/permissions/overview.
Accessed on 29th of April, 2022.

[Gplc] *Google Play App Store.*
https://play.google.com/store/apps/.
Accessed on 29th of April, 2022.

[Gra] *PermissionManagerService.java.*
https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/services/core/java/com/android/server/pm/permission/PermissionManagerService.java#1037.
Accessed on 29th of April, 2022.

[Hiya] *Hiya.*

https://hiya.com/.

Accessed on 29th of April, 2022.

[Hiyb] *Hiya Partners.*

https://hiya.com/hiya-data-policy.

Accessed on 29th of April, 2022.

[Htc] *T-Mobile G1: Full Details of the HTC Dream Android Phone.*

https://gizmodo.com/t-mobile-g1-full-details-of-the-htc-dream-android-phon-5053264.

Accessed on 29th of April, 2022.

[Huaa] *CVE-2017-2709.*

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2709.

Accessed on 29th of April, 2022.

[Huab] *Europe should be wary of Huawei, EU tech official says.*

https://www.reuters.com/article/us-eu-china-huawei-idUSKBN1O611X.

Accessed on 29th of April, 2022.

[Huac] *Huawei App Store.*

https://appgallery1.huawei.com/#/Featured.

Accessed on 29th of April, 2022.

[Huad] *Huawei's Android App Store Launches Internationally.*

https://www.androidheadlines.com/2018/04/huaweis-android-app-store-launches-internationally.html.

Accessed on 29th of April, 2022.

[Inf] *Infinum Inc.*

https://infinum.co.

Accessed on 29th of April, 2022.

[Insa] *Commit c247fa1: Change protection level from ephemeral to instant.*

https://android.googlesource.com/platform/frameworks/base/+/c247fa136639dd07278b1954e5fba78ade60614c.

Accessed on 29th of April, 2022.

[Insb] *Google Play Instant.*

https://developer.android.com/topic/google-play-instant/.

Accessed on 29th of April, 2022.

[Int] *Intent - Android Developers.*
https://developer.android.com/reference/android/content/Intent.

[Iroa] *IronSource — App monetization done right.*
https://www.ironsrc.com/.
Accessed on 29th of April, 2022.

[Irob] *IronSource - AURA.*
https://company.ironsrc.com/enterprise-solutions/.
Accessed on 29th of April, 2022.

[Iroc] *IronSource - Aura for Advertisers.*
https://www.slideshare.net/ironSource/aura-for-advertisers.
Accessed on 29th of April, 2022.

[Jav] *Trail: The Reflection API.*
https://docs.oracle.com/javase/tutorial/reflect/index.html.
Accessed on 29th of April, 2022.

[Jaya] Jay Freeman (saurik). *Android bug superior to master key.*
http://www.saurik.com/id/18.
Accessed on 29th of April, 2022.

[Jayb] Jay Freeman (saurik). *Exploit and fix android master key.*
http://www.saurik.com/id/17.
Accessed on 29th of April, 2022.

[Jayc] Jay Freeman (saurik). *Yet another android master key bug.*
http://www.saurik.com/id/19.
Accessed on 29th of April, 2022.

[Joh] Johnson, Ryan and Stavrou, Angelos and Benameur, Azzedine.
All Your SMS & Contacts Belong to ADUPS & Others.
https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf.
Accessed on 29th of April, 2022.

[Knoa] *New permissions names.*
https://docs.samsungknox.com/dev/knox-sdk/new-permission-names.htm.

[Knob] *Permissions.*
https://docs.samsungknox.com/dev/common/license-permissions.htm.

[Krya] *Kryptowire Discovers Mobile Phone Firmware that Transmitted Personally Identifiable Information (PII) without User Consent or Disclosure.* https://www.kryptowire.com/adups_security_analysis.html. Accessed on 29th of April, 2022.

[Kryb] *Kryptowire Provides Technical Details on Black Hat 2017 Presentation: Observed ADUPS Data Collection & Data Transmission.* https://www.kryptowire.com/observed_adups_data_collection_behavior.html. Accessed on 29th of April, 2022.

[Len] *Lenovo Mobility.* https://solutions.lenovo.com/pc-solutions/mobility/. Accessed on 29th of April, 2022.

[Lgb] *LG Enterprise Mobile Solutions.* https://www.lg.com/us/business/enterprise-mobility/business-resources/solution/mobile-device-management. Accessed on 29th of April, 2022.

[Loc] *locationlabs by Avast.* https://www.locationlabs.com/. Accessed on 29th of April, 2022.

[Lok] *Android Adware and Ransomware Found Preinstalled on High-End Smartphones.* https://www.bleepingcomputer.com/news/security/android-adware-and-ransomware-found-preinstalled-on-high-end-smartphones/. Accessed on 29th of April, 2022.

[Lum] *Lumen Privacy Monitor.* https://play.google.com/store/apps/details?id=edu.berkeley.icsi.haystack. Accessed on 29th of April, 2022.

[Mer] *Merge multiple manifest files.* https://developer.android.com/studio/build/manifest-merge. Accessed on 29th of April, 2022.

[Mir] *Mirrorlink - Connected Car Consortium.* https://mirrorlink.com. Accessed on 29th of April, 2022.

[Mis] *Xiaomi Mi App Store.*
http://app.mi.com/.
Accessed on 29th of April, 2022.

[Mob] *Digital Turbine – Mobile Posse.*
https://mobileposse.com/.
Accessed on 29th of April, 2022.

[Mst] *Securing the System: A Deep Dive into Reversing Android Pre-Installed Apps | Black Hat 2019.*
https://www.blackhat.com/us-19/briefings/schedule/index.html#securing-the-system-a-deep-dive-into-reversing-android-pre-installed-apps-16040.
Accessed on 29th of April, 2022.

[New] *App Traps: How Cheap Smartphones Siphon User Data in DevelopingmCountries.*
https://www.wsj.com/articles/app-traps-how-cheap-smartphones-help-themselves-to-user-data-1530788404.
Accessed on 29th of April, 2022.

[New] New York Times.
Facebook Gave Device Makers Deep Access to Data on Users and Friends.
https://www.nytimes.com/interactive/2018/06/03/technology/facebook-device-partners-users-friends-data.html.
Accessed on 29th of April, 2022.

[Nyt] *Facebook's Data Deals Are Under Criminal Investigation.*
https://www.nytimes.com/2019/03/13/technology/facebook-data-deals-investigation.html.
Accessed on 29th of April, 2022.

[Oem] *Commit 087dce2: Add new OEM permission flavor.*
https://android.googlesource.com/platform/frameworks/base/+/087dce20e3a7137e94607c060fd825d1f8952572.
Accessed on 29th of April, 2022.

[Off] *Android Developers | Standard partitions.*
https://source.android.com/devices/bootloader/partitions.
Accessed on 29th of April, 2022.

[Oma] *Open Mobile API Specification v3.3.*
https://globalplatform.org/specs-library/open-mobile-api-specification-v3-3/.
Accessed on 29th of April, 2022.

[Onea] *OnePlus Device Root Exploit: Backdoor in EngineerMode App for Diagnostics Mode.*
https://www.nowsecure.com/blog/2017/11/14/oneplus-device-root-exploit-backdoor-engineermode-app-diagnostics-mode/.
Accessed on 29th of April, 2022.

[Oneb] *OnePlus left a backdoor in its devices capable of root access.*
http://www.androidpolice.com/2017/11/15/oneplus-left-backdoor-devices-capable-root-access/.
Accessed on 29th of April, 2022.

[Onec] *OnePlus OxygenOS built-in analytics.*
https://www.chrisdcmoore.co.uk/post/oneplus-analytics/.
Accessed on 29th of April, 2022.

[Oned] *OnePlus Secret Backdoor.*
https://www.theregister.co.uk/2017/11/14/oneplus_backdoor/.
Accessed on 29th of April, 2022.

[Pera] *Manifest.permission_group.*
https://developer.android.com/reference/android/Manifest.permission_group.html.
Accessed on 29th of April, 2022.

[Perb] *Permission groups.*
https://developer.android.com/guide/topics/manifest/permission-group-element.
Accessed on 29th of April, 2022.

[Perc] *Permission trees.*
https://developer.android.com/guide/topics/manifest/permission-tree-element.
Accessed on 29th of April, 2022.

[Perd] *PermissionTainter.*
https://github.com/Android-Observatory/PermissionTainter.

[Pere] *PermissionTracer.*
https://github.com/Android-Observatory/PermissionTracer.

[Perf] *Privileged Permission Allowlisting.*
https://source.android.com/devices/tech/config/perms-allowlist.
Accessed on 29th of April, 2022.

[Plaa] *Android Developers - GoogleSignInApi.*
https://developers.google.com/android/reference/com/google/android/gms/auth/api/signin/GoogleSignInApi.
Accessed on 29th of April, 2022.

[Plab] *Android Developers - Play Install Referrer Library.*
https://developer.android.com/google/play/installreferrer/library.
Accessed on 29th of April, 2022.

[Plac] *https://developers.google.com/android/play-protect.*
https://developers.google.com/android/play-protect.
Accessed on 29th of April, 2022.

[Prea] *Commit a90c8de: Add new "preinstalled" permission flag.*
https://android.googlesource.com/platform/frameworks/base/+/a90c8def2c6762bc6e5396b78c43e65e4b05079d.
Accessed on 29th of April, 2022.

[Preb] *Commit de15eda: Add new permissions flag, saying the permission can be automatically granted to pre-api-23 apps.*
https://android.googlesource.com/platform/frameworks/base/+/de15edaa9bf486a4050bb067317d313fd807bb10.
Accessed on 29th of April, 2022.

[Pria] *Privacy Grade.*
http://privacygrade.org/.
Accessed on 29th of April, 2022.

[Prib] *PrivacyStar.*
https://privacystar.com.
Accessed on 29th of April, 2022.

[Pric] *PrivacyStar Privacy Policy.*
https://privacystar.com/privacy-policy/.

Accessed on 29th of April, 2022.

[Pro] *Prolific - Quickly find research participants you can trust.*
https://www.prolific.co/.
Accessed on 29th of April, 2022.

[Pusa] *Amazon Push Notification Service.*
https://developer.amazon.com/docs/adm/integrate-your-app.html.
Accessed on 29th of April, 2022.

[Pusb] *Baidu Push Notification Service.*
http://push.baidu.com/doc/android/api.
Accessed on 29th of April, 2022.

[Pusc] *Google Maps Receive Permission.*
https://stackoverflow.com/questions/14832911/android-map-v2-why-maps-receive-permission.
Accessed on 29th of April, 2022.

[Pusd] *Google Push Notification Service.*
https://web.archive.org/web/20121004073640/https://developers.google.com/android/c2dm/.
Accessed on 29th of April, 2022.

[Puse] *Huawei Push Notification Service.*
https://stackoverflow.com/questions/57860791/how-to-access-payload-of-hms-push-notifications.
Accessed on 29th of April, 2022.

[Pusf] *Jiguang Push Notification Service.*
https://docs.jiguang.cn/en/jpush/client/Android/android_guide/#configuration-and-code-instructions.
Accessed on 29th of April, 2022.

[Pusg] *Xiaomi Push Notification Service.*
https://docs.moengage.com/docs/android-xiaomi-push.
Accessed on 29th of April, 2022.

[Qih] *Qihoo 360 App Store.*
http://zhushou.360.cn/.
Accessed on 29th of April, 2022.

[Rea] Joel Reardon. *Why Google Should Stop Logging Contact-Tracing Data.* https://blog.appcensus.io/2021/04/27/why-google-should-stop-logging-contact-tracing-data/.
Accessed on 29th of April, 2022.

[Rec] *Recents screen | Android developers.* https://developer.android.com/guide/components/activities/recents.
Accessed on 29th of April, 2022.

[Reda] *China Mobile Network Partner Redstone Moves into Robotics.* https://www.prweb.com/releases/2017/04/prweb14212503.htm.
Accessed on 29th of April, 2022.

[Redb] *Redstone.* http://www.redstone.net.cn/.
Accessed on 29th of April, 2022.

[Ref] *java.lang.reflect - Android Documentation.* https://developer.android.com/reference/java/lang/reflect/package-summary.
Accessed on 29th of April, 2022.

[Req] *RequiresPermission — AndroidX.* https://developer.android.com/reference/androidx/annotation/RequiresPermission.
Accessed on 29th of April, 2022.

[Rooa] *Rootnik Android Trojan Abuses Commercial Rooting Tool and Steals Private Information.* https://unit42.paloaltonetworks.com/rootnik-android-trojan-abuses-commercial-rooting-tool-and-steals-private-information/.
Accessed on 29th of April, 2022.

[Roob] *Simple to use root checking Android library.* https://github.com/scottyab/rootbeer.
Accessed on 29th of April, 2022.

[Rsa] *Challenges in Android Supply Chain Analysis.*

https://published-prd.lanyonevents.com/published/rsaus20/sessionsFiles/17497/2020_USA20_MBS-R09_01_Challenges%20in%20Android%20Supply%20Chain%20Analysis.pdf.
Accessed on 29th of April, 2022.

[Runa] *Commit a5d70a: Allow permissions to be runtime-only.*
https://android.googlesource.com/platform/frameworks/base/+/a5d70a17ebd1b3ffe026879c5d9d96f04d10d4f2.
Accessed on 29th of April, 2022.

[Runb] *Runtime Permissions.*
https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-runtime-permissions.
Accessed on 29th of April, 2022.

[Sama] *CVE-2017-2709.*
https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0864.
Accessed on 29th of April, 2022.

[Samb] *Samsung Knox : secure mobile platform and solution.*
https://www.samsungknox.com/en.
Accessed on 29th of April, 2022.

[Samc] *Samsung Mobile Device Management Solutions.*
https://developer.samsung.com/tech-insights/knox/mobile-device-management.
Accessed on 29th of April, 2022.

[Sca] *Firmware Scanner.*
https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader.
Accessed on 29th of April, 2022.

[Set] *Commit b233466: Add new protection level for setup wizard.*
https://android.googlesource.com/platform/frameworks/base/+/b23346639b66783c1662fd8ffa5345ef5cef336c.
Accessed on 29th of April, 2022.

[Sha] *android:sharedUserId.*

https://developer.android.com/guide/topics/manifest/manifest-element#uid.
Accessed on 29th of April, 2022.

[Sig] *Application signing.*
https://developer.android.com/studio/publish/app-signing.
Accessed on 29th of April, 2022.

[Sma] *Smaato Blog.*
https://blog.smaato.com/everything-you-need-to-know-about-location-based-mobile-advertising.
Accessed on 29th of April, 2022.

[Smi] *Smith Micro Software.*
https://www.smithmicro.com.
Accessed on 29th of April, 2022.

[Sof] *SOFTBANK announces the completion of Vodafone K.K.'s acquisition.*
https://group.softbank/en/news/press/20060427.
Accessed on 29th of April, 2022.

[Sto] *Securing the System — A Deep Dive into Reversing Android Pre-Installed Apps.*
https://github.com/maddiestone/ConPresentations/blob/master/Blackhat2019.SecuringTheSystem.pdf.
Accessed on 29th of April, 2022.

[Syn] *Synchronoss Technologies - Privacy Policy.*
https://synchronoss.com/privacy-policy/#datacollected.
Accessed on 29th of April, 2022.

[Sys] *Manifest permissions.*
https://developer.android.com/reference/android/Manifest.permission.
Accessed on 29th of April, 2022.

[Tena] *Tencent Android Appstore.*
https://android.app.qq.com.
Accessed on 29th of April, 2022.

[Tenb] *Tencent App Store.*
https://android.myapp.com/.
Accessed on 29th of April, 2022.

[Texa] *Commit 700feef: Shortcut permissions for default text classifier.*
https://android.googlesource.com/platform/frameworks/base/+/700feef8a60e06784d28d1db9502e650df854cad.
Accessed on 29th of April, 2022.

[Texb] *Implementing Text Classification.*
https://source.android.com/devices/tech/display/textclassifier.
Accessed on 29th of April, 2022.

[Tri] *Triada Trojan Found in Firmware of Low-Cost Android Smartphones.*
https://www.bleepingcomputer.com/news/security/android-adware-and-ransomware-found-preinstalled-on-high-end-smartphones/.
Accessed on 29th of April, 2022.

[Trua] *"Betrayed by an app she had never heard of" - How TrueCaller is endangering journalists.*
https://privacyinternational.org/node/2997.
Accessed on 29th of April, 2022.

[Trub] *How does Truecaller get its data?*
https://support.truecaller.com/hc/en-us/articles/212638485-How-does-Truecaller-get-its-data.
Accessed on 29th of April, 2022.

[Truc] *Phone Number Search — TrueCaller.*
https://www.truecaller.com/.
Accessed on 29th of April, 2022.

[Trud] *Your Data Is Our Data: A Truecaller Breakdown.*
https://techcabal.com/2018/05/02/your-data-is-our-data-a-truecaller-breakdown/.
Accessed on 29th of April, 2022.

[Ups] *Upstream - Low-end Android smartphones sold with pre-installed malicious software in emerging markets.*
https://www.upstreamsystems.com/pre-installed-malware-android-smartphones/.
Accessed on 29th of April, 2022.

[Ven] *Commit 002fdbd: Support privileged vendor apps.*

https://android.googlesource.com/platform/frameworks/base/+/002fdbdb950ebbf40331a27de33b80db33e40d30.
Accessed on 29th of April, 2022.

[Ver] *Commit 3e7d977: Grant installer and verifier install permissions robustly.*
https://android.googlesource.com/platform/frameworks/base/+/3e7d977ff7c743713f0ad6336a039d7760ba47d1.
Accessed on 29th of April, 2022.

[Vir] *VirusTotal.*
https://www.virustotal.com/.
Accessed on 29th of April, 2022.

[Vpn] *VpnService — Android Developpers.*
https://developer.android.com/reference/android/net/VpnService.html.
Accessed on 29th of April, 2022.

[Weba] *WebUSB API.*
https://wicg.github.io/webusb/.
Accessed on 29th of April, 2022.

[Webb] *@yume-chan/adb - npm.*
https://www.npmjs.com/package/@yume-chan/adb.
Accessed on 29th of April, 2022.

[Webc] *@yume-chan/adb-backend-webusb - npm.*
https://www.npmjs.com/package/@yume-chan/adb-backend-webusb.
Accessed on 29th of April, 2022.

[Webd] *@yume-chan/adb-credential-web - npm.*
https://www.npmjs.com/package/@yume-chan/adb-credential-web.
Accessed on 29th of April, 2022.

[Xda] *XDA-Developers Forum (Galaxy Note 4). com.facebook.appmanager.*
https://forum.xda-developers.com/note-4/themes-apps/com-facebook-appmanager-t2919151.
Accessed on 29th of April, 2022.

Acronyms

ADB Android Debug Bridge.

AOSP Android Open Source Project.

APEX Android Pony EXpress.

API application programming interface.

APK Android Package.

ART Android RunTime.

ATS Advertising and Tracking Service.

AV Anti-Virus.

CDD Android Compatibility Definition Document.

CFG Control Flow Graph.

CTS Compatibility Test Suite.

DEX Dalvik Executable File.

DPA Data Protection Agency.

DPO Data Protection Officer.

DVM Dalvik Virtual Machine.

FOTA Fimware Over The Air.

FQDN fully qualified domain name.

GAEN Google-Apple Exposure Notifications.

GCM Google Cloud Messaging.

GSMA GSM Association.

HAL Hardware Abstraction Layer.

IAC Inter-App Communication.

ICC Inter-Component Communication.

IMEI International Mobile Equipment Identity.

IPC Inter-Process Communication.

IRB Intitutional Review Board.

JNI Java Native Interface.

MDM Mobile Device Management.

MNO Mobile Network Operator.

NDK Native Development Kit.

ODEX Optimized Dalvik Executable File.

ODM Original Device Manufacturer.

OEM Original Equipment Manufacturer.

OHA Open Handset Alliance.

OOBE Out-of-the-box Experience.

OS operating system.

PII Personally Identifiable Information.

PPI Pay-per-Install.

PUP Potentially Unwanted Programs.

RPI rolling proximity identifiers.

SDK Software Development Kit.

SLD second-level domain.

TPL Third-Party Library.

UID unique identifier.

VTS Vendor Test Suite.