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[†]
[*]IMDEA Networks Institute, [†]Universidad Carlos III de Madrid, [‡]NortonLifelock Research Group, [§]ICSI

***Abstract*—Android firmware updates are typically managed by the so-called FOTA (Firmware Over-the-Air) apps. 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 Original Equipment Manufacturers (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. This paper performs the first large-scale and systematic analysis of the FOTA ecosystem through a dataset of 2,013 FOTA apps detected with a tool designed for this purpose over 422,121 pre-installed apps. We classify the different stakeholders developing and deploying FOTA apps on the Android update ecosystem, showing that 43% of FOTA apps are developed by third parties. We report that some devices can have as many as 5 apps implementing FOTA capabilities. By means of static analysis of the code of FOTA apps, we show that some 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 discover 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 study telemetry data collected from real devices by a commercial security tool. We demonstrate that FOTA apps are responsible for the installation of non-system apps (*e.g.*, entertainment apps and games), including malware and Potentially Unwanted Programs (PUP). Our findings suggest that FOTA development practices are misaligned with Google's recommendations.**

I. Introduction

Android is now the most used operating system ever, with over 2.5 billion active Android devices [25] and a global market share of over 40% [21]. Part of Android's success is due to the openness of the platform, which allows any device manufacturer to customize and deploy their own Android version. Paradoxically, this open model has resulted in a poorly understood ecosystem of actors that play key roles at different stages of the Android supply chain [43]. This model has also accentuated platform fragmentation problems despite Google's efforts for harmonization [4]: millions of devices got stuck on outdated and no-longer supported Android versions [53], [58].

Platform updates are a particularly critical and highly privileged element of the Android ecosystem. Once a patch or a new Android version is released, each vendor needs to ship it over-the-air to their userbase. A FOTA (Firmware-Over-The-Air) app is the software responsible for downloading and applying these updates on the device, receiving this name from the way in which firmware updates are shipped. Traditionally, device vendors have been named responsible for applying system updates (including security patches) because they are technically the ones that build the operating system after customizing the Android Open Source Project (AOSP) code maintained by Google. In truth however, updates are often delegated to third- and even fourth-party services in the Android ecosystem. This state-of-affairs not only accentuates platform fragmentation, but also opens the ground for potential abuse if appropriate supervisory mechanisms are not in place. As a result of their privileged position, an attacker (or a deceptive FOTA provider or partner) could install potentially harmful system-level components over a large userbase, as it has already occurred in desktop platforms [16].

In Android, there is anecdotal evidence of at least one so-called FOTA app being used to distribute potentially harmful apps. This was the case of Adups [19], [47], a Chinese wireless update service provider offering support to millions of low-cost Android vendors, which was reported as spyware. However, the system update components in Android devices have been largely overlooked by the research community. To fill this gap, this paper studies in depth the software responsible for the platform update process across Android devices and the ecosystem of FOTA providers offering such services worldwide.

In this paper, we perform the first systematic analysis of Android FOTA components at scale. We characterize the underlying FOTA supply chain, and the privacy and security risks of these components. We focus on those providers making use of the standard capabilities offered by AOSP to implement FOTA components. We rely on the dataset of pre-installed applications collected by Gamba *et al.* [43] for our analysis. As of June 2020, this dataset contains 422,121 pre-installed apps collected from 40,165 users worldwide and covering 12,539 different devices. We extend the coverage due to the analysis of compiled DEX files (ODEX), and also complement this dataset with reputation logs and installation telemetry offered by a major security firm, NortonLifeLock. Our analysis leverages this dataset to make the following contributions aimed at better understanding the Android FOTA ecosystem:

- We develop a tool for automatic detection of apps implementing FOTA capabilities. This tool is based on code features and signatures extracted from official Android documentation, enhanced with a manual inspection of 18 FOTA apps from main Original Equipment Manufacturers (OEM) vendors and well-known third-party FOTA providers (§IV). This process allows us to obtain 4 main signals to detect FOTA components, and 11 secondary ones related to additional installation capabilities, with each signal showing varying levels of confidence. We also design a helper tool to recover DEX files from ODEX binaries, since many of the pre-installed apps present in the dataset are in this format. We automatically discover and classify 2,013 FOTA apps deployed in real-world devices. A significant number of these have dual behavior as they can (silently) install both system

and non system apps. Upon manual verification, we find no false positives (*i.e.*, wrongfully detected FOTA apps).
- We study the developer ecosystem behind FOTA apps to draw a picture of its supply chain. We observe that FOTA providers can be classified in four different categories: (i) OEMs, (ii) Mobile Network Operators (MNO), (iii) System On Chip (SoC), and (iv) Specialized FOTA Developers (SFD) (§V). We find that 43% of FOTA apps are developed by a third party and that there are generally multiple apps with FOTA capabilities in a single device, with some devices having as many as 5 FOTA-enabled apps. We also find a critical security issue: 40 FOTAs (in devices from 20 brands) are signed with test-keys part of the AOSP, thus allowing any app signed with the same key to gain system privileges.
- We statically analyze each discovered FOTA to identify potential security and privacy threats (§VI). We find the presence of social networks, advertising or tracking SDKs in 10% of the FOTA apps analyzed. We also observe potentially privacy-intrusive behaviors such as the sharing of GPS-level location data and unique device identifiers with online servers. Some FOTA apps use their own `SharedUserID`, thus allowing any app from the same developer to gain the privileges and permissions from the FOTA component. A common bad coding practice is the lack of verification of the downloaded updates, thus going against Google's recommendation. We also discover another critical vulnerability: 5% of the 1,747 devices where we find an `otacerts.zip` file (which lists the entities that can ship uploads) contain an AOSP default key, thus accepting any update signed with the same key.
- We complement our static analysis with telemetry data from a security vendor, NortonLifeLock, to analyze the behavior of FOTA apps in the wild (§VII). We confirm that FOTA apps, in addition to system updates, are used for secondary or commercial purposes—possibly for promoting third-party apps. We also find FOTA apps distributing unwanted apps, mostly Potentially Unwanted Programs (PUP). We detect that 92% of the apps installed by one FOTA app are malware.

Our findings confirm that FOTA apps might constitute an overlooked vector for security and privacy incidents. We consider this particularly critical because of its impact over a very large user base, as some FOTAs deliver updates to tens of millions of devices worldwide. We reported our findings to Google and the main vendors involved, and make our tools [11], [14] and aggregated dataset [5] available to the community.

II. Android system updates

In Android, system update capabilities are implemented by a privileged system component called the Firmware Over-The-Air (FOTA) app. Android has supported various mechanisms to update pre-installed software and other system- or vendor-specific components stored in the system partition. These mechanisms have evolved as new OS versions were released, but all of them rely on modifying the (otherwise) read-only system partition. While Android offers standard mechanisms, some OEMs implement their own ad-hoc implementations via custom vendor libraries and other privileged apps. There is not a clear set of rules that a FOTA app must follow. Google provides recommendations of the steps to follow in the update process [3], [20]: (1) retrieving update information from the update server (*i.e.*, description and URL of the update zip file); (2) downloading the update package; (3) verifying the package; (4) installing the update; and (5) rebooting the device into the new system. This section provides a historical overview and a description of the system update mechanisms implemented in the official AOSP.

A. Recovery system updates

This update process uses the Android *recovery* partition and starts by downloading a zip file from the update server. This zip file includes a patch script with the update instructions, a binary interpreter for the update file called `update_binary` (based on a template from AOSP [9]), additional files to be added to the system, a metadata file, and (optionally) a file with an updated set of signing keys. This is the recommended mechanism for devices running Android versions up to 7.0 to perform system updates [38], [50], as well as newer devices without two system partitions (see §II-B). For security reasons, the update file must be signed with the provider's key, which is later checked against the system OTA certificates stored in the `otacerts.zip` file by the system update API. The owners of the certificates included in `otacerts.zip` can thus install system packages as part of the update process.

Android provides FOTA developers with the `RecoverySystem` library [6] that implements this process. The library implements two key functions, (i) `verifyPackage()` which verifies the downloaded zip's signature against the certificates in `otacerts.zip`; and (ii) `installPackage()`, which calls the recovery service and writes into the bootloader control block. Developers may not use this library and perform installations by writing directly into `/cache/recovery/command`. In both cases, simple commands (*e.g.*, `--update_package` or `--wipe_data`) are written for *recovery* to apply requested changes once device is rebooted.

B. A/B seamless system updates

Android 7.0 introduced a new update mechanism known as A/B system updates or, simply, *seamless updates*. This method uses two separate disk partitions or *slots*: one where the system currently runs, called the *current* slot; and an *unused* slot, which is modified during the update process. Each slot has a number of attributes. The *active* attribute defines the slot from which the device can boot. After the update, the *unused* slot becomes *active* and, upon reboot, the bootloader will try to boot from it. If booting succeeds, an Android daemon called *update_verifier* marks the current active slot as *successful*.

This update process aims to ensure that there is always a workable booting system on disk during the update process. Thus, if the bootloader is not able to boot the new version, it can roll back to the old one. Another advantage is the possibility of updating the system while it is running, thus improving the usability. Finally, this process also allows for a streaming update where patches are applied directly to a partition while downloading, so no cache or extra data space is needed. The `UpdateEngine` class [2] provides the

API for seamless updates. The `applyPayload()` method interfaces with the `update_engine` Android daemon [1], which effectively applies the update and uses the `boot_control` HAL (Hardware Abstraction Layer) interface to reboot the device.

C. Projects Treble and Mainline

Device vendors are traditionally responsible for system updates. However, many vendors have not been able to ship updates at a reasonable pace, resulting in a substantial number of outdated and unpatched Android versions [53]. 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 [17], 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. Additionally, Project Mainline [15], launched in 2019 on top of Project Treble, allows to update core OS components through Google Play, similar to app updates. This allow critical security updates to be delivered without intervention from the manufacturer.

These mechanisms improves devices' security by reducing the time it takes to push an update. Furthermore, FOTA apps would no longer be needed for applying patches in AOSP code. Still, studying the ecosystem of FOTA apps is important and necessary, as these are needed for deploying HAL and vendors-specific updates. Also, only a few of Google's certified vendors (devices that passed a series of tests and fulfill a set of requirements [70]) implement this update mechanism at the time of this writing [15] (see Appendix A). FOTA apps are present in both certified and non-certified vendors with capabilities for installing anything on those devices. An analysis of these projects is out of the scope of this paper due to the lack of representative data, and it is left for future work.

III. Dataset

We use two complementary datasets in this paper: (1) Firmware Scanner's set of pre-installed apps, and (2) app reputation and installation logs obtained from NortonLifeLock. For clarity and consistency purposes, we will refer to *apps* as a unique instance of an Android app by its MD5 hash, and to *packages* as all app versions under the same package name.

Firmware Scanner: Most FOTA apps come pre-installed on Android devices, and are not available on public app stores. To overcome this limitation, we got access to the dataset of pre-installed apps collected by Gamba *et al.* [43] using Firmware Scanner, a purpose-built app publicly available on Google Play [13]. Firmware Scanner extracts pre-installed apps from the possible system partitions paths, (*i.e.*, `/system`, `/vendor`, `/oem`, `/odm` and `/product`). Preinstalled apps are crowd-sourced in a privacy-preserving fashion from volunteers without collecting any personal data from users. We refer the reader to the paper by Gamba *et al.* [43] for a technical description of Firmware Scanner and for a discussion of its ethical implications. As of June 2020, this dataset contains 422,121 pre-installed apps collected from 40,165 users, from 184 countries, with 26% of the users in Europe, 26% in America, and 40% in Asia according to the Mobile Country Codes (MCC) of the users. We discard 152,097 apps that were only found on rooted devices. In doing so, we aim at maximizing the validity of our results by focusing on devices that have not been tampered with by the user, or a malicious actor.

Reputation and Installation Logs: One critical aspect to consider in the characterization and analysis of FOTA apps is what type of software they are installing on user devices. We use a dataset provided by a NortonLifeLock that captures the presence of apps in real devices and the process responsible for their installation. The customers opted-in to share their data and the devices are anonymized to preserve their privacy. The dataset includes only app metadata and not the actual apps (*i.e.*, APK files). A reputation log record contains an anonymized device identifier, APK's SHA256 hash, package name, signer key, parent package name (potentially null) and, for a subset of those logs, the parent APK's SHA256 hash and signer key. The parent package information is obtained via the Android's Package Installer using the `PackageInstaller.getInstallerPackageName` method. As such, this dataset is limited to installations from FOTA apps that invoke the `PackageInstaller`. We further discuss how this limitation affects our analysis in §VII-A.

Customer devices regularly query a cloud-based reputation system to obtain reputation for the APKs installed on the device. This means that the client may query the same APK multiple times. To remove duplicated events we obtain the earliest date that a record is observed in a device and use it as an approximation of the installation time of the app. In total, this dataset contains 1.6 B installation events from 19.3 M Android devices collected from January to December 2019. Prior work has used the same dataset to analyze distribution vectors of unwanted apps [48]. In this work, we extend the dataset to contain installation events for a 12-months period.

IV. Discovery of FOTA apps

Detecting FOTA apps is a challenging process, as these apps can differ in their implementation details. FOTA apps may not have distinctive package names, their functionality may be split among multiple apps, or may contain customized code. Thus, we design *FOTA Finder*, a tool to automatically identify and classify FOTA capabilities in a given APK file. In this section, we first describe *FOTA Finder* (§IV-A) and its limitations (§IV-B). We then present the results obtained over our dataset of pre-installed apps (§IV-C), and validate the accuracy of this discovery process (§IV-D).

A. FOTA Finder

We begin our process by extracting FOTA-specific code fingerprints from Google's FOTA documentation and manually analyzing the code of 18 well-known FOTA apps. To collect these apps, we search the Firmware Scanner dataset for apps with descriptive names—*i.e.*, package names that contain terms such as `update` or `fota`. Armed with a complete list of code fingerprints, we build *FOTA Finder* to automatically classify a given APK as FOTA or not. The tool is based on Androguard [36], an Android analysis framework that offers

Signal	Description
RV	Call to the method `verifyPackage` from the API class `android.os.RecoverySystem`
RI	Call to the method `installPackage` from the API class `android.os.RecoverySystem`
CMD	Use of the strings "`/cache/recovery/command`" and "`--update-package`" in the code
A/B	Call to the method `applyPayload` from the API class `android.os.UpdateEngine`

Table I: Features for detecting FOTA apps.

APIs to facilitate static analysis of Android apps. For each APK, *FOTA Finder* first detects the compiled code in DEX (Dalvik EXecutable) format that is typically included in one or more `classes.dex` files. If an app uses ahead-of-time optimized code, *FOTA Finder* parses the ODEX file and extracts the original DEX file(s). Since the ODEX file format is not publicly documented (the only source being the *dex2oat* tool [59]), we go through a reverse engineering effort to develop a tool for DEX extraction, Dextripador [11]. Finally, we parse the DEX code for specific method calls and strings related to FOTA system updates.

Table I summarizes the key FOTA detection features (flags) used to detect the mechanisms provided by the Android platform to install new apps in the system partition (§II). These features cover both the old update procedures based on the `RecoverySystem` as well as the newer A/B system updates. If an app has at least one of these signals, *FOTA Finder* will categorize it as FOTA.

Detecting OTA capabilities. During the manual analysis of FOTA apps, we identified other signals associated to behaviors implemented by OTA apps—*i.e.*, apps that can install and remove user (*i.e.*, non-system) apps under the `/data` directory. *FOTA Finder* also uses these signals, since they allow us to identify other package installation capabilities in FOTA apps. We provide a complete list of OTA features along with further details in Appendix B.

B. Scope and Limitations

We identify three main technical limitations in *FOTA Finder* during our design, development, and exploratory research efforts. First, our tool can only FOTA apps that follow Google's recommendations and use FOTA-specific API functions. It cannot detect FOTA apps that follow custom implementations (*e.g.*, use of custom services, or native code). Nevertheless, we also look for the presence of `/cache/recovery/command` and `--update_package` strings, as their usage are part of the internal implementation of an update for recovery [50].

Second, *FOTA Finder* fails to parser ODEX code for 37% of the apps. This happens because of the lack of documentation of the ODEX file format, and changes on the internal structure can be wrongly handled by the *FOTA Finder* parser.

Finally, our detection of FOTA apps relies exclusively on static analysis of DEX code. This is problematic because FOTA apps may use dynamic code loading, reflection, native code or obfuscation of strings. This can cause *FOTA Finder* to wrongly classify FOTA apps as non-FOTA. However, we resort to static analysis only, as running pre-installed apps in a sandbox environment at scale remains an open problem.

C. FOTA Finder *Results*

We run *FOTA Finder* on 422,121 pre-installed apps found in non-rooted devices from 40,165 users in the Firmware Scanner dataset (§III). *FOTA Finder* could not process 37% (154,922) of the apps. All of these errors are due to the usage of ODEX files by these apps. In total, *FOTA Finder* detects 2,013 FOTA apps in 20,924 devices. 24% of these apps are using ODEX code, which *FOTA Finder* managed to parse successfully.

An analysis of the FOTA apps package names shows that 32% (647) of them do not contain any string tokens that may reveal the purpose of the app (*e.g.*, `update`, `install`, `fota`). We also find that 93% (1,878) of the FOTA apps rely on the `RecoverySystem` API for the update process, while the remaining 7% (135) support the A/B system updates. 32% (651) of the FOTA apps can also perform installations at the user level. We investigate these capabilities in detail in §VII, as FOTA apps can use them to install malicious apps.

D. FOTA Finder *Evaluation*

We consider acceptable for *FOTA Finder* to miss some apps (False Negatives, FN) due to its technical limitations (§IV-B). Our methodology offers sufficient coverage of FOTA apps to analyze the ecosystem and draw general conclusions. We therefore focus on reducing False Positives (FP), *i.e.*, non-FOTA apps missclassified as FOTA, since these can introduce bias and affect the validity of our results. We thus consider *FOTA Finder* as a best-effort yet accurate approach that does not aim to be complete. To this extent, we focus our evaluation on detecting potential FPs, mainly in two ways. First, we perform a manual review of a subset of 50 FOTA apps. Second, we search Google Play Store for the presence of FOTA apps. Due to the intrinsic characteristics of FOTA apps (*e.g.*, they must be installed on a read-only system partition) we do not expect to find these apps on the Play Store market, as Google Play policies prohibit apps with the ability to install others [41], [56] (concrete wording in Appendix E).

Manual review: We validate our method by manually investigating a subset of 50 FOTA apps randomly chosen. We chose these apps using the features in Table I, and include `RecoverySystem` based, *A/B system update* based and *command based* FOTAs. One of the authors manually classified them as FOTA apps, discovering three apps that are potential FPs. However, a closer look on those apps reveals that these are not fully-fledged FOTA apps, but part of a larger FOTA system composed of multiple apps, each one of them responsible for different phases (*e.g.*, download, verify and install the packages). Specifically, a FOTA (`com.samsung.sdm`) verifies the update package using the method `RecoverySystem.verifyPackage`, but relies on a native library (`libmno_dmstack.so`) to apply the update. Another app (`com.qualcomm.qti.loadcarrier`) applies the verification but relies on the service `CarrierAccessCacheService` from another package for the actual installation. Finally,

com.zte.zdm relies on an custom update method (thus, not implemented in *FOTA Finder*), which uses an intent with the action android.intent.action.RECOVERY_REBOOT with the specific parameters to perform the update. Accordingly, we do not consider them as actual FPs and we do not remove them from our dataset.

Samsung is an interesting example of the use of native code: we find a total of 149 devices with a FOTA app, but also the libmno_dmstack.so library. Manual analysis of the library reveals that the update is implemented by writing directly into /cache/recovery/command. Even though there is no enforcement on Google's side, none of its recommendations use native code for system updates.

FOTA apps in the Play Store: We search all FOTA package names in Google's Play Store from Madrid, Spain on the June 24, 2020. We rely on a purpose-built crawler that uses the package name (indexing field on Google Play) to identify the presence of an app in the market. We detect 7 FOTA packages that are available in the store, including Google Play Services. Although Google Play Services does not perform any system updates, it still updates Google related apps that often come pre-installed. We review the descriptions and certificated of the other six apps and verify that these apps are indeed FOTA apps, used for system updates of specific phone models and released by the actual device vendor. We lack sufficient insights to explain why these apps were accepted on Google Play despite Google's Terms of Service [56].

V. Ecosystem

Developing and deploying FOTA apps, as well as operating the updating infrastructure, are critical parts of the Android supply chain. While some OEMs might keep these processes in-house, others rely on third-party FOTA suppliers for some or all of the steps. The number and relationships between the stakeholders involved in the firmware update process are generally unknown, and it is not always possible to determine their identity due to the lack of accurate attribution signals. In this section, we leverage *FOTA Finder*'s results to explore this ecosystem. First, we identify the different stakeholders present in the deployment of FOTA apps (§V-A). Second, we analyze their prevalence across different devices and brands (§V-B).

A. FOTA stakeholders

One critical aspect of the FOTA supply chain is identifying the company or organization responsible for building and deploying the FOTA component. A number of technical challenges prevent us from reliably performing authorship attribution in the Android ecosystem. This is, to a great degree, the result of a lack of a public key infrastructure (PKI) to verify the legitimacy of the certificates used to sign apps. This issue is particularly critical on pre-installed apps [43] as they lack the developer metadata that can be found for regular apps on app stores. Furthermore, confusion is added when brands and developers use multiple organization unit names within their products (*e.g.*, *Samsung Corporation vs. Samsung Electronics*) or when they use generic names such as *Android* [61].

Despite these limitations, we analyze FOTA package names and certificates to identify the companies responsible for the deployment of FOTA apps and, if possible, their developers. We assume that the company that signs a FOTA package is the one behind its deployment. We also check the FOTA package name and, if it contains a company name that differs from the signer, we assume that the app is developed by the company that appears in the package name and deployed by the company that signs it. We combine the methodology proposed by Sebastian *et al.* [61] with the device brand as reported by Firmware Scanner. To get the organizations from the certificates, we rely on the *Organization field (O)* and the domain of the *Email* from the subject's *Distinguished Name (DN)*. From this analysis, we find 269 unique certificates that sign FOTA apps, belonging to 219 subjects from 127 organizations. The spectrum of organizations in the certificates is wide, ranging from certified OEM vendors such as *Samsung* to MNOs such as *Vodafone*.

To better explore the FOTA providers landscape, we classify them by their type of company following a semi-manual snowball sampling method [45]. We performed web searches to identify unknown companies that remained unclassified. This process allow us to identify the following categories, presented by number of package names detected for each category and the percentage of apps within the 2,013 FOTAs detected: (*i*) OEMs: 53% of FOTA apps from 77 different packages; (*ii*) SoCs: 9% of FOTAs apps from 13 packages; (*iii*) SFDs: *e.g.*, *Adups* or *Redstone*: 9% of FOTA apps from 13 packages; (*iv*) MNOs: 1.6% of FOTA apps from 4 packages. Additionally, there are 15% FOTAs from 2 Google packages (com.google.android.gms and com.google.android.gsf). We could not find information for 12% FOTA apps from 24 packages, which we label as Uncategorized (UNC). This analysis reveals that the FOTA ecosystem is rich and goes beyond just OEM vendors.

While one would expect a one-to-one mapping between package names and signatures, we find that this is not the case in the FOTA landscape: 49 (37%) packages are signed by 2 or more different organizations. This is due to different organizations developing the FOTA and deploying them in the devices. The most extreme case is a single package, com.adups.fota.sysoper, signed by 60 different organizations. *Adups* is a FOTA software development company whose products are integrated in (mostly low-end) smartphones [19], [47]. Organizations signing this package include OEMs such as *Konka*, *Tinno* and *Wheatek*. In many of these cases, this a mandatory requirement in order to acquire the system shared UID as we will show in §VI. Another package, com.mediatek.systemupdate.sysoper, is signed by 33 different certificates from various organizations, including OEMs like *Oppo*, *Lenovo* and *HTC*. *Mediatek* is a SoC manufacturer, and the presence of one of its FOTAs might be required to update specific firmware. However, as we describe in §V-B and §VII, some FOTA apps from SoC vendors do install apps available on public markets.

Security implications: The lack of control over the FOTA signing process has attribution implications but also security ones. We find 40 FOTA apps (2%), corresponding to 13 different packages that are signed with default (thus well-known) test-keys released as part of the AOSP. At least 171

Package	# dev.
com.adups.fota.sysoper	98
com.mediatek.systemupdate.sysoper	16
pl.zdunex25.updater	13
com.abastra.android.goclever.otaupdate	11
com.mediatek.googleota.sysoper	10
com.redstone.ota.ui	8
com.freeme.ota	6
com.fw.upgrade.sysoper	4
com.fota.wirelessupdate	3
org.pixelexperience.ota	3
com.android.settings	2
com.adups.fota	1
com.rock.gota	1

Brand	# dev.
Alps	80
Xiaomi	16
Samsung	12
Goclever	11
Allview	10
Doogee	9
Iku	8
Blackview	6
Bravis	6
Cubot	3
Elite_5	2
BQ	2
Others (9)	11

Table II: FOTA Packages (left) signed by default keys from AOSP and Top brands (right) affected.

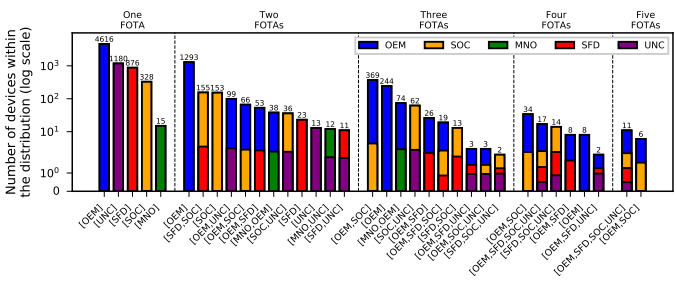

Figure 1: Distribution of FOTA types in devices with 1 to 5 FOTAs. To ease visualization, we show non-Google packages with a higher prevalence in our dataset.

devices from 20 brands present this issue as listed in Table II. The use of such keys is discouraged in the FOTA development guidelines defined in the Android official documentation due to their concerning security implications [30]. Examples of these risks are the replacement of legitimate app as an update or, in case of misconfiguration of the `sharedUID` and components permissions, another app running in the same process memory and getting access to FOTA files.

B. Prevalence

We study the prevalence of FOTA apps across devices and vendors using the device metadata provided by Firmware Scanner. We observe that the set of 133 packages are distributed over 395 different brands. While we find most packages (67%) in devices from the same brand, some devices ship apps from multiple brands. As expected, `com.google.android.gsf` is pervasively present in over 95% of the devices in our dataset. Similarly, `com.google.android.gms` is present in devices from Google-certified OEMs (18%). Additionally, we find high-prevalent cases such as two FOTAs from *Adups* and *Mediatek*, which appear in 578 and 336 devices from 111 and 34 different brands, respectively.

It is difficult to discern whether a given FOTA is developed by the OEM manufacturing the device by just looking at the information available on the certificate. However, by analyzing the relationships between the package names, the OEMs, and the organization signing the FOTA apps, we can gather more insights about the distribution of some particular FOTA packages in Android devices. We refer the reader to Figure 3 in Appendix C for a graphical illustration of this distribution. We observe two different patterns:

1) First-party FOTAs in which there is a consistent relationship between packages, brands, and certificate information (*e.g.*, the `com.samsung.android.app.omcagent` app found on a *"Samsung"* device, signed by *"samsung corporation"*). Note that in some cases the brand reflects MNOs due to the re-branding of devices, *e.g.*, *"verizon"* devices that are actually updated by a FOTA app from *"samsung"*.
2) Third-party FOTA providers like *Adups*, *Redstone* and SoC manufacturers like *Qualcomm*, which are present in devices from multiple vendors, and often signed with different certificates as discussed before.

We discover that 42% of FOTA apps (from 47 packages) are first-party FOTAs, all of them from the OEM category, while 43% of apps (from 84 packages) are third-party FOTAs. Specifically, 10% of apps from 30 packages are from OEMs,[1] 1.6% of apps from 4 packages are from MNOs, 9% of apps from 13 packages are from SoCs, and 9% of app from 13 packages belong to the SFD category. The rest of the FOTAs are either not categorized or belong to Google.

Despite the limitations of this analysis, the results show that a majority of FOTA software from our dataset is not deployed by the OEMs. Instead, they often rely on third parties, either for updating the whole system or for updating the software required by the different hardware components. The externalization of FOTA services translates into vendors losing control over critical system components with the ability to silently install software in the devices. This may introduce privacy concerns, as it is not generally possible to know who is providing a device with updates or if external manufacturers can access sensitive data. This lack of control also has security implications as users have no control over what is installed using a FOTA. For example, malicious apps can be installed without user knowledge [19]. We go further into the analysis of installations through FOTA software in §VII, using data from real devices.

We observe that 80% of the devices have Google FOTA components. If we exclude those components, we find 71%, 20%, 8%, 0.8%, and 0.2% of devices with one, two, three, four and five FOTAs, respectively. A plausible reason for having more than one FOTA is to update packages for different purposes—*e.g.*, the system itself, hardware drivers, or other apps. Figure 1 shows the distribution of FOTAs per device in our dataset, grouped by their type. We report statistics for devices having 1 to 5 FOTAs. For clarity reasons and to reduce combinations, we aggregate repeated types (*e.g.*, we reduce `[SOC, OEM, OEM]` and `[SOC, SOC, OEM]` to `[SOC, OEM]`). In general, we observe that in most of the devices with one or two FOTAs, these are from OEMs (66% and 66% respectively). In absence of an OEM FOTA, it is common to observe either a FOTA from a SFD or SoC. Indeed, often combinations of FOTAs from different categories include one from a SoC. This suggests that these devices include FOTAs used to update specific hardware components, and FOTAs used to update the system itself. A less common type of FOTA found is MNOs.

[1]Note that some OEMs, like *Foxconn*, *JRD* or *Tinno*, also provide FOTA services to other brands (possibly due to re-branding).

Those cases where a FOTA of this type is present might be due to MNOs willing to take the control over the devices by installing specific updates and system apps (not allowing the user to remove them), like app centers or commercial apps.

Limitations. Our analysis is based on 2,013 FOTA apps extracted from 20,924 devices. Due to limitations on *FOTA Finder* (§IV-B), we were not able to process 37% of the Firmware Scanner dataset. A larger coverage might change the distribution of FOTA types. Also, we analyzed the FOTA categories using static lists that were filled by doing manual web searches, in order to classify the different companies based on the package names. These limitations may impact our results. For example, we found devices where only SoC FOTAs are present (devices like *Xiaomi* or *Smartfren* with FOTAs from *Qualcomm* or *SPRD*). One would expect that each device contains at least the corresponding OEM FOTA (or a specialized developer). In cases where neither of these are found, this might be due to: (i) *FOTA Finder* not being able to find the FOTA corresponding to the OEM or SFD in these devices, or (ii) due to limitations in the classification, *e.g.*, an OEM wrongly labeled as a SoC developer.

VI. Behavior analysis

We statically analyze the FOTA apps in our dataset to characterize their capabilities, privacy risks, and identify means to execute potentially harmful or unwanted behaviors. We also perform an analysis of the different aspects relevant to the installation process and other artefacts present both in devices (specific certificate repositories) and FOTA apps (*e.g.*, shared UIDs) that provide additional information about the stakeholders forming the update ecosystem and their relationships. We note that our findings only show *potential* behaviors as we do not have runtime observations. We might also be missing some behaviors as a consequence of code obfuscation techniques or reflection [35], [42], [54]. This is the result of well-known static analysis limitations.

A. Overview

We begin our study with a general analysis of the DEX code of the 1,716 non-Google FOTA apps found on non-rooted devices to identify the presence of potentially harmful behaviors, including the access to sensitive resources and personal data. We rely on a custom analysis pipeline that integrates multiple open-source static analysis tools to elicit behavior in Android app, such as FlowDroid [31] and Amandroid [67] to conduct taint analysis, and a modified version of Androwarn [10] to perform API usage analysis. We are unable to automatically study dynamically FOTA apps in a standard analysis sandbox due to technical and instrumentation impediments, including the need for platform signatures on the code, dependencies on hardware, and different entry points than those often expected by the sandbox (see §VIII). Consequently, our results might miss hidden behaviors that are elicited through techniques such as reflection and dynamic code loading.

Table III provides a summary of our results. We classify the access to 36 different types of sensitive data, resources, or capabilities into 9 categories. The majority of FOTA apps show capabilities that are expected to implement their function. For example, most apps access the current network and phone state. This is expected since Google's documentation [3] recommends that FOTA updates should be scheduled when the device is in idle maintenance mode (*e.g.*, at overnight, when phone is charging) in order to avoid disruptions, and also when it is connected to a Wi-Fi network to avoid monetary costs due to downloading updates through a data plan. However, our findings suggest a prevalent access to user and device identifiers. A small number (<5%) of the analyzed FOTA apps access the device location. Table IV shows, for a subset of FOTA apps, the types of device and user data that is accessed and uploaded to the update servers. The purpose of uploading such identifiers is unknown, though one plausible hypothesis is that they facilitate targeted installations programs. The fact that these apps run silently in the background indicates that personal data is likely uploaded without user consent, showing a lack of transparency in these apps. Other potentially dangerous behaviors include the ability to make phone calls (146 apps) and recording audio (44 apps) or video (40 apps). However, as we will further investigate in §VI-C, their usage is legitimate in most cases.

In the remaining of this section we take a deeper look at some of these potentially dangerous behaviors, with emphasis on the presence of third-party components in FOTA apps and the type of permissions that are requested by FOTA apps. When relevant, we will contextualize our findings with the type of FOTA apps according to the classification in §V-A.

Accessed data type / behaviors		% Apps (#)	% Third-party (#)
Telephony identifiers	IMEI	33.7 (577)	15.2 (260)
	IMSI	31.4 (538)	8.2 (140)
	Phone number	8.8 (151)	4.4 (75)
	MCC & MNC	19.1 (327)	6.3 (108)
	Operator name	5.7 (98)	3.3 (56)
	SIM Serial number	6.5 (111)	2.7 (446)
	SIM State	13.1 (224)	4.5 (77)
	Current country	6.7 (115)	1.3 (22)
	SIM country	7.6 (131)	3.2 (55)
Device settings	Software version	1.0 (17)	1.0 (17)
	Phone state	25.1 (430)	5.5 (95)
	Installed apps	49.2 (843)	17.9 (307)
	Phone type	14.4 (247)	8.3 (143)
	Logs	65.3 (1,119)	24.8 (425)
Location	GPS	0.7 (12)	0.6 (11)
	Cell location	4.3 (73)	2.7 (47)
	CID	4.8 (82)	2.6 (44)
	LAC	3.7 (63)	2.0 (34)
Network interfaces	Wi-Fi configuration	2.0 (35)	1.9 (32)
	Current network	50.0 (856)	15.1 (259)
	Data plan	34.9 (598)	8.9 (153)
	Connection state	4.3 (73)	1.7 (29)
	Network type	17.3 (296)	6.2 (106)
Phone service abuse	SMS sending	0.1 (1)	0.0 (0)
	Phone calls	8.5 (146)	3.3 (57)
Audio/video interception	Audio recording	2.6 (44)	2.4 (41)
	Video capture	2.3 (40)	2.3 (40)
Arbitrary code execution	Native code	27.1 (465)	11.4 (196)
	Linux commands	30.9 (530)	10.8 (185)
Socket conn.	Remote connection	6.7 (114)	1.9 (32)

Table III: Percentage of apps (out of the 1,716 FOTA apps analyzed) accessing personal data or showing potentially harmful behaviors. The rightmost column shows how many of them are third-party FOTAs.

Package	Device Model	MAC Address	User ID	Country	Android version	Fingerprint	System Language	Apps related information	Rom usage	Ram usage	System brightness	Network type availability	Battery level	Charge count	Discharge count	Running time	Running process info	CPU info	IMEI	MEID	IMSI	Number of sms	Number of contacts	Number of Calls
com.sonyericsson.updatecenter	•		•	•	•	•	•												•					
com.motorola.ccc.ota	•		•	•	•	•	•		•	•		•	•						•					
com.redstone.ota.ui	•	•		•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
com.adups.fota	•	•			•	•														•				

Table IV: Data sent to the update server in selected FOTA apps.

B. Third-party components

It is common for Android apps—including pre-installed ones [43]—to embed third-party SDKs (Software Development Kits) in order to include functionality from external sources, such as networking support or advertisement and analytics services [51], [57]. In Android, however, any component embedded in an app runs with the same permissions as the host app. As a result, since FOTA apps are highly privileged (as we will discuss in §VI-C), the presence of libraries offered by companies with data-driven business models would be concerning. These instances might reveal access to sensitive permissions for secondary purposes like user-tracking or advertising.

We rely on an enhanced version of LibRadar [51] to identify all embedded libraries in an app. To maximize the soundness of our results, we only consider those libraries identified by LibRadar with a confidence of 100%. We expect FOTA apps to include SDKs that are non-privacy invasive, such as those related with development support, network protocols or database drivers. Therefore, we report only those SDKs that are related to analytics services and advertisement libraries, *i.e.*, those more likely to collect personal data for secondary purposes [57].

Table V summarizes our findings. In total, we find 8 SDKs related to social networks, user tracking, and advertisement in 171 (10%) FOTA apps. While this number is lower in comparison to what has been reported for "regular" Android apps [52], [57], it is still important to study the presence of SDKs in FOTA apps because of their privileged position. The most common SDK is Firebase, which presents a lot of different functionalities (from analytics support to storage related features). On its own terms of service, Firebase declares that it collects personal data and that it acts as a data processor [46] (concrete wording can be found in Appendix D). Finding privacy policies from pre-installed apps is hard, as they cannot be found on app markets and they often lack their own policy, in fact, we could not access them for this project. Therefore, users will have a hard time to learn that this type of third-party components are part of FOTA apps. Interestingly, we find that Google Ads is also featured in FOTA apps, suggesting that some providers might try to generate advertisement revenue. We do a manual review of several versions of the `com.motorola.ccc.ota` OEM FOTA app in which Google Ads is embedded. We find that the app includes functionality from the Google Mobile Services SDK which relies on some code from the Google Ads SDK. We also find SDKs that are more common in the Asian developer ecosystem, such as Umeng or the social network SinaWeibo. We find that these are more prevalent in apps found in phones from *Lenovo* and *Nubia*. Manual analysis of the app's code suggests that the OEM FOTA app `com.lenovo.ota` uses features related to the social network capabilities of SinaWeibo (*e.g.*, parsing an access token) and that the OEM FOTA app `cn.nubia.systemupdate` uses the analytics capabilities of Umeng. Manual analysis also suggests that the `com.coloros.sau` OEM app uses the crash report capabilities of the Tencent SDK. We next analyze whether these SDKs request access to dangerous permissions or if they piggyback from those requested by the host app.

SDK	# apps	Type
Firebase	133	Development, Analytics, Push, Storage
Google Ads	23	Advertisement
Umeng	8	Analytics
SinaWeibo	6	Social Media
Tencent	4	Social Media
Millennial Media	1	Advertisement
New Relic	1	Analytics
Fabric	1	Analytics

Table V: Social networks, analytics and advertisement SDKs

C. FOTA Privileges

FOTA apps come pre-installed in the device system partition. This means that they can enjoy system privileges—*i.e.*, they cannot be easily uninstalled and they can have access to system protected data and resources [43]. In order to understand the privileged access that FOTA apps have, we first analyze how common it is for apps to have the same user ID (UID). Android allows apps with the same UID to access each other's data and run in the same process, provided that they are signed with the same certificate. Analyzing shared UIDs is relevant since it implies that FOTA apps might be part of a larger code base with access to more functions and permissions, including privileged ones. Then, we analyze the type of permissions that are requested by FOTA apps (and their embedded SDKs) as a proxy to understand the type of data that they might collect. We extract requested and declared permissions and shared UID from the app's manifest.

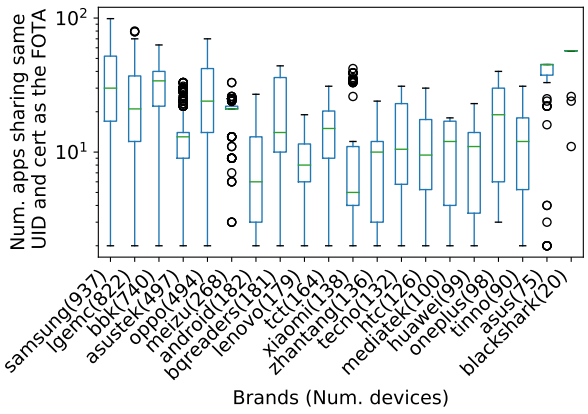

Figure 2: Organizations and distribution of other applications that share the `UID` and are signed with the same certificate as FOTAs in the same device

Shared User IDs: We find a shared `UID` value for 1,737 FOTA apps from 80 different packages, with most of them (60%) having the `android.uid.system` value. This implies that, if such apps are signed with the platform key, they are effectively "the system" and have access to all code and permissions than the superset of all system apps. Interestingly, some FOTA apps share `UID` with apps such as the `android.uid.phone` (29 FOTA apps from *"qualcomm"* and 3 from *"BBK"*), `android.uid.nfc` (one FOTA from *'samsung'*) or `android.media` (one FOTA app from *'gigaset'*). Though this feature (shared `UID`) was recently deprecated in API level 29, it has been widely used by system apps in the past, thus affecting millions of Android devices that do not run newer Android versions. Figure 2 shows, per organization, the number of devices where we observe non-fota apps with the same User ID as a FOTA app as well as the distribution of these apps across devices. Technically, if FOTA apps expose their functionality and data as external services, they would be making them available to non-FOTA apps.

Permission analysis: In order to understand the type of data and resources that FOTA apps might access (and possibly make available to embedded third-party SDKs or to non-FOTA apps via shared `UID`), we study the permissions requested by these apps. To do so, we parse the Android Manifest file to extract requests for both AOSP permissions (*i.e.*, those that are officially released by Android) and custom ones (*i.e.*, those declared by vendors or developers to enable access to functionality/data from other apps [43], [62]). For apps that specify a `sharedUserId` attribute (70% of the FOTA apps in our dataset), we also include official permissions requested by applications that share the same `UID` also found on the same device and signed by the same certificate, to reproduce what happens on real devices. We find that FOTA apps request a median number of 25 AOSP permissions. It is expected that FOTA apps request some permissions, as they enable functionalities that are part of the functionality of these apps. For instance, we find that access to the Internet, Internet connection data, external storage of the phone and the ability to reboot are all requested by over 70% of FOTA apps. Nevertheless, there are other permissions that can be considered potentially dangerous for user privacy, and that might not be part of the expected functionality of a FOTA app.

Dangerous permissions [22] are among the most requested by FOTA apps. This is worrisome as from Android 8, manufacturers can "whitelist" privileged apps and list permissions that should be granted to them without user interaction [24]. The first column in Table VI shows, for some examples of dangerous permissions, the percentage of apps that request them. In particular, 21% of FOTA apps request access to `ACCESS_COARSE_LOCATION` and 15% to `ACCESS_FINE_LOCATION`. These permissions could give apps the ability to customize the list of apps to install based on the location of the users. We also find that 70% of FOTA apps request the `READ_PHONE_STATE` permission, which gives access to the IMEI (a unique identifier that facilitates tracking).

To better understand the intended use (*i.e.*, primary or secondary usages) that FOTA apps make of each request, we manually analyze the code that invokes permission-protected methods. Interestingly, some FOTA apps (*i.e.*, "test/engineer mode" apps) request a high number of permissions because they have the ability to test different functionalities from the phone (*e.g.*, the GPS manager, the camera or call features). Examples of such apps are `com.vivo.bsptest`, `com.lge.hiddenmenu` and `club.dexp.dmfct2`, which are all OEM FOTA apps.[2] While these behaviors are not privacy invasive, we also find examples in which access to sensitive data might result in privacy risks for users. For instance, the `com.redstone.ota.ui` SFD app includes a JSON with information such as the location of the user, its mobile carrier, or the system language when querying an online server for sync information.

We study whether these permissions are requested for secondary purposes (*e.g.*, user-tracking or advertising via third-party SDKs). For the automatic analysis we rely on Cartographer [33] to analyze whether a given API call is triggered by the app's code or by a third-party SDK. Then, we map these API calls to the permission that enables them by using custom mappings that complement the mappings generated by Axplorer [32]. Specifically, we fetch mappings by parsing Android Studio [29] and the `@RequiresPermission` tag present in the AOSP code [26], [27]. Our analysis shows that, even though we find evidence of third-party SDKs related to advertisement and tracking in FOTA apps, these rarely access any dangerous permissions. Table VI shows for each dangerous permission found in FOTA apps, how often it is used in the app's code only, on code from a third-party SDK only, or in both. We also show between brackets how many of these SDKs are related to tracking services (*i.e.*, social networks, analytics and advertisement SDKs). We find that most dangerous permissions are only accessed by the app itself and that most third-party SDKs that access a dangerous permission are not related to advertisement or analytics services (*i.e.*, they are development support libraries such as Volley [28] or Leak Canary [49], which in principle are not dangerous for the privacy of users). We still find examples of analytics

[2]In order to test the call feature, this app calls an emergency number, using critical resources for non-critical situations.

Permission	% apps	App	App & SDK (Tracking)	SDK (Tracking)
READ_PHONE_STATE	70%	90%	9% (2%)	1% (0%)
ACCESS_COARSE_LOCATION	21%	78%	10% (0%)	12% (0%)
SEND_SMS	16%	88%	0% (0%)	12% (0%)
ACCESS_FINE_LOCATION	15%	76%	5% (0%)	18% (0%)
GET_ACCOUNTS	10%	78%	17% (0%)	5% (0%)
READ_SMS	7%	99%	1% (0%)	0% (0%)
READ_PHONE_NUMBERS	1%	99%	1% (0%)	0% (0%)

Table VI: Percentage of apps requesting a given permission, and how often only the FOTA app, a third-party SDK, or both use it in the code.

SDKs leveraging these permissions, such as the "New Relic" SDK accessing the location in the `com.bqreaders.ota` OEM app or "Tencent" and "Firebase" using the permission to access unique identifiers in both the `com.coloros.sau` and `com.samsung.android.app.omcagent` OEM apps.

Finally, we take a look at the custom permissions requested by FOTA apps. This type of permissions are declared by third-party actors (such as other app developers or OEMs) to protect the access to some exposed functionality [43], [62]. We find a total of 401 unique custom permissions by their name. We find that FOTA apps request a median of 6 custom permissions. We find examples of permissions declared by phone vendors (*e.g.*, `com.lenovo.permission.READ_SEARCH_INDEXABLES`), and others related to different Google services (such as `com.google.android.googleapps.permission.GOOGLE_AUTH`). This indicates that FOTA apps rely on functionality outside the AOSP permission model, provided by external sources.

D. Potential for Abuse in the Update Process

We next analyze whether the FOTA apps in our dataset follow Google's recommendations [20] for the verification and installation of the updates (steps 3 and 4 of the FOTA update lifecycle process introduced in §II). We also analyze the organizations that are involved in these processes by looking at the relevant verification certificates. Due to our limitations to conduct dynamic analysis, we do not analyze other steps of the update process, *e.g.*, data exchanges with the update servers.

Most FOTAs in our dataset (93%) use Google's `RecoverySystem` library to implement system updates. These include those running on a device pre-Android 7.0 (where there are no other alternatives) and devices with newer versions that do not support A/B updates (see §IV). While this library is available and easy to use for package verification and installation, developers may decide to implement their own custom update mechanism. During the validation of *FOTA Finder* (§IV-D), we found cases where the FOTA app was using the verification component, but not the actual installation API which was done by other means. In this section, we leverage the signals from *FOTA Finder* to analyze the capabilities of all *non-A/B* FOTAs in terms of whether they do install, verification, or both. Specifically, Table VII shows the prevalence of apps that have installation capabilities (either using Google's recommendation, *i.e.*, the `installPackage` API method, or writing in `/cache/recovery/command`), and verification capabilities (using the `verifyPackage()` API).

We observe that 8% of all the *non-A/B* FOTA apps do use the verification, but we have not found evidence of how they install packages. Such apps might rely on other apps to perform the installation (*e.g.*, through an `Intent`) or on other component (*e.g.*, native code or via reflection). Also, 45% do have installation flags but do not use the `verifyPackage` method from the `RecoverySystem` library. This could mean either that no verification is applied, or that a custom verification is implemented. In an attempt to discover if a custom verification method is being applied, we looked for the API `MessageDigest` in specific packages where the update process occurs in the FOTA apps. Our analysis retrieves all the calls to this API and the hash algorithm used. We observe that, for the 949 that do not verify the updates using Google's API, 51% contain a call to `MessageDigest`, mostly using the MD5 hash algorithm, suggesting that they perform their own verification process.

Either if the verification is not done by any means at the high-level Android code, we observe that the `recovery` binary always applies the verification step as a security measure before applying the update [7]. This verification relies on the default `otacerts.zip` file [8], and thus owners of the certificates included in this file are authorized to install packages in the system. We rely on the Firmware Scanner dataset to inspect this certificate list, obtaining 3,311 `otacerts.zip` files from 1,747 devices. From these, 598 (6%) also contain a FOTA app.[3] In general, we find that the same organization that signs the FOTA app is present in the `otacerts.zip` file from the same device. In these cases, the same organization signing the FOTA is also responsible for the updates. While the majority of apps are related to a single certificate in the file, 8 FOTA developers use two different `otacerts.zip` files. Our manual analysis shows that most of these are from the same organization. Yet, there are 12 FOTA developers for which we do not find a single corresponding certificate in the `otacerts.zip` (*i.e.*, they are not responsible for the update process). In some cases, these FOTAs are related to third-party FOTA providers like *Adups*, which are developers but do not manage the update process. Finally, it is noteworthy the presence of 5% devices that include an AOSP default key in the `otacerts.zip` file, affecting 17 brands. These devices are mostly from the brand *Alps* (57%) and *Goclever* (12%), but we find devices affected from brands such as *Huawei* (3%), *Toshiba* (3%) or *Samsung* (1%). The use of AOSP test keys renders the verification useless, and it might allow non-authorized parties to install system apps.

VII. Telemetry analysis

In this section we complement the analysis of FOTA packages from previous sections with dynamic behaviors observed on real devices. Since sandbox dynamic execution of pre-installed apps remains an open problem, we rely on empirical evidence obtained from the telemetry of a commercial security tool that tracks the installation of Android apps (see §III for details). This section first describes how we detect FOTA

[3]The low ratio of certificates is due to: (i) the lack of support in FirmwareScanner to identify and extract the file, since their developers instrumented this feature on November'19; and (ii) limitations of *FOTA Finder* (§IV) to extract FOTA apps for some devices.

INSTALL		VERIFY	#APKS	%APKS
API	CMD			
✓	×	✓	495	26.36
✓	×	×	459	24.44
×	✓	×	454	24.17
×	✓	✓	249	13.26
×	×	✓	170	9.05
✓	✓	×	36	1.92
✓	✓	✓	15	0.80

Table VII: Analysis of FOTA apps using Recovery system

apps in the telemetry (§VII-A), and then it presents an analysis of the FOTA apps installation behavior (§VII-B).

A. Detecting FOTA apps in telemetry data

We query the reputation logs dataset (§III) with the FOTA packages discovered by *FOTA Finder*. We collect all records where the APK's parent package name matches the package name of one of the FOTA packages. Searching for FOTA telemetry events using only the FOTA package name can be problematic. Malicious apps can impersonate FOTA apps by using the same package name, misleading us into assigning their malicious installations to benign FOTA packages. We address this issue by checking the certificate information of FOTA apps when this is available. To increase our confidence on our matching, we also search the telemetry for other package names signed by these certificates and confirm that these are used for signing other system apps from the same entity.

The resulting dataset contains 15,961,424 installation events originating from 20 FOTA packages on 841,344 devices from 199 different country codes [18]. Although we are able to detect the presence of 101 (63%) FOTA packages in the telemetry, only 20 (12%) show some installation activity. The reason for the low coverage is the fact that the telemetry uses the *PackageInstaller* for obtaining the installer information, and FOTA apps may install apps via other means (*e.g.*, as an update of the system partition, or through the execution of external programs). Thus, our analysis includes only a subset of all FOTA installations and our measurements should be considered as lower-bound estimations.

B. Telemetry analysis

We next analyze the installation behavior of the 20 FOTA packages that we detect in the telemetry. We are particularly interested in checking if FOTA packages, in addition to system updates, are responsible for non-system app installations. These installations can provide evidence of FOTAs being used for secondary purposes like promoting third-party apps [48]. These promotions can be part of commercial agreements between OEMs and third-party developers (*e.g.*, similar to the one between Facebook and Samsung for pre-installing Facebook on Samsung devices [12]). These installations may happen without user's consent and users may not be able to permanently remove these apps if installed under system partitions. We also investigate if FOTA apps contribute in the distribution of potentially unwanted programs (PUP) and malware.

To assist our analysis, we leverage part of the pipeline in [48]. More specifically, we complement the telemetry dataset with two additional sources. First, we query all installed apps to VirusTotal [65]. We consider as malicious any app that is flagged by at least 4 Anti-Virus (AV) engines, a threshold aligned with prior work [69]. Moreover, when AV labels are available, we feed them to AVClass, a malware labeling tool that outputs their malware families [60]. Second, we check the presence of installed apps on the Google Play Store by searching for their package names.

Table VIII summarizes the behavior of the top 10 FOTA installer packages by number of installation events. For each FOTA installer package, the left part of the table shows the package name and the type of the entity that operates it. The middle part summarizes the installations: number of installation events, devices, and countries. The right part summarizes the installed apps: number of apps, malicious apps, packages, packages found in Google Play Store, and signers. From the 20 FOTA packages detected, 13 are from OEMs, 3 from SoCs and 2 from SFDs. We also observe one Google package (`com.google.android.gms`) and one AOSP package (`com.android.settings`) that provide general services for OEMs. Installations from these two packages have nothing to do with Google, but they rather capture the installation behavior of multiple vendors. We also observe that 3 installer packages, `com.sonyericsson.updatecenter`, `com.samsung.android.app.omcagent`, and `com.coloros.sau`, are responsible for the vast majority of the installation events followed by a long-tail of FOTA packages.

Play Store installations: We look for FOTA packages that may perform installations that are not related with the update process. We use the presence of installed packages in the Google Play marketplace as a signal for this behavior. In total, 13 out of 20 FOTA installers install 764 packages that exist in Google Play Store. These 764 packages belong to 50 Google Play categories; the most common ones are *Tools* (20% of the packages), *Entertainment* (9%), *Communication* (9%), *Games* (8%), and *Shopping* (7%). For example, we observe `com.qualcomm.qti.carrierconfigure`, an SoC FOTA app installing `com.dictionary.paid`, an English dictionary app. In addition to Play Store categories, we leverage the classification from §V-A to detect apps that belong to OEM and MNO companies. We further analyze the installed packages of four FOTA packages, `com.google.android.gms`, `com.andoid.settings`, `com.sonyericsson.updatecenter` and `com.samsung.android.app.omcagent`. We select these four because of their intense installation activity in terms of different packages installed, including packages listed on Play Store market. Three out of four of these FOTA packages install packages from all three categories but predominantly packages belonging to OEMs and MNOs. For example, `com.sonyericsson.updatecenter` installs 260 packages, 48% of which belong to *Sony Electronics Corporation*, 25% to 11 MNOs (*e.g.*, NTT DOCOMO, KDDI, Vodafone), 5% to the Gameloft video game publisher, and the remaining 22% to various software publishers including large retailers (*e.g.*, Amazon and Rakuten), social media (*e.g.*, Facebook and

FOTA Installer		Installations				Children			
Package name	Type	Events	Devices	CC	APKs	Mal. APKs (%)	Pkgs	PlayStore (%)	Sig.
com.sonyericsson.updatecenter	OEM	12.6M	381K	156	1.4K	1 (0.1%)	260	149 (57%)	148
com.samsung.android.app.omcagent	OEM	3.0M	332K	181	1.9K	29 (1.5%)	489	330 (67%)	334
com.coloros.sau	OEM	191K	65K	68	985	28 (3%)	77	1 (1.3%)	35
com.google.android.gms	Google	147K	60K	145	244	4 (1.6%)	29	29 (100%)	28
com.android.settings	Unknown	35K	4.7K	30	1.4K	494 (35%)	851	233 (27%)	582
com.meizu.flyme.update	OEM	6K	773	37	521	1 (0.2%)	61	1 (1.6%)	9
com.qiku.android.ota	OEM	310	77	1	12	11 (92%)	12	1 (8%)	12
com.htc.updater	OEM	302	120	15	6	0	6	3 (50%)	3
com.archos.arum	OEM	156	85	4	4	0	1	0	4
com.qualcomm.qti.carrierconfigure	SOC	119	15	3	7	0	7	4 (57%)	4

Table VIII: Top 10 FOTA installers by number of installation events in the telemetry.

Instagram), and streaming services (*e.g.*, Netflix). Similarly, com.samsung.android.app.omcagent installs 489 packages, 42% of which belong to 32 MNOs (*e.g.*, Vodafone, T-Mobile, Movistar), 13% to Samsung, 6% to video game publishers (*i.e.*, Gameloft and Herocraft), 2% to Amazon, and the remaining 37% to various software publishers including social media (*e.g.*, Facebook), and streaming services (*e.g.*, Chilli, Spectrum).

On the other hand, the package com.google.android.gms installs only mobile device management apps, *i.e.*, apps that enable the administration of corporate devices. It is difficult to know the exact reasons behind these installations, but one explanation can be that vendors use system updates to ship apps included in newer commercial agreements with third-party developers. It is also possible that these apps are shared components of the vendors' app ecosystem used for performing installation initiated by other apps, *e.g.*, vendors' app stores.

Malicious installations: Since FOTA packages appear to install a variety of non-system packages, we are interested in checking if any of those are malware or potentially unwanted programs (PUP). Prior work shows that privileged installers (*i.e.*, installers signed with the platform key) can be an important distribution vector of unwanted apps [48]. We observe 7 out of the 20 FOTA packages installing at least one unwanted app. Five of those packages have a low ratio of unwanted apps installations, varying from 0.1% to 2.8%. We use AVClass to analyze the unwanted apps installed from these 7 FOTA packages, finding that these are identified as *adware*, *smsreg*, or *hiddad*. These are PUPs and their behavior may vary from showing intrusive ads to collecting personal identifiable information [55]. However, the remaining two FOTA packages, com.android.settings and com.qiku.android.ota, have an alarmingly high ratio of unwanted apps installed reaching 35% and 92% respectively. The former installs the largest number of malicious apps and, although most of these are PUPs, we observe at least four different malware families, including instances of the *triada* trojan [37] and the *necro* trojan dropper, a trojan found embedded in popular apps available in Google Play [44]. One likely reason behind the high number of unwanted apps for this package is the fact that it captures the installation behavior of multiple vendors; we identify at least two: *Prestigio* and *EastAeon*. Finally, we observe that com.qiku.android.ota surprisingly installs *only* malware which belongs to different malware families including trojans like *triada*, *necro*, and *guerilla* (a trojan that sends SMS messages without the user's knowledge). These malware installations take place possibly due to compromised third-party vendor code included in the OEM images, like in the case of the *triada* trojan that was found in the devices of several OEMs [23]. Unwanted apps installed under the system partitions cannot be removed neither by users nor by security tools, and they require an OEM update.

VIII. Discussion

The stakeholders involved in the operation of FOTA apps have quite some control over user devices during their entire life span. Due to their privileged position, they also have a responsibility to not introduce unnecessary privacy or security harm. We next provide a discussion on the implications of the findings reported in this paper along various dimensions: security, privacy, and transparency. We are confident that our analysis and findings help clarifying the current FOTA ecosystem and the update supply chain, and can inform the design of future processes in this space, regardless of the adoption of projects Treble and Mainline by vendors.

Security and privacy implications. Google provides recommendations on how to implement FOTA updates [3], [20], but these are superficial and vague on best security practices. In fact, developers might opt not to follow them at all as in the case of some Samsung devices. We have shown that this can result in potentially privacy-intrusive behaviors or the presence of third-party SDKs that can leverage the set of permissions of FOTA apps or the dissemination of personal data to online servers. Furthermore, we have observed that 2% of the FOTAs in our dataset are signed by the default AOSP test keys, as well as 90 devices contains a otacerts.zip file with these default AOSP test keys, allowing in the former any app signed by the same key to run in the same process of the FOTA app (*i.e.*, sharing permissions and capabilities), and in the latter to install any update signed with one of these test keys. Any of these issues can be considered a severe vulnerability which might allow a malicious actor to gain full control of the device. The fact that these vulnerabilities have remained undetected suggests that FOTA apps are subject to little oversight.

Separation of purposes and capabilities. We find that many FOTA apps also include install capabilities for regular, non-system apps. This confirms that a key part of the supply chain, the FOTA process, leverages their full control of users' devices for secondary purposes (*e.g.*, for commercial partnerships as

in PPI-like schemes [41], [48]). Because of the privileges they hold, this can happen without user consent. Even if there are legitimate use cases for installing non-system apps through the FOTA app, this opens the door to the installation of unwanted or potentially harmful apps. Our telemetry analysis provides evidence of a small but significant percentage of unwanted app installations originating from FOTA apps. These are mostly PUPs (*e.g.*, adware), but we also identify FOTA apps installing malware and regular market apps which cannot be removed by users and require an OEM system update. As a security good practice, system and non-system updates should be separated and managed by different processes with different privileges and installation mechanisms.

Transparency. In our analysis, we show that FOTA capabilities can be spread over multiple APK files and even be implemented by multiple stakeholders for different tasks (*e.g.*, to update SoC firmware or MNOs' specific apps). In some cases, OEMs are directly responsible for the updates, while in others they opt to externalize the development of the FOTA app and the management of the updates to an external company, *e.g.*, SFD such as Redstone or Adups. This adds complexity to the supply chain and increases risk, since it requires understanding and controlling a wider spectrum of stakeholders. The more actors in the supply chain, the harder it becomes for users to know who has the ability to update and install apps in their system. This also increases the chances that one of the actors along the supply chain is malicious or implements its own commercial agreements, which might include Pay-Per-Install programs. Furthermore, it is very rare for a FOTA apps to be publicly available or documented, and their privacy policies are hard or impossible to find. This limits the ability of external parties to asses their security and privacy implications. In this regard, our analysis reveals that a substantial amount of FOTA apps use native code and Linux-level commands. Even if there might be valid uses for that, the behaviors located there remain unknown. We leave a thorough analysis of this for future work.

Limitations. While analizing FOTA apps dynamically would provide us with actual evidence of their behavior, there are limitations inherent to pre-installed apps that prevent us from doing this. First, we cannot run apps with a sharedUserId equals to `android.uid.system` (60% of 1,737 FOTAs) in existing sandboxing environments, since these apps must be signed with the same platform key as the device. FOTA apps also have device-specific dependencies that prevent us from running them in any other device. Finally, running the app might require interacting with a production FOTA's server, which we choose to avoid. The telemetry analysis helps us to overcome this limitation, providing us with data obtained from real user devices where FOTA apps are originally installed.

IX. Related work

To the best of our knowledge, this work is the first large scale study to analyze the ecosystem of Android FOTA apps, including its stakeholders, behaviors, and security and privacy risks. Recent studies have investigated similar concerns related to Android pre-installed apps and its supply chain. Our work complements these previous efforts by exploring another integral part of the supply chain: the software update process. The study conducted by Gamba *et al.* [43] provided a first and general overview of such issues, showing how the openness of Android's supply chain, together with the lack of transparency, opens the door for security and privacy issues derived from pre-installed software We complement this study showing that pre-installed FOTA apps allows for a dynamic supply chain, *i.e.*, it does not finish once the device is purchased. Elsabag *et al.* presented a static analysis tool to automatically discover unwanted functionality on Android pre-installed firmware [39]. They applied this method to a dataset of Android firmware images from different vendors and reported 850 vulnerabilities, many of them being zero-day.

Keeping systems updated is a difficult problem and several studies have explored why [58], [63], [64], [66]. Farhang *et al.* performed a study of the ecosystem of Android vulnerability patches [40]. They found that there are different delays between the time when a provider offers a patch and when Android provides the same patch for its OS, the number of patch releases that appears before or in the same month as the public disclosure of the vulnerability (94%), and the maximum time between the first line of code related to a vulnerability and the publication time of a security bulletin (1350 days) [40]. Rula *et al.* showed that up to 25% of web requests were coming from software that was outdated by more than 100 days. Previous work has also looked at the presence of vulnerabilities and malware on custom Android firmwares [34], [68], showing that pre-installed apps found in Android custom ROMs were leaking user data.

X. Conclusions

In this paper, we conducted the first analysis of the Android FOTA ecosystem using a dataset of 2,013 FOTA apps collected in the wild. Our results depict a complex and fragmented ecosystem, with FOTA apps being developed by different first- and third-party actors. This suggests a dynamic supply chain where privileged apps are installed and updated by FOTA apps, together with other pre-installed apps, in the system partition. We demonstrated how this fragmentation can lead to potential privacy-intrusive practices, as well as insecure and potentially harmful behaviors (*e.g.*, FOTA apps signed with well-known certificates). Finally, we showed that FOTA apps can install other non-system apps in user devices, including abusive malware and PUPs. We hope that our results can contribute to inform better designs in this space and increase the security and privacy of the overall Android ecosystem. We have publicly released our tools [11], [14] and the aggregated dataset [5] to foster future research on this topic.

Acknowledgments

The authors would like to thank the anonymous reviewers of this paper and our shepherd, Prof. Xiaojing Liao, for their valuable feedback. This work was partially funded by the US NSF under grant number CNS-1564329; the Spanish grants ODIO (PID2019-111429RB-C21 and PID2019-111429RB-C22); the Region of Madrid grant CYNAMON-CM (P2018/TCS-4566), co-financed by European Structural Funds ESF and FEDER; NortonLifeLock; Google; and Consumer Reports. Author Narseo Vallina-Rodriguez is a Co-founder of AppCensus Inc.

REFERENCES

[1] A/B System Update update_engine android daemon code. https://cs.android.com/android/platform/superproject/+/master:system/update_engine/. [Online; accessed 26-Mar-2020].

[2] A/B System Update UpdateEngine.java client class. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/os/UpdateEngine.java. [Online; accessed 26-Mar-2020].

[3] About A/B system updates. https://source.android.com/devices/tech/ota/ab#overview. [Online; accessed 25-November-2020].

[4] Android Compatibility Program Overview. https://source.android.com/compatibility/overview. [Online; accessed 31-March-2019].

[5] Android Observatory — Datasets. https://androidobservatory.com/datasets/. [Online; accessed 2021-03-17].

[6] Android old update client class. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/os/RecoverySystem.java. [Online; accessed 26-Mar-2020].

[7] Android recovery code. https://cs.android.com/android/platform/superproject/+/master:bootable/recovery/recovery.cpp;l=1?q=Recovery.cpp. [Online; accessed 26-Mar-2020].

[8] Android recovery code installer, VerifyAndInstallpackage method. https://cs.android.com/android/platform/superproject/+/master:bootable/recovery/install/install.cpp;l=530. [Online; accessed 30-Nov-2020].

[9] Android recovery package template. https://android.googlesource.com/platform/bootable/recovery/+/93ffa75/updater/. [Online; accessed 26-Mar-2020].

[10] Androwarn–Yet another static code analyzer for malicious Android applications. https://github.com/maaaaz/androwarn. [Online; accessed 31-March-2019].

[11] Dextripador. https://github.com/Android-Observatory/Dextripador. [Online; accessed 2021-03-17].

[12] Facebook deal makes it impossible to delete app from Android smartphones. https://www.independent.co.uk/life-style/gadgets-and-tech/news/facebook-app-delete-android-smartphones-samsung-galaxy-a8719081.html. [Online; accessed 2020-11-28].

[13] Firmware Scanner. https://play.google.com/store/apps/details?id=org.imdea.networks.iag.preinstalleduploader. [Online; accessed 31-March-2019].

[14] FotaFinder. https://github.com/Android-Observatory/FotaFinder. [Online; accessed 2021-03-17].

[15] Fresher OS with Projects Treble and Mainline. https://android-developers.googleblog.com/2019/05/fresher-os-with-projects-treble-and-mainline.html. [Online; accessed 28-Jun-2020].

[16] 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. [Online; accessed 31-March-2019].

[17] Here comes Treble: A modular base for Android. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html. [Online; accessed 28-Jun-2020].

[18] Iso 3166-1 alpha-2. https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2.

[19] 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. [Online; accessed 31-March-2019].

[20] Life of an OTA update. https://source.android.com/devices/tech/ota/nonab#life-ota-update. [Online; accessed 25-November-2020].

[21] Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share. [Online; accessed 22-Jun-2020].

[22] Permissions overview. https://developer.android.com/guide/topics/permissions/overview.html. [Online; accessed 31-March-2019].

[23] PHA Family Highlights: Triada. https://security.googleblog.com/2019/06/pha-family-highlights-triada.html. [Online; accessed 2020-11-30].

[24] Privileged Permissions Whitelisting. https://source.android.com/devices/tech/config/perms-whitelist. [Online; accessed 17-September-2020].

[25] There are now 2.5 billion active Android devices. https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote. [Online; accessed 22-Jun-2020].

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

[27] Android Developers. RequiresPermission AndroidX. https://developer.android.com/reference/androidx/annotation/RequiresPermission. (accessed 2020-03-23).

[28] Android Developers. Volley overview. https://developer.android.com/training/volley. (accessed 2020-10-28).

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

[30] Android Source. Signing Builds for Release. https://source.android.com/devices/tech/ota/sign_builds. (accessed 2020-10-28).

[31] 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. *Proceedings of the ACM Special Interest Group on Programming Languages (SIGPLAN)*, 2014.

[32] 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 *25th USENIX Security Symposium (USENIX Security 16)*, pages 1101–1118, Austin, TX, August 2016. USENIX Association.

[33] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla. What did really change with the new release of the app? In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 142–152, 2018.

[34] N. T. Cam, V. Pham, and T. Nguyen. Sensitive data leakage detection in pre-installed applications of custom android firmware. In *2017 18th IEEE International Conference on Mobile Data Management (MDM)*, pages 340–343, 2017.

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

[36] Anthony Desnos, Geoffroy Gueguen, and Sebastian Bachmann. Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!). https://github.com/androguard/androguard, 2012. [Online; accessed 17-Jan-2020].

[37] Dr Web. Trojan preinstalled on Android devices infects applications' processes and downloads malicious modules. http://news.drweb.com/news/?i=11390&lng=en. [Online; accessed 31-March-2019].

[38] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, San Francisco, CA, USA, 1st edition, 2014.

[39] 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 *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[40] Sadegh Farhang, Mehmet Kirdan, Aron Laszka, and Jens Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities, 05 2019.

[41] Shehroze Farooqi, Álvaro Feal, Tobias Lauinger, Damon McCoy, Zubair Shafiq, and Narseo Vallina-Rodriguez. Understanding incentivized mobile app installs on google play store. In *Proceedings of the ACM Internet Measurement Conference*, pages 696–709, 2020.

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

[43] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 197–213.

[44] Sergiu Gatlan. Trojan Dropper Malware Found in Android App With 100M Downloads, August 2019. https://www.bleepingcomputer.com/news/security/trojan-dropper-malware-found-in-android-app-with-100m-downloads/.

[45] Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, 1961.

[46] Google Firebase. Privacy and Security in Firebase. https://firebase.google.com/support/privacy. (accessed 2020-10-28).

[47] 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. [Online; accessed 31-March-2019].
[48] Platon Kotzias, Juan Caballero, and Leyla Bilge. How did that get in my phone? unwanted app distribution on android devices, 2020.
[49] Leak Canary. Leak Canary: A memory leak detection library for Android. https://github.com/square/leakcanary. (accessed 2020-10-28).
[50] Jonathan Levin. *Android Internals::A Confectioner's Cookbook. Volume I: The Power User's View*. Technologeeks.com, 1st edition, 2015.
[51] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 653–656, 2016.
[52] 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 Companion*, 2016.
[53] Mehran Mahmoudi and Sarah Nadi. The android update problem: An empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 220–230, New York, NY, USA, 2018. Association for Computing Machinery.
[54] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.
[55] Mihai Grigorescu. Hiddad Android malware gets top user ratings for all the wrong reasons. https://www.avira.com/en/blog/top-rated-android-malware. (accessed 2020-03-27).
[56] Google Play. Developer Program Policy, October 2020. https://support.google.com/googleplay/android-developer/answer/10177647?hl=en&visit_id=637408674530764193-63172764&rd=1.
[57] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
[58] John P. Rula, Philipp Richter, Georgios Smaragdakis, and Arthur Berger. *Who's Left behind? Measuring Adoption of Application Updates at Scale*, page 710–723. Association for Computing Machinery, New York, NY, USA, 2020.
[59] Paul Sabanal. Hiding Behind ART. https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf. [Online; accessed 30-Jun-2020].
[60] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, 2016.
[61] Silvia Sebastian and Juan Caballero. Towards attribution in mobile markets:identifying developer account polymorphism. In *Proceedings of the 27th ACM Conference on Computer and Communications Security*. ACM, 2020.
[62] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and C Gunter. Resolving the predicament of android custom permissions. 2018.
[63] Kami Vaniea and Yasmeen Rashidi. Tales of software updates: The process of updating software. pages 3215–3226, 05 2016.
[64] Kami E. Vaniea, Emilee Rader, and Rick Wash. Betrayed by updates: How negative experiences affect future security. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 2671–2674, New York, NY, USA, 2014. Association for Computing Machinery.
[65] VirusTotal. http://www.virustotal.com/.
[66] Francesco Vitale, Joanna McGrenere, Aurélien Tabard, Michel Beaudouin-Lafon, and Wendy Mackay. High costs and small benefits: A field study of how users experience operating system upgrades. pages 4242–4253, 05 2017.
[67] 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.
[68] Min Zheng, Mingshen Sun, and John CS Lui. Droidray: a security evaluation system for customized android firmwares. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 471–482, 2014.
[69] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. In *USENIX Security Symposium*, 2020.
[70] Łukasz Siewierski. 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. [Online; accessed 2021-03-18].

APPENDIX A
PROJECT MAINLINE DEVICES

As of December 2020, the list of devices in the Project Mainline beta program includes the Google's devices *Pixel*, *Pixel2* and *Pixel3/3a* and 15 devices from other OEMs: *Huawei Mate 20 Pro, LG G8, Sony Xperia XZ3, OPPO Reno, Vivo X27, Vivo NEX S, Vivo NEX A, OnePlus 6T, Xiaomi Mi Mix 3 5G, Xiaomi Mi 9, Realme 3 Pro, Asus Zenfone 5z, Nokia 8.1, Tecno Spark 3 Pro* and *Essential PH-1*.

APPENDIX B
SIGNALS USED IN FOTA FINDER

Table IX describes the set of signals used in *FOTA Finder* to identify apps with FOTA or OTA capabilities.

APPENDIX C
FOTA ECOSYSTEM RELATIONSHIPS

Figure 3 showcases the complex supply chain and relationships that exist in Android's FOTA ecosystem across vendors. We only depict those package names, vendors, and certificates that do not have a 1-to-1 mapping to other elements. The figure illustrate the presence of the same FOTA package across different vendors and brands. This is particularly clear for SFDs such as Adups and Redstone. The mappings also illustrate the use of different certificates and how it adds complexity to the attribution problem.

APPENDIX D
FIREBASE TERMS OF SERVICE

Taken from Firebase's terms of services in November 2020:

Firebase support for GDPR and CCPA

On May 25th, 2018, the EU General Data Protection Regulation (GDPR) replaced the 1995 EU Data Protection Directive. On January 1, 2020, the California Consumer Privacy Act (CCPA) took effect. Google is committed to helping our customers succeed under these privacy regulations, whether they are large software companies or independent developers.

The GDPR imposes obligations on data controllers and data processors, and the CCPA imposes obligations on businesses and their service providers. Firebase customers typically act as the "data controller" (GDPR) or "business" (CCPA) for any personal data or information about their end-users they provide to Google in connection with their use of Firebase, and Google generally operates as a "data processor" (GDPR) or "service provider" (CCPA).

This means that data is under the customer's control. Customers are responsible for obligations like fulfilling an individual's rights with respect to their personal data or information.

Signal	Strength	F/OTA	Silent install	Description	Purpose
RV	Strong	FOTA	N/A	Call to the method `verifyPackage` from the API class `android.os.RecoverySystem`	Signature checking of the downloaded packages to be installed
RI	Strong	FOTA	No	Call to the method `installPackage` from the API class `android.os.RecoverySystem`	Install downloaded packaged by rebooting in recovery mode
CMD	Strong	FOTA	No	Use of the strings *"/cache/recovery/command"* and *"–update-package"* in the code	Alternative to the *android.os.RecoverySystem* API to install system packages. This is always followed by a reboot in recovery mode
A/B	Strong	FOTA	No	Call to the method `applyPayload` from the API class `android.os.UpdateEngine`	Install downloaded packages through the A/B method
OC	Weak	FOTA	N/A	Use of the string *"otacerts.zip"* in the code	The *"otacerts.zip"* file is the default container for the certificates used to verify the packages to be installed.
OU	Weak	FOTA	N/A	Use of the string *"ota_update.zip"*	This is the default, suggested name given to the downloaded file containing all the packages to be installed
PM_I	Strong	OTA	Yes	Call to the method `installPackage` from the API class `android.content.pm.PackageManager`	Method used to install non-system packages
PM_D	Strong	OTA	Yes	Call to the method `deletePackage` from the API class `anroid.content.pm.PackageManager`	Method used to delete non-system packages
PMI	Strong	OTA	Yes	Use of the string `"pm install"` in the code	Alternative method to install non-system packages using the command line.
VND	Strong	OTA	No	Use of the MIME type `application/vnd.android.package-archive` in an intent	Method used in installation intents directed to the PackageManager when requesting it to install a package.
GRANT	Weak	OTA	Yes	Call to the method `grantRuntimePermission` from the API class `android.content.pm.PackageManager`	Method used to grant install permissions at runtime without requiring user approval
REVOKE	Weak	OTA	Yes	Call to the method `revokeRuntimePermission` from API class `android.content.pm.PackageManager`	Used to revoke install permissions to other packages in runtime without requiring user approval
PermI	Strong	OTA	N/A	Request of the AOSP permission `android.permission.INSTALL_PACKAGES`	Permission needed to install non-system packages
PermD	Strong	OTA	N/A	Request of the AOSP permission `android.permission.DELETE_PACKAGES`	Permission needed to remove non-system packages
NAME	Weak	F/OTA	N/A	Use of the strings *"ota"*, *"update"*, *"upgrade"* or *"install"* as part of the package or the APK name)	Common practice observed in many F/OTA apps

Table IX: Features used to automatically discover F/OTA apps.

APPENDIX E
ANDROID DEVELOPER POLICY

The Developer Program Policy explicitly says: *"We don't allow apps that let users install other apps to their devices."*. In addition, the policy section regarding device and network abuse lists this as an example of abusive behavior *"apps that install other apps on a device without the user's prior consent"*.

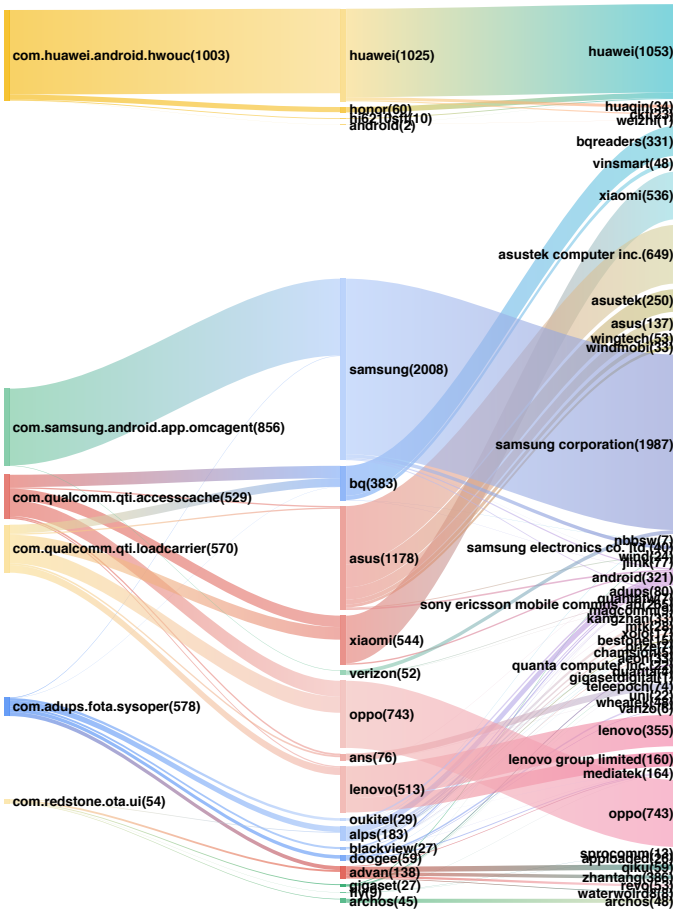

Figure 3: Relationship among FOTA package names, vendor and the subject organization from the app certificate.