2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft) | 979-8-3503-1182-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/MOBILSoft59058.2023.00009

FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps

Thomas Sutter
Institute of Applied Information Technology
Zurich University of Applied Sciences
Winterthur, Switzerland

Dr. Bernhard Tellenbach
Cyber-Defence Campus
Armasuisse
Zurich, Switzerland

***Abstract*—Supply chain attacks are an evolving threat to the IoT and mobile landscape. Recent malware findings have shown that even sizeable mobile phone vendors cannot defend their operating systems fully against pre-installed malware. Detecting and mitigating malware and software vulnerabilities on Android firmware is a challenging task requiring expertise in Android internals, such as customised firmware formats. Moreover, as users cannot choose what software is pre-installed on their devices, there is a fundamental lack of transparency and control. To make Android firmware analysis more accessible and regain some transparency, we present FirmwareDroid, a novel open-source security framework for Android firmware analysis that automates the extraction and analysis of pre-installed software.**

FirmwareDroid streamlines the process of software extraction from Android firmware for static security and privacy assessments. With FirmwareDroid, we lay the groundwork for researchers to automate the security assessment of Android firmware at scale, and we demonstrated the capabilities of FirmwareDroid by analysing 5,728 Android firmware samples from various vendors. We analysed 75,141 unique pre-installed Android applications to study how common advertising tracker libraries (a piece of software that collects user usage data) are used and which permissions pre-installed Android apps inherit. We conclude that 20.53% of all apps in our dataset include advertising trackers and that 88.14% of all used permissions are signature-based.

***Index Terms*—Android Firmware, Pre-Installed Apps, Static Analysis, Security, Vulnerability**

I. Introduction

When we buy a smartphone or IoT device, we buy a piece of hardware and the pre-installed software on the hardware. If the device is part of the Android ecosystem, this software is called firmware, or Android firmware (ROM). If the device manufacturer provides it for reloading onto the device, it is usually in the form of a so-called firmware image. Android firmware consists of the Android operating system with its system apps and other applications developed by the device manufacturer or third parties (e.g., mobile providers or social media companies). The Android Open Source Project (AOSP) provides the basis for the Android operating system, which device manufacturers need to supplement with drivers for the hardware used (e.g., chipset, camera, and various sensors). What is also missing from AOSP are the Huawei or Google Mobile Services (HMS / GMS), a set of applications, APIs, and cloud-based services usually present on Android devices. As these services have to be licensed and pre-installed separately, they are not present on all smartphones by default.

All apps mentioned so far and thus already installed at delivery are commonly called pre-installed apps. However, just like regular apps downloaded and installed by users after the device is shipped, these pre-installed apps also potentially pose a security risk. On the one hand, they can contain vulnerabilities that attackers can exploit. On the other hand, they could have been intentionally equipped with functionality to spy on users or harm them in other ways. This is even more of a problem because pre-installed apps often have higher permissions than regular apps, cannot be easily removed by users, and, in contrast to apps from app stores, there are also no mechanisms like commenting, rating, and reporting potentially problematic apps.

Various well-documented incidents show that both security risks are not only theoretical. The risk of malware introduced by members of the supply chain is demonstrated, for example, by the malware families known as Chamois [1], [2], [3] and Triada [4], [5]. According to these sources, these have infected several million devices and remained undetected for months. In addition to these security risks, privacy risks from advertising trackers contained in the pre-installed apps are moving into the spotlight of the media [6]. Unfortunately, there are hardly any studies on how prevalent such advertising trackers are used in pre-installed apps nowadays.

Indications that this could be a problem are provided by the study of Gamba et al. [7]. It shows that a worrying number of companies known for their aggressive advertising strategies are associated with providers of Android firmware components. However, whether this complies with privacy regulations such as the GDPR or the CCPA remains unclear in the study. On the other hand, some providers of alternative Android firmware, such as LineageOS [8] and GrapheneOS [9], see in the commitment to a privacy-friendly Android firmware and the renunciation of so-called bloatware at least a market gap and opportunity. If we look at all of these risks and take the many incidents, some of them well documented, as a reference, we have to conclude that even manufacturers like Google, which have sophisticated security measures and processes in place, are not immune to them. Moreover, it raises the question of how less security-conscious device manufacturers can detect and respond to attacks via the supply chain and how secure pre-installed apps are in general.

Certification of manufacturers or of devices could eventually be part of the answer. For example, users of Android firmware that comes with Google Mobile Services can be sure that their firmware had to go through the Google Play Protect[1]

[1] https://www.android.com/certified/

DOI 10.1109/MOBILSoft59058.2023.00009

certification process. However, the exact testing methodology and the technical details of the test are not known to the public. The same applies to a device that has been certified according to the Android profile of the ioXt Alliance[2]. In addition to the requirement that it must already be certified for GMS, the profile defines various non-technical criteria such as the existence of a vulnerability reporting program and some criteria such as the risk posed by the pre-installed apps (”very low”, ”low”, or ”medium”), for which it is not clear where the relevant data should come from. In an assessment, the NCC Group used the Uraniborg tool [10], [11] for this purpose, for example.

Hence, while certification might shed some light on what has been tested and how, we ultimately still know little about manufacturers' security measures and internal testing procedures. At the same time, many tools exist, including freely available ones, for analysing individual security aspects of Android apps (e.g., [12], [13], [14], [15], [16], [17], [18]), but none of these can directly cope with Android firmware images, let alone the entire Android firmware ecosystem.

Consequently, without proper tools for verification and large-scale independent investigations into security aspects of the Android firmware ecosystem, users have no choice but to trust in good faith that device manufacturers are trustworthy and act in the customer's best interest. In addition, customers have to trust that manufacturers have proper procedures and analysis methods in place to minimise security and privacy risks in the supply chain. In this paper, we aim to contribute to making the security of the Android firmware ecosystem more transparent. We provide two contributions to this end. The first contribution is the FirmwareDroid framework, published as an open-source project, enabling the research community to conduct large-scale, independent investigations into the security aspects of the Android firmware ecosystem much faster and cheaper. FirmwareDroid achieves this primarily by solving various challenges in automating the various steps such as collecting and unpacking firmware samples or distributing the analysis across various analysis tools integrated and containerised in FirmwareDroid. Section III gives an overview of FirmwareDroid and describes the different steps FirmwareDroid performs in an automated way to collect and analyse Android firmware. We also outline some of our approaches to solving problems that we needed to overcome to turn FirmwareDroid into a solution that can analyse the Android ecosystem faster and with less manual work. We demonstrate that our framework is capable of managing a wide variety of independently developed tools for forensics, security, and data analysis.

Our second contribution is to publish the results of our analysis of the Android firmware ecosystem using FirmwareDroid concerning the following research questions:

- $\boldsymbol{RQ_1}$: Which permissions are used by pre-installed apps? More specifically, we investigate (i) how common pre-installed apps use dangerous and other permissions categories and (ii) whether there are significant differences in permission usage between the vendors in our dataset.
- $\boldsymbol{RQ_2}$: How common are advertising tracker libraries used in pre-installed apps? Here, we want to know (i) which advertising tracker libraries we can detect and (ii) how often these libraries are used within pre-installed apps.

For the second contribution, we analysed 75,141 unique pre-installed apps from 5,728 Android firmware samples. We found that (1) the average number of dangerous permissions used by the apps has decreased from 10.2% to 3.6% from Android version 10 to version 11, at least on Google firmware, and (2) there are over 40,000 advertising trackers and 20.53% of all apps use at least one advertising tracker, and (3) while there are 3.56% apps that use dangerous permission a more significant fraction, 88.14%, of pre-installed apps use signature permissions.

The rest of the paper is structured as follows. Details on the data sources used to collect the Android firmware samples and some key figures of the resulting dataset are presented in Sections III and IV. The analysis results related to RQ_1 can be found in Section V and those related to RQ_2 in Section VI. If you want to use our dataset or the code for your own research, please see Section X for further details.

II. Related Work

Gamba et al. [7] analysed pre-installed apps from more than 200 vendors and showed privacy concerning relationships between third-party app developers, vendors, and device manufacturers by using LibRadar [19] to detect known advertising libraries. Their analysis focused on 82,501 pre-installed android applications collected from 2,748 devices with 1,795 unique package names. It is unknown how many of the apps were unique and thus their dataset may include some duplicates. We assume that Gamba et al. [7] could give us only some limited insights into how common advertising companies are represented in pre-installed apps, but their insights show already concerning relations between advertising companies and mobile phone vendors.

Using FirmwareDroid, we overcome scalability limitations and demonstrate that static analysis of several hundred thousand apps is possible within a reasonable amount of time and computation power. Regarding RQ_2, we demonstrate that we can detect over 40,000 instances of tracker libraries by using Exodus [20] in pre-installed apps, which is so far lacking in the scientific literature. Exodus Privacy is a non-profit organisation that provides a static analysis tool for detecting advertising trackers on Android. Exodus can detect 405 known advertising tracker libraries at the time of writing, and the Exodus Privacy team continuously detects trackers of apps in the Google Play Store and publish their results on their official website[3].

Another more recent work by Kollnig et al. demonstrated in [21] that Android apps from the Google Play Store often violate European privacy regulations regarding consensual advertising tracking. Their results show that many Android apps

[2] https://www.ioxtalliance.org/

[3] https://exodus-privacy.eu.org/en/

do not ask for the user's consent to be tracked by advertising frameworks and that many app developers violate the GDPR. Our work contributes to this research as we can provide the names and exact numbers of apps that use advertising trackers, which allows testing of apps at a finer granularity that may not comply to such regulations as the GDPR. Moreover, we provide the dataset together with our codebase to allow other researchers to reproduce or enhance our results. With FirmwareDroid we strive to significantly lower the burden for other researchers to analyse pre-installed Android software.

Other researchers have focused on Android's Mandatory Access Control (MAC) and Discretionary Access Control (DAC) policy model [22], [23], [24]. BigMac by Hernandez et al. [24] has shown discrepancies between MAC and DAC permissions that lead to the fact that untrusted apps could access root processes over inter-process communication (IPC) and could load kernel modules. Aafer et al. [25] developed several differential analysis algorithms to detect inconsistent security configurations between different builds of the same firmware. They tested their approach on 591 custom firmware samples and found in several cases that newer OS releases downgraded some of the security settings, which lead to potential security risks if done unintentionally.

The development of tools like DiffDroid [25], BigMac [24], and PolyScope [22] demonstrate that analysing Android firmware often requires researchers to implement common tasks such as extracting files from Android firmware from scratch. With the FirmwareDroid framework, we aim to provide the groundwork for such common tasks and reduce researchers' time to implement such components and focus more on their research. Moreover, by selecting two common static analysis tools, we demonstrate that using a common dataset is beneficial for future research, as it makes benchmarking and comparison of the tools and results possible. Therefore, one of our main contributions is to provide a framework that allows to extract and analyse pre-installed Android software from the firmware without the need of the hardware.

Other research has focused on developing a rating system for Android firmware, for example, a white paper by Lau et al. [10] proposed a scoring system for Android firmware, called Uraniborg, based on the permissions given to pre-installed apps. Their idea was to create an easily understandable risk metric for Android firmware based on scoring the risk implied by specific permissions and on the number of apps that allow clear-text traffic. The NCC group used this risk metric to evaluate Google Pixel devices for an ioXt Audit [11]. Cam et al. [26] described a system, uitXROM, to analyse the relationships of pre-installed apps for sensitive data leakages, which could help in detecting privacy concerning apps. As we have automated the process of permission extraction within FirmwareDroid, it is an ideal framework for using it as well to rate the security majority of an Android firmware and use rating systems such as the one proposed by the Uraniborg team.

By integrating the tool from Desnos and Gueguen [27] into FirmwareDroid, we provide an easy way for users to extract the used permissions of pre-installed apps, and we demonstrate in this paper how we can extract the used number of permissions for every permission level for further evaluations. As an all-rounder tool (decompiler, disassembler, XML parser, call-graph extraction), AndroGuard [27] became sort of a standard tool for the static analysis of Android applications. Other studies [7], [17] used AndroGuard for extracting app meta-data like used permissions, certificates, string-analysis, and defined components. Some studies even used AndroGuard for malware analysis [28] or as the basis for their tools: BLECryptracer [29], and HSandroguard [30]. Within FirmwareDroid, we use AndroGuard mainly to automate the extraction of permissions, certificates, and strings.

Previous research can only be considered the first step towards a more profound understanding of how vendors customise Android's permission system in practice and what permissions pre-installed app developers use. We, therefore, address in this paper how common pre-installed apps use dangerous and custom permissions and provide a framework that can automate the analysis for pre-installed apps. We accomplish this by collecting 5,728 firmware samples from freely available sources.

Another key aspect of pre-installed app research focuses on vulnerability and malware detection. For example, the team around FirmScope [31] has developed a novel static taint analysis tool and found 3,483 privilege-escalation vulnerabilities in pre-installed apps by examining 2,017 Android firmware images. In [32] You et al. analysed eleven Android smart TV boxes with fuzzing techniques which resulted in the finding of 37 unique vulnerabilities, and in [33] Zheng et al. found a specific privilege escalation vulnerability in 99.6% (249 samples) of their firmware dataset. In addition, Hou et al. [34] showed on a dataset of 6,261 firmware images that patch delays are widely spread among Android images and that pre-installed apps often contain publicly known vulnerabilities. While many tools find vulnerabilities, there are hardly any studies that analyse their findings in terms of "can this be exploitable and is it relevant", or if so, then very limited in scope or to particular vulnerabilities. The lack of a dataset with ground truth is undoubtedly a problem, which will probably never be solved at large scale since this currently still requires manual verification. Thus, we contribute to this area of cyber security research by publishing our dataset and providing researchers with a tool that allows them to combine the results of several static analysis tools.

III. FirmwareDroid

In this section, we provide an overview of the FirmwareDroid framework. We briefly describe the different steps it performs to collect and analyse Android firmware and outline some of the challenges we had to overcome to turn it into a solution with which the Android ecosystem can be analysed quicker and with less manual work. Next, we discuss our choice of data sources for Android firmware images and provide metrics for the set of Android firmware collected by FirmwareDroid's crawler component.

A. Data Acquisition

To collect firmware samples, we developed a web crawler and downloaded firmware samples from official Android firmware vendor websites that publish their firmware for free and with cryptographic checksums. We used Puppeteer [35] to control a headless chromium-browser and implemented custom routines to download the firmware samples. Our crawler collects all links from a predefined list of websites by recursively going through the website's sitemap and searching for links that refer to compressed (.zip, .tar) files that we later download. After collecting the download links, we removed unwanted files (such as drivers and wallpapers) and downloaded the remaining firmware files. During import, FirmwareDroid would automatically detect if a file is a valid firmware and delete invalid files.

We selected websites based on their popularity, the number of samples, and their sharing policy. Moreover, for our research we used only firmware from official Android vendors to have precise measurements. Unfortunately, many of the market leading Android firmware vendors do not provide their firmware publicly on official channels.

Nevertheless, with our data collection method, we were able to collect 6,169 firmware samples from seven operating system vendors (OSV) and FirmwareDroid was able to unpack 5,728 of these samples. Table I shows the number of firmware samples collected for every OSV and Android version[4]. Our dataset contains some of the most popular custom ROMs, LineageOS [8], GrapheneOS [9], Paranoid [36], OmniROM [37], and Resurrection Remix OS (RROS) [38] as well as all available smartphone firmware from Google [39]. All the firmware was collected in July 2021.

As another data source we could have used websites that claim to provide official Android firmware but do not have an official affiliation with the vendor. In a previous study [40] we tested a set of such websites and came to the conclusion that using such data sources inherit a fundamental problem: Firmware from an unknown source could have been modified and as many firmware vendors do not publish their cryptographic checksums, we cannot verify if the firmware is from the original vendor or a modified version. Thus, in this study we use only official and verifiable firmware.

As an alternative data source, we could have collected our pre-installed apps with a crowd-sourced approach as for example Gamba et al. [7] did. However, this approach does not allow the extraction of large parts of the firmware, such as the kernel or the baseband operating system, and is limited through the number of users that are willing to share their pre-installed apps. With FirmwareDroid we provide a framework which is not only able to analyse pre-installed Android apps, moreover it provides the possibility to analyse the dependencies such as native libraries of the operating system or other system components.

Another fact that we would like to express is that no other research team did provide their firmware nor their data set of pre-installed apps so far, which makes it impossible to reproduce their results or use any of their data sources for our experiments. We overcome this gap in the literature by providing the full data set with 5,728 firmware samples and 850,555 pre-installed apps.

B. Design and Implementation

We had to solve several challenges to automate the process of static analysis for pre-installed apps or any other file within the firmware. This section explains our solution design for the FirmwareDroid framework and gives some details about the challenges to overcome. We start by giving an overview of FirmwareDroid in Figure 1 and explain step by step how we can automate the scanning process.

(1) **Firmware pre-processing.** Android operating system and hardware manufacturers use various formats and customize their firmware according to their needs. Different compression algorithms are used to store the firmware. To access the data within the firmware, we have to detect which compression formats are used and then decompress these formats in the correct order. For example, an Android 11 firmware can be first Brotli [41] compressed, then zipped, and then '.dat' formatted. To get access to the files within the .dat formatted image, we first need to unzip the archive, decompress the resulting Brotli files, and then extract the '.dat' files. The current version of FirmwareDroid can process eight different compression formats, including Brotli and nb0, in various combinations by using a recursion algorithm. With the current implementation, we extracted more than 8,000 firmware samples (counting as well samples from our previous work [40]) and support a wide variety of compression combinations. From the total of 6,169 downloaded firmware samples for this study, we were not able to import 441 samples (297 OmniROM, 92 RROS, 46 Carbon, 6 Google) due to currently unknown file formats or naming patterns. We hope to overcome this limitation in future versions of FirmwareDroid by including additional unpacking tools, or compression formats.

(2) **File extraction.** Android firmware partitions are Sparse or Yaffs2 formatted [42], [43]. After decompressing the firmware, we need to locate the files of interest that store the system, vendor, OEM, or other data partitions. Since manufacturers have no strict naming convention for partition files, locating the correct files can be challenging and is an error-prone task for automation. However, our approach overcome these shortcomings by using a rule system based on regular expressions (regex). We analysed manually the names of several hundred data partitions and created regex patterns for matching filenames (e.g., *'user.img'*, or *'kernel.img'*). Depending on the image file format, we developed different routines to extract pre-installed apps from the Android firmware. In general, whenever possible, we attempt to convert Sparse image files with simg2img [44] to an ext formatted image that we can directly mount into the host filesystem. We experienced that using simg2img alone is not sufficient due to the fact that converting to a valid ext image with simg2img would often fail because the host system would prevent us from mounting

[4] We denote 'v0' as unknown Android version.

TABLE I
ANDROID FIRMWARE DATASETS WITH NUMBER OF SAMPLES PER ANDROID VERSION.

ROM Vendor	# Firmware	# Apps	# Unique Packages	# Unique Apps (SHA-256)	v0	v4	v5	v6	v7	v8	v9	v10	v11
Google	1,023	169,392 (19.92%)	652 (0.38%)	37,255 (21.99%)	530	0	0	0	34	68	102	156	133
GrapheneOS	18	3,597 (0.42%)	243 (6.67%)	2,131 (59.24%)	0	0	0	0	0	0	0	0	18
Paranoid	99	19,332 (2.27%)	322 (1.67%)	1,616 (8.36%)	0	0	0	0	0	0	0	99	0
Carbon	1,300	68,901 (8.10%)	317 (0.46%)	2,725 (3.95%)	0	0	0	0	0	94	386	820	0
LineageOS	644	122,838 (14.44%)	457 (0.37%)	4,620 (3.76%)	0	0	0	0	0	0	0	200	444
OmniROM	2,107	292,879 (34.43%)	503 (0.17%)	15,815 (5.40%)	932	51	6	9	252	3	124	530	200
RROS	537	173,616 (20.41%)	688 (0.4%)	10,979 (6.32%)	0	0	0	0	0	0	0	537	0
Total	5,728	850,555	3,182 (0.37%)	75,141 (8.83%)	1,462	51	6	9	286	165	612	2,342	795
					25.52%	0.89%	0.10%	0.16%	4.99%	2.88%	10.68%	40.89%	13.88%

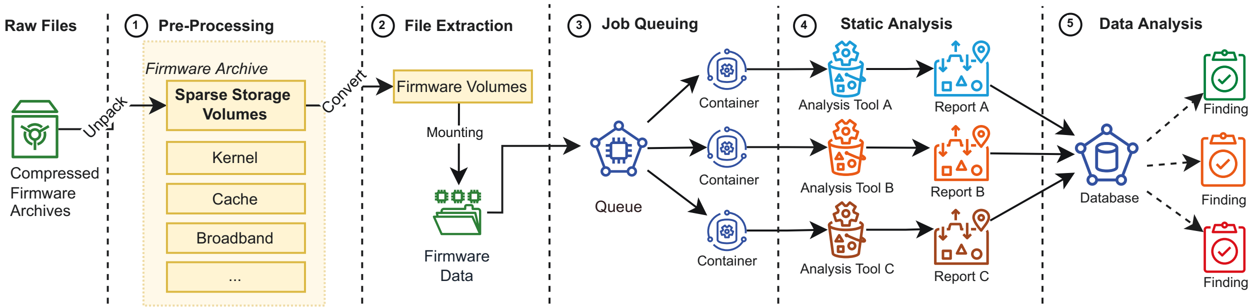

Fig. 1. Overview of the FirmwareDroid framework

customized images successfully. We, therefore, use the kernel module *fuseext2* [45] allowing us to mount modified images. The usage of simg2img in combination with fuseext2 is a novel approach in extracting the data from firmware images as it allows us to extract the data from nearly all firmware vendors. In cases when even the combination of simg2img and fuseext2 is not able to mount an image, FirmwareDroid attempts to extract the firmware with the *imgpatchtools* [46] or the *ubi-extract* [47] tools. Both tools allow to extract the image content directly to the host file system without mounting it first. As soon as the firmware is mounted or extracted, we copy relevant firmware files to a permanent storage on the host system. By default, FirmwareDroid searches for apk-, vdex-, odex-, dex-, and art- files and copies them to a permanent storage. After copying all the relevant files from the firmware, we unmount the firmware and store the firmware images for later use within a permanent storage.

③ **Job queuing.** FirmwareDroid is based on docker and docker-compose and has an integrated queuing system for job management. The complete framework runs within docker, which allows us to separate the environments and scale the number of containers as needed. By default, every static analysis tool has its queue for scheduling jobs. This architecture allows to scale the number of instances for any analysis tool as needed. Moreover, using our queuing system we were able to scan an average of $\approx 55,000$ apps per day on one virtual server with 32 cores (Intel, Broadwell, no TSX, IBRS) and 128 GB RAM. As FirmwareDroid is completely dockerized it is possible to scale the performance by using cluster techniques or several instances of the framework.

④ **Static analysis.** We selected a base set of eight existing open-source static analysis tools (AndroGuard [27], Androwarn [13], SUPER Android Analyzer (SUPER) [14], Quick Android Review Kit (QARK) [15], QuarkEngine [16], APKiD [48], APKLeaks [18], Exodus [20]). We included them into FirmwareDroid with the idea in mind to have an extendable framework that can handle the different requirements of any static analysis tool. It is challenging to unite these tools based on different programming languages, and with individual software requirements and dependencies, within one framework. In addition, these tools often have outdated, conflicting, or undocumented dependencies that make an integration complex. We, therefore, created for every tool a docker container and developed an API to access the generated scanning reports. Another advantage of this architecture is that we are able to modify or remove tools individually, which makes FirmwareDroid scaleable for larger environments and reduces the maintenance workload significantly. In this study we will focus only on the result provided by AndroGuard and Exodus to answer our research questions.

⑤ **Data analysis.** The static analysis tools in our corpus use JSON formatted output files for their result reports or other text-based output formats. A concern we had at the beginning when we were integrating these reports was the long-term maintainability of FirmwareDroid. Over time, the reports generated by the static analysis tools are likely to change, resulting in breaking changes for FirmwareDroid and generating large and continued workloads to keep up with the latest changes. To cope with changing output formats, we decided to use a dynamic schema based on a NoSQL database that allows us to store the generated data by the static analysis tools without defining a strict report schema. We think this should allow us to keep up with the latest changes in the static analysis tools with a minimal workload.

IV. Methodology

This section explains our approach for collecting specific firmware samples and why it is essential to use verifiable firmware for further studies.

Dataset. Our dataset contains 5,728 firmware samples and 850,555 pre-installed apps. As shown in Table I, we have 40.89% Android 10 and 13.88% Android 11 firmware samples. Every Android app defines within its *AndroidManifest.xml*, an XML file with essential app details as for example the app's package name (e.g., *com.android.settings*). The Android framework assures that every app on the system uses a unique package name. Using the package name, we can quantify the diversity of pre-install apps in the dataset by examining the number of unique package names. The dataset contains 3,182 unique packages, meaning that we have mostly apps with the same package name from different firmware or, in other words, apps with multiple builds. To be more precise, we calculate for every app file a SHA-256 hash and use this hash to identify the number of unique apps in our dataset. We have 75,141 unique apps in our corpus, meaning that our dataset contains 8.83% unique app builds. Note that we have 5,728 unique firmware images and that an Android app can behave differently based on its dependencies on the firmware environment. Therefore, it does make sense to analyse duplicated apps in cases where the environment influences the app's behaviour directly. For example, such influences can be: Access to shared native libraries of the operating system, environment variable checks, conditional dynamic code loading, or inter-process communication.

At the time of writing, not all smartphone manufacturers make their stock firmware freely available, and it seems unlikely to change soon. Various websites on the web offer stock firmware for specific OSV's for download but without the official[5] cryptographic checksums, it is not possible to verify the integrity of these firmware samples, and therefore the possibility of malicious modifications exists as mentioned before. Moreover, OSV's and OSM's often do not publish the cryptographic checksums of their firmware, so it is not possible to conduct any cross-reference checks. As a result, verifying the integrity of the firmware from unofficial sources is currently not possible without insider knowledge. Using a dataset of unverifiable firmware can lead to wrong conclusions. For example, if we detect a harmful app in vendor A's firmware, we cannot verify if the malicious app was initially included in the firmware or later modified, resigned, and then redistributed.

Furthermore, using unverifiable firmware makes the security analysis more challenging because attackers and developers tend to change the meta-data used for the analysis. For example, Siewierski [49] found out that attackers had used tampered build fingerprints to hinder analysts from identify to which manufacturer or brand a firmware belongs to. Such modifications are typical for tampered firmware but are often hard to detect because we cannot verify the integrity of the firmware without the vendor's cryptographic checksums. In this study, we, therefore, only use verifiable firmware downloaded from official vendor websites. This allows us to draw conclusions about specific vendors without worrying that the firmware is not original stock firmware. Furthermore, as mentioned before, we focus our analysis on the static analysis of pre-installed apps and not on other parts of the Android firmware for this study. More precisely, we analyse the usage of permissions and advertising trackers within pre-installed apps by using the static analysis tools AndroGuard [27] and Exodus [20].

V. Permission usage

In this section we give a detailed examination of the results extracted with AndroGuard [27]. Using FirmwareDroid we analysed the permission usage of all collected Android apps in our dataset. AndroGuard extracts permission declarations directly from the *AndroidManifest.xml* file and can list the used custom or framework API permissions. Therefore, using AndroidGuard gives us precise results of the declared permissions but cannot test if the permission is actually used at runtime.

We compare the permission usages of the OS vendors by categorising them into the three permission levels defined by the Android framework: signature, dangerous, and normal. Analysing the declared app permissions allows us to compare how many apps access potentially dangerous APIs such as the camera, location, or microphone and to answer our research question RQ_1. For a comparison at a finer level of granularity we use a stratified sampling approach to compare the app manifests' permissions of randomly selected apps.

As a first step, we aggregate the number of permissions declared for each OS vendor, OS version, and permission level. Using this data, we determine the ratios for each permission level and conclude that 88.14% of the permissions are signature-, 3.56% are dangerous-, and 8.21% are normal declared permissions over all vendors. When having a look at some specific vendors, we can conclude the following facts without using stratified sampling. 10.22% of the permissions found in Google Android 10 firmware are dangerous permissions. Compared to Google Android 11 firmware, we see that the dangerous permission usage has decreased to 3.6% and that the number of normal permissions on Android 11 increased to 23.83% from its previous version. We detect on Resurrection Remix OS 801 (0.23%) dangerous permissions, which is the fewest number of dangerous permissions detected overall, but it has with 338,758 (95.91%), the highest number of signature permissions. GrapheneOS has the highest number of normal permissions (59.54%). In addition, we detect 35.26% signature-based permissions in GrapheneOS, whereas LineageOS uses fewer dangerous permissions than GrapheneOS with 3.08% on its Android 11 firmware. Please note that some of these numbers have to be considered biased as the number of samples differs.

[5] Published by the original operating system developer.

TABLE II
TOP 20 OF REQUESTED DANGEROUS PERMISSIONS FOR PRE-INSTALLED APPS ON ANDROID 10 AND 11.

# Rank	Permission	RROS V10	Paranoid v10	OmniROM v11	OmniROM v10	LineageOS v11	LineageOS v10	GrapheneOS v11	Google V11	Google v10	Carbon v10
1	android.permission.WRITE_EXTERNAL_STORAGE	16,512	2,895	4,755	12,676	11,652	5,251	378	5,305	6,218	6,142
2	android.permission.READ_EXTERNAL_STORAGE	12,682	2,680	4,486	11,871	8,837	3,701	378	4,258	4,948	5,746
3	android.permission.READ_PHONE_STATE	10,586	2,024	2,699	8,868	7,639	3,695	280	5,545	6,584	3,874
4	android.permission.READ_CONTACTS	9,224	2,270	2,771	8,005	6,664	3,112	302	4,284	5,332	4,008
5	android.permission.ACCESS_FINE_LOCATION	7,273	1,813	3,164	6,806	6,729	2,209	262	4,649	4,946	3,388
6	android.permission.GET_ACCOUNTS	7,890	1,877	2,171	6,592	4,960	2,666	248	4,234	5,328	3,218
7	android.permission.ACCESS_COARSE_LOCATION	6,139	1,835	2,042	5,823	4,490	1,800	234	4,556	5,088	2,929
8	android.permission.WRITE_CONTACTS	6,137	1,482	1,651	4,692	4,042	2,066	176	2,394	2,992	2,356
9	android.permission.CALL_PHONE	5,314	1,336	1,720	4,640	4,247	1,786	162	2,654	3,304	2,636
10	android.permission.CAMERA	5,055	1,112	1,498	4,203	3,043	1,152	180	2,588	3,072	1,890
11	android.permission.RECORD_AUDIO	4,721	1,129	1,182	3,026	3,760	1,531	126	2,698	3,234	1,215
12	android.permission.READ_CALL_LOG	3,615	940	1,399	3,248	3,169	1,209	126	1,463	1,782	1,767
13	android.permission.SEND_SMS	3,630	846	1,198	3,249	2,761	1,218	108	1,735	2,222	1,724
14	android.permission.READ_SMS	3,169	846	1,036	2,720	2,761	1,009	108	1,723	2,108	1,408
15	android.permission.WRITE_CALL_LOG	2,431	651	998	2,190	2,256	809	90	1,197	1,588	1,135
16	android.permission.READ_CALENDAR	2,925	652	600	1,989	1,332	1,000	54	931	1,248	862
17	android.permission.WRITE_CALENDAR	2,697	468	600	1,916	1,332	1,000	54	665	780	862
18	android.permission.ACCESS_BACKGROUND_LOCATION	1,519	273	1,073	1,404	1,437	200	54	2,017	1,502	273
19	android.permission.PROCESS_OUTGOING_CALLS	1,422	395	420	1,358	1,040	446	36	1,064	1,328	631
20	com.android.voicemail.permission.ADD_VOICEMAIL	1,200	281	400	1,058	884	400	36	532	690	589

Furthermore, it is essential to notice that signature permission may give similar privileged access to an app as dangerous permissions. Signature permissions give access if the requiring app is signed with the same signing key as the app declaring the permission. In general, signature permissions are either used for custom permissions or access to privileged system components. Depending on the usage, they give an app access to critical system resources like Android's PackageManager.

As a second step, we assessed the number of custom permissions. We created a list of all the requested permissions for Android 10 and 11 and counted how often apps requested the permissions. This method allows us to investigate how many custom permissions were used or requested.

In total, we detected 1,113 defined permissions. We found out that 518 (46.54%) were permissions from the main *'android.'* package, 68 (6.11%) of *'com.android.'*, 209 (18.78%) of *'com.google.'*, and 318 (28.57%) were third-party vendors packages. Consequently, 28.57% of the permissions in our dataset are custom third-party permissions. We further evaluated the frequencies of permission requests for Android 10 and 11 to give the reader an insight into the most used ones. In Table II we show the top 20 of the most requested dangerous permissions (by the total number of declarations) for the Android 10 and 11 firmware. The most used dangerous permissions are write and read to the external storage, followed by reading of the phone state. The fourth most used permission is the reading the user contacts, and the fifth most used is the access to the fine location of the phone. Under the top 20 of all permissions (grouped by normal, signature, and dangerous), there are six dangerous permissions, six signature permissions, and eight normal permissions.

As a third step, we apply stratified sampling and select randomly from every vendor 1,000 unique apps (by SHA-256) for comparison from Android version eight and above. We show in Table III the result of this comparison. In our experiment an average of 2,777 signature, two system, 119 dangerous, and 505 normal permissions were used. Signature permissions are the most frequently used permission by all vendors. As shown in Table III, Google used the fewest dangerous permissions, with 32 (1.19%), and Paranoid used with 197 (2.4%) the highest number of dangerous permissions in our experiment.

TABLE III
COMPARISON OF APP PERMISSIONS WITH STRATIFIED SAMPLING.

ROM Vendor	# signature	# system	# dangerous	# normal	# Total
Google	2,297 (85.36%)	10 (0.37%)	32 (1.19%)	352 (13.08%)	2,691
GrapheneOS	1,334 (75.62%)	0 (0%)	90 (5.1%)	340 (19.27%)	1,764
Paranoid	7,299 (88.99%)	0 (0%)	197 (2.4%)	706 (8.61%)	8,202
Carbon	2,939 (73.33%)	0 (0%)	170 (4.24%)	899 (22.43%)	4,008
LineageOS	1,946 (83.52%)	3 (0.13%)	101(4.33%)	280 (12.02%)	2,330
OmniROM	1,841 (71.03%)	0 (0%)	97 (3.74%)	654 (25.23%)	2,592
RROS	1,785 (79.94%)	3 (0.13%)	143 (6.4%)	302 (13.52%)	2,233
Total	**19,441**	**16**	**830**	**3,533**	**23,820**
Average	**2,777**	**2**	**119**	**505**	**3,403**

TABLE IV
TOTAL NUMBER OF DETECTED TRACKERS WITHIN ALL PRE-INSTALLED APPS.

Vendor	# Apps >0 trackers	# Apps no trackers	# Reports	% of Apps
Google	19,757	149,544	169,392	11.66%
GrapheneOS	0	3,597	3,597	0.00%
Paranoid	1,279	18,053	19,332	6.62%
Carbon	415	68,486	68,901	0.60%
LineageOS	162	122,676	122,838	0.13%
OmniROM	879	291,998	292,877	0.30%
RROS	2,105	171,105	173,210	1.22%
Total	**24,597**	**825,459**	**850,147**	**20.53%**

In conclusion, we can say that signature permissions are the most frequently requested and used permissions by pre-installed apps. The fact that pre-installed apps often do not have to ask the user's consent for signature and dangerous permissions seems to be a concerning trend. Our random sampling experiment shows that the number of requested dangerous permissions is relatively small compared to the number of signature permissions used.

TABLE V
ADVERTISING TRACKER LIBRARY DETECTION RATES BY EXODUS.

OS Vendor	**Version**	**Google Analytics**	**Google AdMob**	**Google CrashLytics**	**Amazon Analytics (Amazon insights)**	**Google Firebase Analytics**	**Google Manager**	**Facebook Share**	**Facebook Analytics**	**Facebook Login**	**Mapbox**	**OpenTelemetry (OpenCensus, OpenTracing)**	**AutoNavi Amap**	**Total**
Google	v0	3,993	1,714	61	0	1,563	795	0	0	0	0	0	0	**8,126**
	v7	739	289	68	0	702	187	0	0	0	0	0	0	**1,985**
	v8	0	567	136	1,168	660	334	0	0	0	0	0	0	**2,865**
	v9	1,840	790	408	38	1,204	484	0	0	0	0	204	0	**4,968**
	v10	2,434	1,268	772	66	2,920	474	0	0	0	0	582	0	**8,516**
	v11	1,294	1,106	532	48	2,557	241	0	0	0	0	0	0	**5,778**
Carbon	v8	0	0	0	0	0	0	0	0	0	0	0	0	**0**
	v9	13	0	73	0	88	0	11	11	11	0	0	0	**207**
	v10	0	0	316	0	273	0	43	43	43	0	0	0	**718**
RROS	v10	566	497	443	0	1,512	96	0	0	0	0	133	37	**3,284**
LineageOS	v10	2	0	0	0	18	0	0	0	0	0	0	0	**20**
	v11	40	7	4	7	98	0	4	4	4	0	0	0	**168**
GrapheneOS	v11	0	0	0	0	0	0	0	0	0	0	0	0	**0**
Paranoid	v10	637	364	278	0	728	0	5	5	5	0	91	0	**2,113**
OmniROM	v0	5	5	5	0	20	5	0	0	0	0	0	0	**40**
	v4	0	0	0	0	0	0	0	0	0	0	0	0	**0**
	v5	1	0	0	0	0	0	0	0	0	0	0	0	**1**
	v6	0	0	0	0	6	0	0	0	0	0	0	0	**6**
	v7	10	10	10	0	20	10	0	0	0	0	0	0	**60**
	v8	0	0	0	0	0	0	0	0	0	0	0	0	**0**
	v9	3	0	0	0	3	0	0	0	0	0	0	0	**6**
	v10	439	220	0	0	439	73	0	0	0	220	0	125	**1,516**
	v11	264	109	0	0	264	52	0	0	0	109	0	0	**798**
Total	**All**	**12,280**	**6,946**	**3,106**	**1,327**	**13,075**	**2,751**	**63**	**63**	**63**	**329**	**1,010**	**162**	**41,175**

VI. TRACKING THE TRACKERS

We scanned 850,147 of the apps in our dataset with Exodus and found 41,175 known trackers. Please note that we were not able to scan 408 apps because Exodus crashed for unknown reasons on these samples. Table V shows the summarised results of the advertising tracker libraries detected by Exodus. We analysed the data from different points of view, such as the number of detections per Android version and OSV.

We detected the most significant number of advertising tracker libraries in Google firmware with 32,238 trackers. Taking into consideration that the number of app samples differs for each OSV, we extracted the number of apps with at least one tracker and the number of apps without any detected trackers. We show the results in Table IV. Google has with 11.66% (19,757) the highest number of apps with at least one tracker followed by Paranoid 6.62% (1,279) and RROS with 1.22% (2,105). In total 24,597 (20.53%) of the pre-installed apps in our dataset use advertising tracker libraries.

Overall, as shown in Table V the most common trackers in our datasets are Google Firebase Analytics (13,049, 31.74%), Google Analytics (12,269, 29.85%), and Google AdMob (6,936, 16.87%). Amazon Analytics seems to be most used on Android 8 on Google firmware with 1,168 (2.84%) detected trackers. After Android 8, the detection rates of Amazon Analytics decrease for all subsequent Android versions by Google. We can only assume that this is either due to undetected trackers by Exodus or due to Google policy changes. Another effect that could be due to undetected trackers is the low number of detected Facebook Analytics. The Exodus community reports on their official website [20] that around 18% (19,296 apps[6]) of the scanned apps in the Google Play Store use Facebook Analytics. However, on our pre-installed app dataset, we could only detect 63 (0.15%) apps that use Facebook Analytics.

TABLE VI
COMPARISON OF DETECTED TRACKERS WITH STRATIFIED SAMPLING (1K APPS).

Vendor	**# Apps >0 trackers**	**# Apps with no trackers**
Google	847	153
GrapheneOS	0	1,000
Paranoid	15	985
Carbon	5	995
LineageOS	4	996
OmniROM	14	986
RROS	18	982
Total	**903**	**6,097**

Nevertheless, an interesting fact is that the two OSVs focusing on privacy, LineageOS (188, 0.15%) and GrapheneOS (0, 0%), use a rather low number of advertising tracker libraries. Other custom ROM providers like OmniROM (2,427, 0.83%) and Paranoid (2,113, 10.93%) have more detections compared to LingeageOS or GrapheneOS.

For a better comparison, we use stratified sampling once more to select 1,000 pre-installed apps randomly from every vendor. As shown in Table VI the number of trackers in Google apps is significantly higher than for all the other vendors. Undoubtedly, this could be due to Exodus detection rate for Google libraries as these are more prominent in the Google Play Store than other tracker libraries. However, these experiments showcase the potential of FirmwareDroid as a framework for the comparison of firmware from different vendors. We see using tools such as Exodus for data gathering only as the first step in analysing and benchmarking such privacy issues.

[6] Statistics from September 2021.

VII. Results Summary

In this section we summarize the findings of the analyzed data in the last Sections and answer our research questions defined in Section I.

App permissions. In RQ_1, our goal was to determine how many dangerous permissions pre-installed apps use and which permissions are the most used ones. We found out that under the top 20 permissions were six dangerous permissions: read+write of the external storage, read of the phone state, read of contacts, access to the fine location, and the get accounts permission. We determined that 88.14% of the permissions are signature-, 3.56% are dangerous-, and 8.21% are normal declared permissions. We conclude, that OSVs have significant usage difference of dangerous and signature permissions and that the average use of dangerous permission (55,859, 3.56%) seems to be rather high. The complete evaluation of the permission data is shown in Section V.

Advertising tracker libraries. Our goal for RQ_2 was to determine how common are advertising tracker services within pre-installed apps. We found out that in total 24,597 (20.53%) of the pre-installed apps in our dataset use ad trackers. Thus, the data supports the premise that pre-installed apps of commercial vendors like Google include high amounts of tracking libraries. We detected the highest number of ad trackers (32,238) within Google firmware, followed by Resurrection Remix OS with 3,284 detected trackers. The most often detected ad trackers were Google Firebase Analytics (13,049, 31.74%) and Google Analytics (12,269, 29.85%). The least number of ad trackers were detected in LineageOS (188, 0.15%) and GrapheneOS (0, 0%). This study is enough to point out that we could identify 41,108 known ad trackers within 850,555 pre-installed apps. We assume that the detection rates for some specific ad trackers like Facebook and Amazon Analytics are too low when compared to the rates reported by Exodus Privacy on their official website[7] for Google Play Store apps. It is up to future research to prove this assumption and to further enhance detection mechanisms.

VIII. Discussion

Pre-installed apps have some fundamental differences compared to apps from any app store, making the analysis more challenging from a technical perspective. Moreover, we will resolve more technical challenges to further enhance FirmwareDroid as a framework and to enhance Android firmware analysis in general.

Unpacking. The number of formats FirmwareDroid supports is limited and might needs to be enhanced in the future to support more Android image formats. Integrating additional unpacking tools such as unblob [50] or binwalk [51] might help to overcome these shortcomings in future versions of FirmwareDroid.

Optimized Dalvik-Code. OEM manufacturers use dex optimizing file formats to increase the performance of pre-installed apps on device start-up. Formats like .odex, .vdex, .art, and .cdex allow splitting the Dalvik byte-code [52] and app resources into separate files on the file system, which the Android Runtime (ART) [53] optimizes for different CPU architectures (mainly x86, ARM). Consequently, Android splits the app's code into several files that a researcher needs to consider for security analysis. More precisely, it leads to the fact that ART loads different Dalvik-byte code depending on the CPU architecture the app is executed on. As a result, analysing just the apk of a pre-installed app itself is not enough to get a complete static analysis of the actual code executed at run-time on the device. This is a significant difference from analysing regular apps from any app store because an apk of a pre-installed app may not include large parts of the code. Thus, focusing only on apk files may lead to wrong conclusions. An apparent limitation of our method is that FirmwareDroid extracts, whenever present all the optimised Dalvik byte code files found but is missing a module that can deoptimise the Dalvik-byte code into one apk file so that all the static analysis tools can scan the complete code.

Native Libraries. A major source of limitation is that none of our static analysis tools can scan native libraries for vulnerabilities or malware. Most static analysis tools focus on scanning an apk file but do not scan any referenced native libraries. Scanners like Androwarn [13] and QARK [15] can detect that native libraries are loaded to some extent, but they do not check the libraries for vulnerabilities or malicious behaviour. In addition, a recent study by Wang et al. [54] has shown that malware developers are targeting Android software development kits (SDK), which are often included as native libraries, to harvest data from apps integrating the SDK. We assume that libraries for pre-installed apps could be similar targets for such kind of attacks. Thus, it is up to future research to have better static analysis tools that are capable of scanning native libraries as well.

Permissions. We can extract how many permissions a specific app uses. However, the number of permissions is often not sufficient to come to the conclusion if one firmware is more trustworthy than another. This is especially true when we consider the fact that signature permissions might give the same access rights as dangerous permissions. Therefore, more sophisticated approaches, such as combining static- and dynamic analysis tools, are necessary to further automate the analysis and benchmark process.

Tracking detection. Detecting advertisement trackers with Exodus can only be seen as the first step towards an automated analysis of tracking numbers. Additional tools such, for example, LibRadar [19], or dynamic analysis techniques, as described by He et al. [55], could help further enhance the detection of third-party libraries.

Tool limitations. Some of the static analysis tools integrated into FirmwareDroid are not capable of scanning all apk files in our dataset. For example, SUPER [14] is only capable of scanning 51% of the apps in our corpus due to technical limitations of the tool itself. Another problem we faced, is that not every tool scales well in terms of time consumption. For example, it would take several months to scan all apps

[7] https://reports.exodus-privacy.eu.org/en/trackers/stats/

in our corpus with the QARK [15] due the fact that QARK decompiles every apk file, which is a time-consuming task, especially for large apk files. Moreover, we have experienced similar performance issues with Androguard's [27] call-graph feature, which can take several minutes or in worst case hours to generate a call-graph for a single apk file. Despite slow performance, such tools can be useful to analyse specific apk files for researchers. Thus, we integrated these tools into FirmwwareDroid for other use-cases, such as malware analysis.

Error rate. Another limitation of many static analysis tools is that their error rate is unknown. For example, we tested APKleaks [18] which is capable of detecting information leakage vulnerabilities (e.g., exposed Google API keys) in Android apps. Testing the tool on our corpus showed that over 90% of the detected information leakages by APKLeaks [18] in our corpus were false positives. Analysing the root cause led to the fact, that APKleaks probes for regex patterns[8] of known secrets but does not conduct sanity checks, such as considering the location where the information leakage was detected. Thus, the tool would often report SHA-256 digests found in the *MANIFEST.MF* file, as generic secrets. However, the *MANIFEST.MF* file contains SHA-256 digests for every jar file included in the apk and thus, it is an obvious false positive.

Nevertheless, including such tools into FirmwareDroid can still be helpful for researchers and the developer community. For example, even with APKLeaks high false positive rate, we were able to detect hard coded Github credentials in one of the firmware samples in our corpus. In addition, we are working on an improved version of APKLeaks to minimise the false positive rate in future versions of the tool.

IX. Conclusion and Final Remarks

We showed significant differences between different firmware vendors regarding permission usages of Android apps. More generally, our findings are consistent with other research [7] showing that pre-installed apps have a concerning amount of access to dangerous permissions and therefore to privacy-related content on smartphones. In addition, we study the number of ad trackers detected on pre-installed apps and compared different operating system vendors, showing that major differences in the usage of ad trackers exist.

The majority of the static analysis tools we used in this study are non-profit open-source projects that volunteers maintain. Unfortunately, the tools often have a time delay to keep up with Android's latest changes. Therefore, it is likely that many of the newer ad trackers or permissions are not detected and that the used tools have unknown error rates.

Concluding, we showed that utilizing static analysis tools with FirmwareDroid can help gathering the necessary data for further studies. It seems that we have just seen the tip of the iceberg when it comes to detecting ad tracking libraries and that large parts of pre-installed app analysis remains unknown.

[8] https://github.com/dwisiswant0/apkleaks/blob/master/config/regexes.json

X. Availability

Code: FirmwareDroid is open-source and freely available on GitHub: https://github.com/FirmwareDroid/FirmwareDroid
Dataset: Our dataset is free of charge available for accredited researchers and students. More information on GitHub: https://github.com/FirmwareDroid/Datasets

References

[1] B. Grill, M. Ruthven, and X. Zhao, "Detecting and eliminating chamois, a fraud botnet on android," https://android-developers.googleblog.com/2017/03/detecting-and-eliminating-chamois-fraud.html, March 2017, accessed: September 2021.

[2] C. Brook, "Google eliminates android adfraud botnet chamois," https://threatpost.com/google-eliminates-android-adfraud-botnet-chamois/124311/, March 2017, accessed: September 2021.

[3] F. Y. Rashid, "Chamois: The big botnet you didn't hear about," https://duo.com/decipher/chamois-the-big-botnet-you-didnt-hear-about, April 2019, accessed: September 2021.

[4] L. Siewierski, "Pha family highlights: Triada," https://security.googleblog.com/2019/06/pha-family-highlights-triada.html, June 2019, accessed: September 2021.

[5] a. J. Snow, "Triada: organized crime on android," https://www.kaspersky.com/blog/triada-trojan/11481/, March 2016, accessed: September 2021.

[6] N. Cohen, "Android: Researchers tell troubling findings of pre-installed software," 03 2019. [Online]. Available: https://techxplore.com/news/2019-03-android-pre-installed-software.html

[7] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1039–1055.

[8] "Offical LineageOS website," https://lineageos.org/, 2021, accessed: September 2021. [Online]. Available: https://lineageos.org/

[9] "Offical GrahpeneOS website," https://grapheneos.org/, 2021, accessed: September 2021. [Online]. Available: https://grapheneos.org/

[10] B. Lau, J. Zhang, A. R. Bereford, D. Thomas, and R. Mayrhofer, "Uraniborgs device preloaded app risks scoring metrics," https://www.ins.jku.at/publications/2020/Lau_2020_Uraniborg_Scoring_Whitepaper_20200827.pdf, August 2020, accessed: September 2021.

[11] N. G. ioXt Certification Lab, "Pixel 4/4xl and pixel 4a ioxt audit - google," https://research.nccgroup.com/wp-content/uploads/2020/08/NCC_Group_Google_GOOG065C_Report_2020-08-13_v2.0.pdf, August 2020, accessed: September 2021.

[12] Yu-Cheng Lin, "AndroBugs framework," November 2015, accessed: September 2021. Conference Talk: https://www.blackhat.com/docs/eu-15/materials/eu-15-Lin-Androbugs-Framework-An-Android-Application-Security-Vulnerability-Scanner.pdf. [Online]. Available: https://github.com/AndroBugs/AndroBugs_Framework

[13] Thomas Debize, "Androwarn," https://github.com/maaaaz/androwarn, 2012, accessed: September 2021. [Online]. Available: https://github.com/maaaaz/androwarn

[14] I. Eguia, "SUPER Android Analyzer," https://github.com/SUPERAndroidAnalyzer/super, 2016, accessed: September 2021. [Online]. Available: https://github.com/SUPERAndroidAnalyzer/super

[15] LinkedIn, "Quick Android Review Kit (QARK)," https://github.com/linkedin/qark/, LinkedIn Corp, 2015, accessed: September 2021. [Online]. Available: https://github.com/linkedin/qark/

[16] Q. E. Team, "Quark Engine," https://github.com/quark-engine/quark-engine, 2019, accessed: September 2021. [Online]. Available: https://github.com/quark-engine/quark-engine

[17] A. Martín García, R. Lara-Cabrera, and D. Camacho, "A new tool for static and dynamic android malware analysis," September 2018, pp. 509–516.

[18] "Apkleaks," https://github.com/dwisiswant0/apkleaks, 2021, accessed: September 2021. [Online]. Available: https://github.com/dwisiswant0/apkleaks

[19] Ma, Ziang and Wang, Haoyu and Guo, Yao and Chen, Xiangqun, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, p. 653–656, accessed: September 2021. [Online]. Available: https://doi.org/10.1145/2889160.2889178

[20] Exodus Privacy, “Exodus Privacy offical website,” https://reports.exodus-privacy.eu.org/en/, 2017, accessed: September 2021. [Online]. Available: https://reports.exodus-privacy.eu.org/en/

[21] K. Kollnig, P. Dewitte, M. V. Kleek, G. Wang, D. Omeiza, H. Webb, and N. Shadbolt, “A fait accompli? an empirical study into the absence of consent to third-party tracking in android apps,” in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, August 2021, pp. 181–196. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/kollnig

[22] Y.-T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger, “Polyscope: Multi-policy access control analysis to compute authorized attack operations in android systems,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 2579–2596, accessed: September 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yu-tsung

[23] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi, “Seapp: Bringing mandatory access control to android apps,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 3613–3630. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/rossi

[24] G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. Butler, “Bigmac: Fine-grained policy analysis of android firmware,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020, pp. 271–287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

[25] Y. Aafer, X. Zhang, and W. Du, “Harvesting inconsistent security configurations in custom android roms via differential analysis,” in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, August 2016, pp. 1153–1168, accessed: September 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aafer

[26] N. T. Cam, V.-H. 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)*, 2017, pp. 340–343.

[27] A. Desnos and G. Gueguen, “Android: From reversing to decompilation,” *Proc. of Black Hat Abu Dhabi*, 01 2011, tool online available at: https://github.com/androguard/androguard.

[28] H. Gao, S. Cheng, and W. Zhang, “Gdroid: Android malware detection and classification with graph convolutional network,” *Computers & Security*, vol. 106, p. 102264, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821000882

[29] P. Sivakumaran and J. Blasco, “A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, August 2019, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/sivakumaran

[30] J. Jiang, Z. Liu, M. Yu, G. Li, S. Li, C. Liu, and W. Huang, “Hetersupervise: Package-level android malware analysis based on heterogeneous graph,” in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 328–335.

[31] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, “FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020, pp. 2379–2396, accessed: September 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/elsabagh

[32] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin, “Android smarttvs vulnerability discovery via log-guided fuzzing,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 2759–2776, accessed: September 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/aafer

[33] M. Zheng, M. Sun, and J. C. Lui, “Droidray: A security evaluation system for customized android firmwares,” June 2014.

[34] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, “Large-scale security measurements on the android firmware ecosystem,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1257–1268. [Online]. Available: https://doi.org/10.1145/3510003.3510072

[35] “Puppeteer,” https://github.com/puppeteer/puppeteer/, 2020, accessed: September 2021. [Online]. Available: https://github.com/puppeteer/puppeteer/

[36] “Offical Paranoid website,” https://paranoidandroid.co/, 2021, accessed: September 2021. [Online]. Available: https://paranoidandroid.co/

[37] “Offical OmniROM website,” https://omnirom.org/, 2021, accessed: September 2021. [Online]. Available: https://omnirom.org/

[38] “Offical Resurrection Remix OS website,” https://resurrectionremix.com/, 2021, accessed: September 2021. [Online]. Available: https://resurrectionremix.com/

[39] “Factory images for nexus and pixel devices,” https://developers.google.com/android/images, 2021, accessed: September 2021. [Online]. Available: https://developers.google.com/android/images

[40] T. Sutter, “Firmwaredroid: Security analysis of the android firmware ecosystem,” 2021. [Online]. Available: https://arxiv.org/abs/2112.08520

[41] “Brotli github repository,” https://github.com/google/brotli, 2020, accessed: September 2021. [Online]. Available: https://github.com/google/brotli

[42] Google, “Images,” https://source.android.com/devices/bootloader/partitions-images, 2020, accessed: September 2021. [Online]. Available: https://source.android.com/devices/bootloader/partitions-images

[43] ——, “Standard partitions,” https://source.android.com/devices/bootloader/partitions, 2020, accessed: September 2021. [Online]. Available: https://source.android.com/devices/bootloader/partitions

[44] “simg2img,” https://github.com/anestisb/android-simg2img, 2020, accessed: September 2021. [Online]. Available: https://github.com/anestisb/android-simg2img

[45] “Fuse ext2,” https://github.com/alperakcan/fuse-ext2, 2020, accessed: September 2021.

[46] “Imgpatchtools github repository,” https://github.com/erfanoabdi/imgpatchtools, 2021, accessed: September 2021. [Online]. Available: https://github.com/erfanoabdi/imgpatchtools

[47] “Ubi reader github repository,” https://github.com/jrspruitt/ubi_reader, accessed: September 2021.

[48] *APKiD*. Rednaga, July 2016, accessed: September 2021. Slides online available at: http://hitcon.org/2016/CMT/slide/day1-r0-e-1.pdf. [Online]. Available: https://github.com/rednaga/APKiD

[49] Łukasz Siewierski, “Challenges in android supply chain analysis,” https://youtu.be/9IozYfGWoRE, February 2020, accessed: September 2021. [Online]. Available: https://www.youtube.com/watch?v=9IozYfGWoRE&feature=youtu.be&ab_channel=RSAConference

[50] Onekey, “Unblob.” [Online]. Available: https://github.com/onekey-sec/unblob

[51] ReFirmLabs, “Binwalk.” [Online]. Available: https://github.com/ReFirmLabs/binwalk

[52] Google, “Dalvik bytecode,” https://source.android.com/devices/tech/dalvik/dalvik-bytecode, 2020, accessed: September 2021. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[53] ——, “Android runtime (art) and dalvik,” https://source.android.com/devices/tech/dalvik, 2020, accessed: September 2021. [Online]. Available: https://source.android.com/devices/tech/dalvik

[54] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang, and Y. Zhang, “Understanding malicious cross-library data harvesting on android,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 4133–4150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-jice

[55] Y. He, X. Yang, B. Hu, and W. Wang, “Dynamic privacy leakage analysis of android third-party libraries,” *Journal of Information Security and Applications*, vol. 46, pp. 259–270, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212618304356