Large-scale Security Measurements on the Android Firmware Ecosystem

Qinsheng Hou[1,2,3], Wenrui Diao[1,2], Yanhao Wang[3(✉)], Xiaofeng Liu[1,2], Song Liu[3], Lingyun Ying[3], Shanqing Guo[1,2,6(✉)], Yuanzhi Li[3], Meining Nie[3], and Haixin Duan[4,5]

[1]*School of Cyber Science and Technology, Shandong University*

[2]*Key Laboratory of Cryptologic Technology and Information Security of Ministry of Education, Shandong University*

[3]*QI-ANXIN Technology Research Institute,* [4]*Institute for Network Science and Cyberspace, Tsinghua University*

[5]*Tsinghua University-QI-ANXIN Group JCNS,* [6]*Quancheng Laboratory, Jinan, China*

houqinsheng@mail.sdu.edu.cn, wangyanhao136@gmail.com, guoshanqing@sdu.edu.cn

ABSTRACT

Android is the most popular smartphone platform with over 85% market share. Its success is built on openness, and phone vendors can utilize the Android source code to make products with unique software/hardware features. On the other hand, the fragmentation and customization of Android also bring many security risks that have attracted the attention of researchers. Many efforts were put in to investigate the security of customized Android firmware. However, most of the previous work focuses on designing efficient analysis tools or analyzing particular aspects of the firmware. There still lacks a panoramic view of Android firmware ecosystem security and the corresponding understandings based on large-scale firmware datasets. In this work, we made a large-scale comprehensive measurement of the Android firmware ecosystem security. Our study is based on 6,261 firmware images from 153 vendors and 602 Android-related CVEs, which is the largest Android firmware dataset ever used for security measurements. In particular, our study followed a series of research questions, covering vulnerabilities, patches, security updates, and pre-installed apps. To automate the analysis process, we designed a framework, AndScanner, to complete ROM crawling, ROM parsing, patch analysis, and app analysis. Through massive data analysis and case explorations, several interesting findings are obtained. For example, the patch delay and missing issues are widespread in Android images, say 24.2% and 6.1% of all images, respectively. The latest images of several phones still contain vulnerable pre-installed apps, and even the corresponding vulnerabilities have been publicly disclosed. In addition to data measurements, we also explore the causes behind these security threats through case studies and demonstrate that the discovered security threats can be converted into exploitable vulnerabilities via 38 newfound vulnerabilities by our framework, 32 of which have been assigned CVE/CNVD numbers. This study provides much new knowledge of the Android firmware ecosystem with deep understanding of software engineering security practices.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510072

CCS CONCEPTS

• **Software and its engineering → Software safety**.

KEYWORDS

Android Firmware Ecosystem, Security Measurements, Security Patches, Pre-installed Apps

ACM Reference Format:

Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. 2022. Large-scale Security Measurements on the Android Firmware Ecosystem. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510072

1 INTRODUCTION

Nowadays, Android plays a fundamental role in people's daily life with an 85% market share of smartphones [33]. Besides smartphones, other devices running Android OS can be seen everywhere, like smartwatches and smart TVs. Such success is largely credited with the openness of Android, which allows vendors to build their own customized ROM images for different devices based on AOSP (Android Open Source Project) [4].

On the other hand, the openness of Android comes with risks. The large number of customized ROM images built by different vendors make the Android ecosystem seriously fragmented [45], which is a big challenge to guarantee the security of the Android ecosystem. To vendors, the technical abilities and the attitudes towards security result in the differential security of their ROM images. For example, although Google releases security patches to the security bulletins [5] every month, many vendors did not follow up the patches in time [29], even some vendors just change the updated date without fixing vulnerabilities [55]. Also, the pre-installed apps may have security problems. It could be caused by the phone vendors themselves, like Themes (`CVE-2019-15442`) in Samsung devices, which is a vulnerable pre-installed app with an arbitrary app installation vulnerability. Some problems are caused by the suppliers' apps, like WfoService (`CVE-2019-15378`) in a Panasonic device, a pre-installed app from the chip-supplier MediaTek allowing any app co-located on the device to modify any system properties.

In recent years, many efforts have been taken to investigate and improve the Android ecosystem security. For example, Elsabagh

et al. [38] presented FirmScope, an automated analysis system to uncover different types of vulnerabilities in pre-installed apps. Lell et al. [49] implemented an app named SnoopSnitch to do binary-only analysis on Android patches without access to source code. Zhang et al. [59] investigated the Android kernel patch ecosystem, revealing the relationship among different parties as well as the bottleneck in patch propagation. Gamba et al. [41] studied the ecosystem of pre-installed Android apps and its potential impact on consumers. However, most of the previous work focuses on designing efficient analysis tools or analyzing particular aspects of the firmware. There still lacks a panoramic view of Android firmware security based on large-scale firmware datasets and the corresponding understanding behind the current security situations. In fact, some research questions cannot be solved well without large-scale measurements and numerous case studies. For example, it has been noted that some phone vendors failed to fix some CVEs confirmed by Google [49]. However, *is the security patch missing issue a common security risk or a false statement (with rare cases)? Why do the vendors miss some patches? Which phone models perform well (or poorly) in the patch missing aspect, and what are the reasons?*

Our work. In this work, we carried out a large-scale comprehensive security measurement of the security of Android firmware from both the system and pre-installed app levels. Also, we try to uncover the causes and influences behind the security risks. To guide our measurement, we refine four important research questions on Android firmware security, including:

- **Q1** ***Do the Android phone vendors follow up the released AOSP security patches in time?*** Regular security update plays an essential role in protecting mobile devices. Android partners (vendors) are notified of all vulnerability details at least a month before the public disclosure, and they have enough time window to update their firmware. However, whether the vendors indeed follow up the security update of AOSP in time is still a question. We are also curious about the causes of the patch delay.

- **Q2** ***Whether the patching claims of phone vendors are reliable?*** Though the phone has been updated to the latest security patch level, there is still no guarantee that the corresponding vulnerabilities have been fixed by the vendors completely. It needs to answer whether the patch missing issue is a widely existing security practice mistake. The further question is why vendors missed some patches provided by Google, say deliberately or technical issues.

- **Q3** ***How many pre-installed apps containing unfixed vulnerabilities (CVEs)?*** In addition to the OS patch missing issue mentioned by Q2, pre-installed apps also may have known and unfixed vulnerabilities. That is, the ROM images contain vulnerable apps with CVEs, which attackers will quickly exploit. Apart from the detection statistics, we also want to explore the sources of these vulnerable pre-installed apps and the actual scope of influence.

- **Q4** ***Whether the "non-vulnerable" pre-installed apps are secure?*** The security risks of pre-installed apps are not limited to CVEs. Many common configuration mistakes can result in severe security issues, like exported components and cryptographic misuse. We plan to measure the overall security situations of pre-installed apps from multiple levels and explore whether these configuration mistakes can result in practical security threats.

To answer the proposed questions and explore the reasons behind them, we built a large-scale firmware dataset, covering 6,261 Android firmware images from 153 phone vendors with 443,475 pre-installed APK files (in all unpacked images). To the best of our knowledge, ***it is the largest Android firmware dataset ever used for security measurements***. Also, apart from the Android officially released 411 CVEs, we collected 191 CVEs associated with pre-installed apps. In particular, we designed an automated analysis framework, named AndScanner, to execute a series of targeted measurements and primarily explorations. After massive data analysis and case studies, finally, we can answer the above research questions and provide new knowledge of the Android firmware ecosystem (with deep understandings) to the security practices in software engineering.In this work, we hope to help consumers learn more about the security knowledge of Android phones, such as the timeliness and efficacy of security updates and whether there are security vulnerabilities in pre-installed applications. Therefore, they can have security-relevant references when purchasing an Android phone.

Contributions. Here we summarize our main contributions.

- **Large-scale datasets.** This study covers 6,261 Android firmware images of 153 phone vendors and 602 related CVEs, which can provide sufficient representative data support.
- **Automated tool.** We designed an automated analysis framework, AndScanner, to analyze ROM images, including ROM crawling, ROM parsing, patch analysis, and app detection.
- **Measurements and explorations.** Based on massive data analysis and case explorations, we answered the proposed questions, and detailed analyses are provided in Section 4. Also, we responsibly notified the relevant vendors of the security issues we discovered. Currently, 38 vulnerabilities from 15 vendors have been confirmed, of which 32 were assigned CVE/CNVD IDs.

To foster the future study, we will make the source code of AndScanner and the experiment data available at https://github.com/chicharitomu14/AndScanner.

2 BACKGROUND AND METHODOLOGY

In this section, we provide the necessary background of Android firmware and discuss the methodology of this work.

2.1 Android Firmware

To Android phones, firmware is the system software that provides low-level control and makes the hardware work. Firmware is usually stored in particular types of memory, called flash ROM (Read Only Memory) [48]. Therefore, we sometimes refer to the firmware of a phone as ROM or ROM image[1]. Typically, the firmware of an Android phone contains a bootloader, Linux kernel, Android runtime framework, radio firmware, and various pre-installed apps [38].

[1] In this paper, we use “ROM images”, “ROM files”, and “firmware” interchangeably.

Phone vendors may publicly provide the ROM images to facilitate the users to flash their devices (factory reset or regular OS update). For example, Google releases the official ROM images of Nexus and Pixel phones periodically [10, 11].

Format and structure. The firmware is usually released by phone vendors in a compression format, such as `.zip`. Sometimes, the ROM files may be encrypted to prevent reverse engineering. After unpacking a ROM file, in general, we can find several common partition images, such as `boot.img`, `vendor.img`, `recovery.img`, and `system.img` [23]. They serve different functions for the device. For example, `boot.img` contains a kernel image and a ramdisk image for loading the device before the filesystem mount. Among them, `system.img` is the most important one, which contains the Android framework. It provides the binary files and configuration files for running the Android OS, such as system apps and libraries. Also, `system.img` belongs to file system images, and it can be Yaffs2 format or sparse image format [15].

Pre-installed apps. The pre-installed apps are the apps installed in the Android firmware. They may be the core system components provided by Android and the phone vendors, such as Package Installer[2] for app installing and Mi Cloud[3] for Xiaomi cloud service. Third-party apps (e.g., Twitter) can also be pre-installed due to commercial cooperation or other reasons. In general, pre-installed system apps are put in the `/system/app` and `/system/privapp` folders. Third-party apps are put in the `/data/app` folder.

Customization. Android is an open-source OS designed for mobile devices, and vendors tend to conduct the secondary development based on the source code of AOSP (Android Open Source Project), that is, adding extra features and optimizations to their mobile products. Due to such a customization process, the system configurations and pre-installed apps become quite diverse in different Android phones (and the corresponding ROM images). According to the previous research [26, 27, 54, 61], ROM customization may bring new security risks due to the carelessness of developers or even malware installed on purpose. Research details and more related work are reviewed in Section 6.

Security update. During system software developments, security vulnerabilities cannot be avoided. On the other hand, the vendors should fix the discovered vulnerabilities and provide security updates for users in time. Following such a policy, Google publishes the details of security vulnerabilities affecting Android devices every month [5]. The partners (phone vendors) will be notified before vulnerability publication. In the ideal case, the vendors should also provide security updates for their phones by every month.

2.2 Methodology

Nearly all previous work only focuses on some specific aspects of the custom firmware, like malware and privacy leakage. There still lacks a deep understanding of its ecosystem and security status, especially the causes behind the current status. To bridge the gap, this work aims to conduct a large-scale and comprehensive security measurement on Android firmware in the wild.

[2]Package name: `com.google.android.packageinstaller`
[3]Package name: `com.miui.cloudservice`

To answer the research questions proposed in Section 1, our measurements and analyses should be based on a large-scale dataset covering diverse ROM images. Also, an automated analysis tool is needed, that is, the design of AndScanner. It takes the ROM images as the input and carries out a series of security analyses on different layers. On the OS level, we focus on the analysis of security patches. To the aspect of pre-installed apps, apart from the actual influence of known vulnerabilities, the discovery of unknown vulnerabilities is also considered, such as attribute misconfiguration and cryptographic misuse. In addition, to quantify the security threats, we also should investigate the fundamental reasons behind the statistical data.

As our work focuses on comprehensive measurements, it will consider as many types of vulnerabilities as possible. We assume the attacker has a stronger ability that can induce the victim to click on the URL, install the app, even physically touch the device.

The main research challenges in this work include 1) confirmation of CVEs related to pre-installed APPs, 2) efficient large-scale security patch detection, and 3) selection of the detection features of the pre-installed app security. Section 3 describes the corresponding solutions to these challenges.

3 DATA COLLECTION & ANALYZER

Since we aim to evaluate the security status of Android firmware ecosystem, a large amount of ROM images are needed. Besides, CVE information is also crawled for security feature checking and correlation analysis. We implemented a crawler to collect the ROM images and the target CVE data. Meanwhile, we implemented several analysis tools to facilitate our large-scale security measurement. This section illustrates the data source and detailed design of AndScanner, which contains the *Crawler*, *ROM Parser*, *Patch Analyzer*, and *App Analyzer*. The overall workflow is illustrated in Figure 1.

3.1 Crawler

ROM images. Most major phone vendors, such as Google and Xiaomi, release their ROM images on their official websites to facilitate users downloading. In some cases, the users need to recover their bricked phones with the factory images. However, some vendors, like Samsung and Huawei, do not directly provide the public downloading URLs for their firmware. The images only can be obtained through some official support channels. In such cases, we try to construct all possible URLs based on a legal downloading URL by changing some key parameters. In addition, we also obtain the data collected by another open-source repository, Android Dumps [2], a public project hosting phone images. This project contains images from small phone vendors, increasing the scale and diversity of our dataset. For the soundness of Android Dumps, we verified the unpacked files of our crawling firmware with those of the identical firmware in Android Dumps to ensure the integrity of firmware images in Android Dumps. In addition, we reproduced the open-source tool [36] of Android dumps and verified the accuracy of the unpacking results.

CVE data. As described previously, the CVE data is related to multiple security feature analysis and correlation analysis. The initial research challenge in this work is to establish a link between CVEs

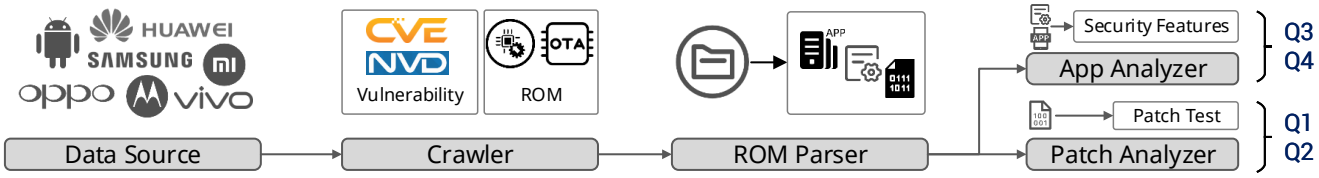

Figure 1: Workflow of AndScanner.

and pre-installed apps. Through traversing all CVE data from 2009 to 2020, we collected the basic data at first. That is, CVE IDs, disclosed time, and descriptions are collected from the official CVE website [9]. The CPE (common platform enumeration) information, like vulnerable app names and versions, are collected from the NVD website [18]. Then, we match the CPE information with the metadata of pre-installed apps (e.g., app name, and version), collected from ROM Parser (Subsection 3.2), to filter out the pre-installed apps related CVEs. However, as the irregular formats of some CVE submissions, we indeed missed some CVEs. Hence, we conduct the filtering again by manually check each CVE and remove the duplicate and irrelevant CVEs.

3.2 ROM Parser

As the critical step before security measurement, accurate ROM image parsing is quite essential. The ROM parser module achieves such a target. That is, in this step, unpacked image files and the corresponding firmware metadata are outputted. In our implementation, two sub-steps are involved, ROM unpacking and data extraction. First, the ROM parser unpacks the ROM image and obtains the individual file-system images for subsequent security analysis. Next, multiple kinds of firmware metadata will be extracted from the unpacked files.

ROM unpacking. As described in Subsection 2.1, firmware is usually released in a standard compression format. It is easy to handle this case. To the encrypted ROM images, we use some dynamic methods, such as ADB (Android Debug Bridge) debugging, to obtain the decryption key from the running system. To the uncompressed Android ROM files, we still need to unpack the partition images, especially `system.img`. If the vendors store the images using the official sparse image format, it can be resolved with the "official" tool, simg2img [31]. The other vendors may use their private formats or compression formats to store the factory images or OTA images. We identify the format of image files based on their file extension and format features first, then use the corresponding public parsing tools [44] [58] [6] [35] [56] to unpack them.

Data extraction. After unpacking the images, we generate the identification for the ROM image and each internal file. We extract the metadata from each ROM image based on their property files (i.e., `build.PROP`) for sub-process analysis, including Fingerprint, Android version, Build time, Security patch level, Model, and Vendor. `Fingerprint` is used as the identifier of a ROM image. `Android version` represents the ROM image's Android OS version. `Build time` is the build time of a ROM image, such as "`Thu Jun 29 18:12:59 UTC 2017`". `Security patch level` represents the time of a device's latest security patch update, such as '`2017-08-05`". `Model` represents the corresponding device model. `Vendor` represents the ROM image's vendor. The internal files consist of two types of files: *pre-installed application packages* (APKs) and *binary executables*. For the internal files, we use a tuple `<File MD5, vendor, ROM fingerprint>` to mark the source of this file.

3.3 Patch Analyzer

Since patches may originate from more than one upstream kernel (e.g., Linux, AOSP, and Qualcomm), the propagation usually takes multiple steps to reach the OEM vendors [59] finally. At the same time, vendor's patch claims may be unreliable [49]. Hence, we need to verify the completeness and effectiveness of the security patches on a ROM sample independently. It is also a critical step to evaluate the ROM file's security status. The Patch Analyzer module generates reports about whether an Android firmware has been patched in time and whether the vulnerabilities indeed have been fixed on the firmware. In summary, this module is designed to answer Q1 & Q2 and explore the fundamental reasons behind them with the *Patch Delay Analysis Module* (for Q1) and the *Patch Missing Analysis Module* (for Q2).

Based on the firmware's metadata collected in Subsection 3.2, the *Patch Delay Analysis Module* used the build time and security patch level to check whether one firmware followed up the released AOSP security patches in time. The value of the build time minus the security patch is defined as the patch delay time. For example, in a ROM image, if the build time is 2019-09-13 (UTC time) and the security patch date is 2019-08-05, so we can calculate the delay as 39 $days/(30\ days/per\ month)$ = 1.3 months (rounded down to 1 month). Based on Google's recommendation [3], we decided 30 days as the ideal time to patch the Android firmware image. If the delay period is less than 30 days, we do not count it and believe it is a reasonable delay for vendors processing the security update.

To address the second research challenge, we should balance Patch Analyzer's efficiency and accuracy. Hence, we implement the *Patch Missing Analysis Module* based on the patch test set and patch verification logic of the state-of-the-art open-source app SnoopSnitch [49] and make it work for the large scale of firmware samples. In detail, the workflow of signature generation for vulnerabilities and patches [49] is as follows: (1) compiling reference source code before patching and after patching, (2) parsing disassembly listing for relocation entries, (3) sanitizing instructions to toss out irrelevant destination addresses of the instruction, (4) generating a hash value of remaining binary code, (5) generating signature containing function length, position/type of relocation entries, and the code's

hash value. SnoopSnitch collects nearly 1,100 source code trees and hundreds of different Android revisions, and re-compiles each source code leveraging different compilers with different optimization levels on all supported CPU types. In total, the patch test set contains 411 test cases related to 411 CVEs and 520 patch files from Oct 2015 to May. 2020 (Android 5.0 - Android 10).

However, the original version of SnoopSnitch is implemented as an Android app and can only run on a single phone, which is not suitable for large-scale analysis. Therefore, we re-implemented the verification logic of SnoopSnitch based on Python. We take verification on each executable file in the ROM images as an independent detection task, send it to the thread pool, and then use multiple processes to test each patch test case for the binary. Compared with the original app, our analysis time for a ROM image has been reduced by ten times. Because we discard patch test cases related to the runtime state, it may result in false positives. In order to verify the correctness of our work in time and correct our results, we take samples for manual verification based on the following two principles: 1) choose from vulnerabilities that are centrally disclosed and identified as fixed in a specific firmware; 2) randomly select the vulnerabilities that have been verified as fixed and unfixed.

In the *Patch Missing Analysis Module*, the test set contains 411 test cases. One test case is a signature generated by the corresponding CVE-related patch files. The *Patch Missing Analysis Module* first selects the patch that affects the firmware according to the firmware's OS version, selects the corresponding patch test cases from the test set, and filters the patch-related binary files from the unpacked files of the firmware. Then, the patch-related binary files' signatures will be generated in the same way as the test cases. Then these signatures will be compared with the corresponding test cases to determine whether these patches exist. Finally, if one patch does not exist, we will combine the security patch level of the ROM image and the release time of this patch to determine the patch missing. For example, if the patch was released in Jan 2019 and the ROM image's security patch level is Feb 2019, it will be patch missing. If the ROM image's security patch level is Dec 2018, it will not be patch missing.

3.4 App Analyzer

To answer Q3 and Q4, App Analyzer, a static analysis tool implemented based on Androguard [28] and CryptoGuard [34], is designed to extract the identification features and vulnerability features of pre-installed apps, including the *CVE Matching Module* (for Q3) and the *Risk Detection Module* (for Q4).

The *CVE Matching Module* extracted the identification features from pre-installed apps firstly, including the package name, version, signature and file MD5. Then, the module matched the apps and CVEs based on the identification features, and confirmed whether the known vulnerability in the matching app by checking the related vulnerability code and misconfigured attributes. Next, the module combined the tuple <`File MD5`, `vendor`, `ROM fingerprint`> (mentioned in Subsection 3.2) to establish each CVE's impact tuple <`CVE Num`, `package name`, `version`, `signature`, `File MD5`, `vendor`, `ROM fingerprint`>. Finally, after comparing the CVEs' disclosure time and the build time of ROM images (with vulnerable apps), we can get the images with unfixed app CVEs.

The *Risk Detection Module* focuses on the security risk measurement for the pre-installed apps. To address the third research challenge, we need to select suitable features to detect the security of the pre-installed apps. By analyzing the collected 191 CVE vulnerabilities (from 2013 to 2020) related to the pre-installed apps, we find that more than 90% (176/191) are caused by *attribute misconfiguration* (168) and *cryptographic misuse* (8) issues[4]. As the proportion of other vulnerabilities is too low, we selected the two with the highest proportion, *attribute misconfiguration* and *cryptographic misuse*, as the analysis targets. Meanwhile, all these problems have the corresponding detection items in OWASP Mobile Security Testing Guide [50].

3.4.1 Attribute Misconfiguration. The extracted security features, stored in the AndroidManifest.xml file[5], are chosen based on the CVE records related to Android app's configurations and Google's official security enhancements, as mentioned above.

F1: exported components. The `exported` attribute of a component (e.g., Activity and Service) determines whether the component is public. If a component is public (set to `true`), it can be exported and accessed by other apps. Developers can use the `permission` attribute to restrict the invocation from external entities. Suppose a component involving essential functions is exported to the external apps without permission protection. In that case, an attacker can easily exploit those essential functions to complete malicious actions through a zero-permission malware installed on the same phone, such as screen/messages recording involved in `CVE-2018-14996`.

F2: debuggable. The `debuggable` attribute can be set to `true` to help developer debug the app during development. If a developer releases an app with the `true` `debuggable` attribute, it can be exploited by attackers to access the debuggable app's data with the privileges of the app, even execute arbitrary code with this app's permissions. This is why when a developer tries to upload a debuggable app on Google Play, Google Play will return an error message: "*You uploaded a debuggable APK. For security reasons, you need to disable debugging before it can be published in Google Play* [22]."

F3: networkSecurityConfig. The `networkSecurityConfig` attribute allows an app to customize its network security settings in a safe, declarative configuration file without modifying its source code, which is available since Android 7.0. If the `networkSecurityConfig` feature is not set, this app will be at risk from a man-in-the-middle attack [17, 30]. Google Play prohibited new apps and updates from including insecure certificate validation logic [12–14], like the `networkSecurityConfig` misconfiguration.

F4: allowBackup. The `allowBackup` feature defines whether an app's data can be backed up and restored by a user who has enabled the USB debugging mode. If this feature is set to `true`, it means an attacker can take the backup of the app's data (e.g., account information and passwords) no matter whether the device is rooted, like `CVE-2017-16835`.

[4]The other vulnerabilities are SQL injection and so on, related to the user input parser and validation.

[5]Every app project must have an AndroidManifest.xml file (with precisely that name) at the root of the project source set. The manifest file describes essential app information. The manifest file is required to declare the following: package name, used components, required permissions, and required hardware and software features.

Table 1: Cryptographic Misuse detection rules.

Detection Rules
(1) Predictable/constant cryptographic keys
(2) Predictable/constant passwords for password-based encryption
(3) Predictable/constant passwords for Key Store
(4) Custom Hostname verifiers to accept all hosts
(5) Custom Trust Manager to trust all certificates
(6) Custom SSLSocketFactory w/o manual Hostname verification
(7) Occasional use of HTTP
(8) Insecure hash algorithms (e.g., SHA1, MD4, and MD5)

F5: sharedUserId. The `sharedUserId` attribute allows two or more apps with the same attribute value and the same certificate to access each other's data and run in the same process. Suppose a pre-installed app sets this attribute as `android.uid.system`, and the app has an exported component that can be interacted with any third-party app. In that case, an attacker can conduct some privileged system operations without root, through his own third-party app [53], like `CVE-2016-10136`.

3.4.2 Cryptographic Misuse. Cryptography plays an essential role in securing the user data for the apps. The misuse of Cryptographic techniques will bring severe security threats. For example, in the case of `CVE-2020-5667`, the `Studyplus` app (ver. 6.3.7 and earlier versions) uses a hard-coded API key for an external service, which can be obtained through reversing analysis of the app. As a part of our measurement, we also need to detect the cryptographic misuse issues in the pre-installed apps through the analysis module based on CryptoGuard [34], a program analysis tool to find cryptographic misuse on Android. Considering of attacker's gain and attack difficulty, we chose the high-risk rules that are defined in the CryptoGuard paper [52] as our detection rules. The details of the rules are listed in Table 1.

4 MEASUREMENTS AND FINDINGS

In this section, we present and discuss our measurement results. Particularly, we answer the research questions proposed in Section 1 with deep cause explorations.

4.1 Experiment Setup

Our measurement programs are mainly implemented in Python and shell scripts with 12,278 lines of code. Three powerful servers were deployed for our large-scale measurement and data storage.

Dataset. Our dataset is built from August 2020 to November 2020. As listed in Table 2, in total, we collected 6,261 Android firmware images of 153 Android vendors, covering all mainstream Android phone vendors and some small vendors like Gionee [42] and Leagoo [47]. These images occupied 8.6 TB storage. To the best of our knowledge, *it is the largest Android firmware dataset ever used for security measurement.* The image types of our collected dataset are diverse, like .zip and .ozip. The Android OS versions cover 2.x to 11, and the building time is from 2012 to 2020. All of them could be unpacked successfully. Besides, we obtained 443,475 unique pre-installed APK files from all unpacked ROM images, and 191 CVEs related to these pre-installed apps.

Accuracy. In addition, to verify the accuracy of SnoopSnitch, we randomly selected 50 patch missing analysis results to do the manual verification. Only one result is suspected to be false positive.

4.2 Analysis Results

Following the workflow described in Section 3, we conducted a series of security analyses to answer the research questions proposed in Section 1. Here we give our answers with the corresponding measurement data and explore the reasons behind them.

Q1. ***Do the Android phone vendors follow up the released AOSP security patches in time?***

Overall result. ***Up to 1,516 of the total 6,261 images (24.2%) have at least one month delay***. The overall average delay period of 6,261 images is around 0.8 months. To the firmware existing the patch delay issue, the average delay period is around 3.2 months. The most severe delay is more than three years. That is, the latest image of OPPO A33m (put on sale in 2015) was built in Jan 2019 with Android version number 5.1.1. However, its security patch date was still labeled as Oct 2015, say 40 months' delay.

In Table 2, we list the overall patch delay statistics of the top 10 vendors with the number of ROM images in our dataset. Based on this table, we could summarize that this security patch delay issue is widespread, even for well-known vendors, like Samsung and Huawei. Also, some vendors' performance is entirely terrible in this issue, which indirectly reflects their insufficient security efforts. For example, as one of the top 3 Android phone vendors [46], 46.8% of the ROM images of Xiaomi have patch delay issues. Only one image (total of 110 images) from Nokia has a one-month patch delay, and its security effort is still admirable. In this table, no image from Google has delay issues. It is entirely reasonable because the Android security updates are maintained and published by Google.

Cause exploration. To further understanding the patch delay issue, taking Xiaomi as an example, we compare the delay situations among different models. As listed in Table 3, we can find no patch delay in the models released after 2017, but the earlier models have a relatively bad performance. One possible reason is that as experiencing the trough in 2016, Xiaomi devoted more resources to the development of mobile devices from 2017. Another possible reason is the introduction of Project Treble, as the following analysis.

Google also notices the issue of Android fragmentation and update delay. In Android 8.0, Google re-architected the OS framework (Project Treble) to split the OS framework and vendor implementation to help the vendors update their devices [25]. In our dataset, there are 4,838 images based on Android 8.0 or later versions. Among them, 782 images exist the delay issue, say 16.2%. The average delay is 2.8 months. On the other hand, to the images before Android 8.0, around 51.6% have delay issues, and the average delay is 4.0 months. It means the Project Treble is successful to some extent though we cannot quantify its contribution precisely.

Furthermore, Google launched two certification programs to regulate and promote Android OS, *certified partners* [21] for phone vendors and *supported devices* [24] for individual phone models.

Table 2: ROM images information and patch delay statistics of vendors.

Vendor	Country	Partner	Amount	Versions	Build Time	Pre-installed Apps	Delay Images	Proportion	Delayed Months
Google	US	Official	1,211	5.1.1-10	201509 - 202003	10,444	0	0%	0
Huawei	China	✓	978	4.4.2-11	201612 - 202009	12,029	29	2.9%	0.099
OPPO	China	✓	826	2.3.4-11	201204 - 202010	24,688	307	37.2%	1.109
Samsung	South Korea	✓	707	4.1.2-11	201307 - 202009	12,356	165	23.3%	0.950
Xiaomi	China	✓	524	4.1.1-10	201504 - 202010	8,077	245	46.8%	2.229
Motorola	US/China	✓	397	5.1-10	201602 - 202010	5,412	80	20.2%	0.426
OnePlus	China	✓	173	5.1.1-11	201603 - 202011	3,084	69	39.9%	0.497
Vivo	China	✓	139	4.2.1-11	201307 - 202009	6,618	96	69.1%	1.755
Nokia	Finland	✓	110	8.0.0-10	201711 - 202011	4,751	1	0.9%	0.009
Realme	China	✓	101	8.1.0-10	201901 - 202011	1,979	13	12.9%	0.158
Others (143)	-	-	1,095	-	-	163,658	-	-	-
Total	-	-	6,261	-	-	253,096	-	-	-

Table 3: Examples of Xiaomi phones with patch delay.

Models	Android Version	Delayed Months	Release Date
Xiaomi 10	10.0	0	Feb, 2020
Xiaomi 9	9.0	0	Feb, 2019
Xiaomi 8	8.1	0	May, 2018
Xiaomi 6	7.1.1	3	Apr, 2017
Xiaomi 5s	6.0	12	Oct, 2016
Xiaomi 4	4.4	16	Aug, 2014

Table 4: Patch delay and patch missing between partner vs. non-partner & supported devices vs. non-supported ones.

	Delay Months	Missing Patches
Partner Vendors	0.83	0.97
Non-Partner Vendors	1.37	1.42
Supported Devices	5.18	4.04
Non-Supported Devices	10.96	11.09

For the certified partner, Google certifies the involved vendors to ensure their released devices are secure and ready to run Google and the Play Store apps. There are 145 manufacturers in the certified partners list [7]. Only the Play Protect certified devices are recognized as supported devices with two compulsive requirements. One is including Google apps, and the other is passing the Android compatibility testing [1], including Compatibility Test Suite (CTS), Vendor Test Suite (VTS), and Security Test Suite (STS). As on Sep 3, 2021, there are 33,664 supported devices in total [16]. As shown in Table 4, the average patch delay of non-partner vendors images is around 1.37 months, which is more than 1.5 times of partner vendors (0.83 months). Similarly, the average patch delay of non-supported devices is up to 10.96 months, more than twice supported devices. We can conclude that the mobile phones of certified partners and supported devices are much more reliable in most cases, as they need to pass a series of Google security tests (especially the patch update related tests) for certification.

In addition, we conducted a longitudinal analysis on specific phone models as showcases. As plotted in Figure 2 (a), we selected two phone models with similar release times, maintenance periods,

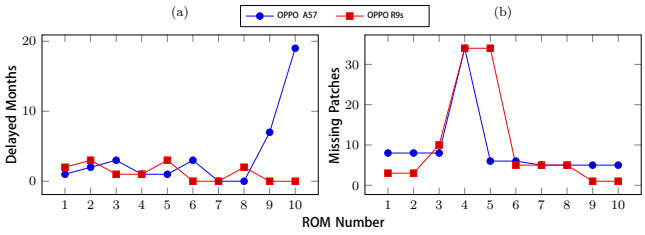

Figure 2: Patch delay and missing trends of OPPO R9s and OPPO A57.

and significant price differences from the same vendor. The high-end model is OPPO R9s (put on sale in Oct 2016), and the low-end model is OPPO A57 (put on sale in Nov 2016). For a device, we arrange its associated ROM images in ascending order of compilation time and observe the changes in patch delay. We can find no remarkable distinction in the first eight images of these two models from this line chart, around 0 to 3 months. However, starting from the ninth ROM image, the difference becomes quite apparent. The patch delay of A57 increases to 7 months and finally reaches the peak of 19 months. On the contrary, there is no patch delay for R9s furthermore. Although A57 was released one month later than R9s, R9s was upgraded to Android 7.1.1 in the ninth ROM image, while A57 was still running Android 6.0.1 until the end. It reflects that OPPO cares more about the maintenance of high-end models. More seriously, the update logs of A57's last two ROM images mentioned that the system security patches were integrated [19]. However, the fact is that their system security patch levels were not changed.

Assessment: The security patch delay is widespread in Android phones, even for leading vendors, like Samsung and Huawei. 24.2% of the images have this issue, and the average delay is about 3.2 months. It means that the updated images of many vendors focus on the feature update, not the security update.

Q2. ***Whether the patching claims of phone vendors are reliable?***

Overall result. ***There are 383 of the total 6,261 ROM images (6.1%) from 50 vendors that have at least one patch missing.***

Table 5: Statistics of images with patch missing.

(Critical/High/Moderate/Low for the rating of the missed patches, and STS-/STS+ for the average number of missed patches before/after STS launching.)

Vendors	# of Images	Proportion	Critical (%)	High (%)	Moderate (%)	Low (%)	Missed Patches	STS-	STS+
Google	51	4.2%	0	0	100.0	0	0.042	0.457	0
Huawei	2	0.2%	0	40.0	60.0	0	0.005	1.000	0.190
OPPO	161	19.5%	25.4	35.8	38.8	0	1.469	6.170	1.168
Samsung	4	0.6%	0	33.3	66.7	0	0.013	1.000	0.056
Xiaomi	19	3.6%	5.4	37.3	57.3	0	0.199	12.600	0.143
Motorola	18	4.5%	2.6	17.9	76.9	2.6	0.098	1.481	0
OnePlus	5	2.9%	0	37.5	62.5	0	0.264	11.500	0
Vivo	4	2.9%	0	61.5	38.5	0	0.078	2.500	0.113
Nokia	0	0%	N/A	N/A	N/A	N/A	0	0	0
Realme	0	0%	N/A	N/A	N/A	N/A	0	N/A*	0

Remarks: ∗ Realme have not gone into the market before August 2018.

Table 6: Statistics of images containing vulnerable apps.

Vendors	Total Images	Images w/ CVE Apps	CVE Apps
Google	1211	0	0
Huawei	978	0	0
OPPO	826	53	85
Samsung	707	3	3
Xiaomi	524	7	11
Motorola	397	30	30
OnePlus	173	0	0
Vivo	139	12	12
Nokia	110	15	19
Realme	101	89	129

The most extreme one has 46 patches missing, which belongs to OPPO R9skt (announced in 2016). This ROM image was built in May 2017, and its security patch level was in Apr 2017. All its missing patches were disclosed from Dec 2016 to Apr 2017.

Table 5 lists the overall patch missing statistics of the top 10 vendors with the number of ROM images. According to this table, we can find that the issue of security patch missing is quite severe, which can also indirectly reflect the involved vendors' insufficient security efforts. For example, as a mainstream vendor, 161 ROM images (19.5%) of OPPO have patch missing issues, and the average missing number is around 1.5. Nokia still has the best performance in patch missing with no patch missing issue. The other vendor with no patch missing issue is Realme, a subsidiary of OPPO.

To better understand the patch missing issue, we took OPPO A59 (announced in 2016) as an example to compare the missing issues among different ROM images. We found that the number of missing patches increases with the ROM image upgrade. The oldest ROM image built in 2016[6] only has one missing patch, but the newest ROM image built in 2018[7] has 23 missing patches. We manually confirmed the effectiveness of contained high-risk vulnerabilities, like `CVE-2017-0479` and `CVE-2017-0480`. The reason may be that the developer gave up maintaining the security patches of this mid-range model, only updated the date of the security patch level.

[6]Fingerprint: `OPPO/A59m/A59:5.1/LMY47I/1449641681:user/release-keys`
[7]Fingerprint: `OPPO/A59/A59:5.1/LMY47I/1519786508:user/release-keys`

Cause exploration. In Table 5, there are 51 ROM images from Google having patch missing issues, which is beyond our expectation, as all patches in the test set are collected from Android security bulletins [5]. After further exploration, we noticed these ROM images belong to 4 models, Nexus 6 (2014), Nexus 5X (2015), Nexus 6P (2015), and Pixel (2016), of which OS versions are all Android 7.x. Also, all missing issues are attributed to one patch, `CVE-2017-0494`, an information disclosure vulnerability in AOSP Messaging and rated as moderate. In addition to the passive oblivion of developers, there is another reason that Google usually fixes moderate vulnerabilities during major version upgrading.

Google introduced Security Test Suite (STS) into Compatibility Test Suite (CTS) in Aug 2018 to test whether the security patches work [8]. We measured the average number of missing patches in the top 10 vendors' images before and after Aug 2018, which is listed in the last two columns of Table 5. To all top 10 vendors, the average number of missing patches after Aug 2018 is much smaller than the number before Aug 2018, proving the effectiveness of STS.

Table 4 also compares the differences of missing patches among vendors (partner vs. non-partner) and devices (supported devices vs. non-supported ones). On average, the images from non-partner vendors miss 1.42 patches, which is higher than the number of partner vendors (0.97). Similarly, the average missing number of supported devices is 4.04, much less than non-supported ones (11.09). Therefore, from the perspective of patch missing, the mobile phones of certified partners and supported devices are also more secure.

Besides, we also measured the longitudinal changes on the missed patches on OPPO R9s and OPPO A57, and their patch missing trends are plotted in Figure 2 (b). The missing patch number of R9s increases markedly in the fourth ROM image, from 10 to 34. A similar situation also appears on A57, from 8 to 34. Our further investigation shows that the missed patches in the previous ROM images have been fixed in the fourth one. However, the fourth image missed 34 new patches that are mainly related to `Mediaserver` (20 related patches) and `Audioserver` (5 related patches). We analyzed the update logs of both devices [19, 20] and found that the fourth ROM image mainly updated `Mediaserver` and `Audioserver` related features. Therefore, it is possible that OPPO temporarily ignores the relevant security patches due to compatibility issues, not updating features, and patching security bugs at the same time.

Furthermore, in the chart, it can be seen that OPPO repaired the previously missed patches in the subsequent ROM images. But the difference between the two devices appears in the last two images. The missed number of A57 maintains at 5, while the number of R9s decreases to 1. The reason is that R9s was upgraded to Android 7.1.1 from 6.0.1, while A57 was not. Due to the upgrade of the OS version, the images of R9s were directly built on the new base Android OS without patching missing. What is more, the last missed patch of R9s only affected Pixel devices, not all Android devices. This longitudinal analysis also indicates that a vendor has different attitudes towards mobile phones' security at different price levels.

Assessment: The claims of vendor patching may not always be reliable, even for some mainstream vendors, such as OPPO. 6.1% ROM images from 50 vendors have security patch missing issues. The possible reason is that vendors do not have sufficient testing time or cannot bear the maintenance costs.

Q3. ***How many pre-installed apps containing unfixed vulnerabilities (CVEs)?***

Overall result. ***There are 462 (7.4%) ROM images from 71 (46.4%) vendors containing the unfixed app CVEs***. Table 6 lists the statistics of images containing unfixed vulnerable pre-installed apps by vendors. In this table, Google, Huawei, and OnePlus have the best performance with no ROM images containing unfixed vulnerabilities. On the other hand, Realme and OPPO are the only two vendors with more than 50 ROM images containing unfixed vulnerabilities. Especially, Realme has the most (89) vulnerable ROM images, and on average, each ROM image has more than one unfixed vulnerable pre-installed app. The reason is that Realme and OPPO have a unique, vulnerable pre-installed app, named DropboxChmodService, appearing in all ROM images containing unfixed vulnerabilities of these two vendors.

Case studies. Vulnerable apps with unfixed vulnerabilities bring severe security threats. Considering the newness and popularity of the device, we took Nokia 6.2 as an example, its firmware[8] pre-installed two apps with known vulnerabilities: (1) `com.google.android.tag` (ver. 1.1) (`CVE-2019-9295`, disclosed on Aug 21, 2019) and (2) `com.qualcomm.qti.callenhancement` (ver. 9) (`CVE-2019-15473`, disclosed on Nov 14, 2019). Specifically, `CVE-2019-9295` is a vulnerability related to the NFC function, leading to local privilege escalation. `CVE-2019-15473` is a vulnerability caused by an exported service (i.e., `CallEnhancementService`). As `com.google.android.tag` is a Google official pre-installed app integrated into the AOSP source code, and `com.qualcomm.qti.callenhancement` is an essential service pre-installed app provided by Qualcomm. It means that some vendors do not pay enough attention to security checking when pre-loading suppliers' apps, which may be due to excessive trust in the security of these well-known suppliers.

Scope of influence. Due to commercial cooperation and app supplier reasons, phones are usually pre-loaded many apps. On the other hand, a pre-installed app can be loaded by dozens of vendors. Therefore, one vulnerable pre-installed apps may affect dozens of vendors. Considering the limitations of the vulnerability reporter's device, the actual scope of influence of such vulnerabilities may be more than the disclosed scope.

According to the app's scope of influence, we showed seven typical apps in Table 7. We can find that the CVE information disclosed publicly is just the tip of the iceberg. Apart from the official vulnerable pre-installed app `com.google.android.tag`, every vulnerable pre-installed app is associated with undisclosed vendors. The two pre-installed apps associated with the most number (15) of undisclosed vendors is `com.qualcomm.qti.callenhancement` (ver. 9) and `com.qualcomm.qti.callenhancement` (ver. 8.1.0). Since they are provided by Qualcomm, the major chip supplier for Android devices, all devices equipped with Qualcomm chips will be affected. This situation also applies to `com.mediatek.factorymode` (ver. 1.0) with 10 undisclosed vendors, a pre-installed app related to MediaTek, another major chip supplier for Android devices. The vulnerable pre-installed app `com.dropboxchmod` (ver. 1.0), related to the vendor customized basic service in OPPO and Realme devices, is associated with the most (328) ROM images. Therefore, the vulnerable apps from basic infrastructure services and hardware suppliers will affect various devices.

In addition, we noticed that 5 vulnerable pre-installed apps (i.e., `com.dropboxchmod` (ver. 1.0, disclosed in 2018), `com.qualcomm.qti.callenhancement` (ver. 9, disclosed in 2019), `com.log.logservice` (ver. 1.0, disclosed in 2019), `com.qualcomm.qti.callenhancement` (ver. 8.1.0, disclosed in 2019), and `com.mediatek.factorymode` (ver. 1.0, disclosed in 2019) – appeared in the ROM images built in 2020, which means those ROM images still load the vulnerable pre-installed apps at least one year after the vulnerabilities were disclosed. All these apps are related to the basic system services, and three of them are provided by chip suppliers.

Assessment: Apps with unfixed CVEs appear in almost half of vendors' ROM images, even some famous vendors, like OPPO and Samsung. The most extreme vendor is Realme, the average number of unfixed vulnerabilities in each ROM image is more than 1. A vulnerable pre-installed app can affect hundreds of ROM images from dozens of vendors. The affected vendors may be excessive trust in the code security of well-known suppliers.

Q4. ***Whether the "non-vulnerable" pre-installed apps are secure?***

Overall result. ***3595 (57.4%) ROM images from all 153 vendors contain potentially vulnerable pre-installed apps***. Table 8 lists the average number of potential security risks in each ROM image by vendors. We can find that Realme has the most potential security risks in each ROM image, and Huawei has the least number. There may be two main reasons for this situation. One is that Realme has the most average number (485) of pre-installed apps in each ROM image. The more the base, the more the potentially vulnerable quantity. The other reason is the third-party pre-installed apps. We found that 83.67% of potentially vulnerable pre-installed apps in Realme come from third-party suppliers, which is also the highest proportion among the top 10 vendors.

It is worth noting that there are 117 potentially vulnerable apps in Google's ROM images on average. As ROM images of all other

[8] Fingerprint: `Nokia/Starlord_00WW/SLD_sprout:9/PKQ1.190118.001/00WW_1_160:user/release-keys`, built in January 2020.

Table 7: Scope of influence of vulnerable pre-installed apps.

(DAVs stand for disclosed affected vendors, and UAVs stand for undisclosed affected vendors.)

Package Name	App Version	Affected ROMs	CVEs	DAVs	UAVs	ROM Versions	Build Time
com.google.android.tag	1.1	840	1	N/A*	60	4.0.4 - 9.0	201203 - 202009
com.dropboxchmod	1.0	328	1	1	1	4.4.4 - 10.0	201601 - 202011
com.qualcomm.qti.callenhancement	9	251	4	1	15	9.0 - 10.0	201804 - 202010
com.log.logservice	1.0	108	2	1	1	7.1.1 - 10.0	201712 - 202005
com.qualcomm.qti.callenhancement	8.1.0	90	3	1	15	8.1.0 - 10.0	201804 - 202002
com.oppo.engineermode	V1.01	76	1	1	1	6.0 - 8.1.0	201611 - 201904
com.mediatek.factorymode	1.0	16	6	4	9	7.0 - 9.1	201803 - 202007

Remarks: ∗ The vulnerability disclosure information does not mention affected vendors.

Table 8: Detection results of potentially vulnerable pre-installed apps. (F1-F5 are defined in Subsection 3.4.)

Vendor	F1	F2	F3	F4	F5	Crypto	Total
Google	34	0	15	35	13	19	116
Huawei	26	0	7	27	16	5	81
OPPO	68	1	13	81	47	23	233
Samsung	31	0	20	27	23	17	118
Xiaomi	104	0	66	118	62	58	408
Motorola	95	0	53	103	46	42	339
OnePlus	93	0	49	120	50	39	351
Vivo	64	0	28	60	37	24	213
Nokia	69	0	46	90	37	28	270
Realme	130	0	83	128	94	70	505

vendors are customized based on AOSP, the potentially vulnerable apps in AOSP will also be pre-loaded in these ROM images. If one of the results in practical security issues, it will have a large scope of influence, like the app `Tag` (`CVE-2019-9295`).

Moreover, we analyzed the sources (the original developers) of the potentially vulnerable pre-installed apps based on their signature information. Most of these apps come from the phone vendors occupying 71.5% (87,825/122,770), and 19,330 (15.7%) potentially vulnerable pre-installed apps coming from third-party developers.

In addition, we did the exploitability verification on these potentially vulnerable pre-installed apps. First, we confirmed which devices contain potentially vulnerable pre-installed apps in our own devices. Then, we used the vulnerability scanning tool based on drozer [40] to detect the target APP to discover the exploitable vulnerabilities. For the exploitable vulnerabilities, we constructed and ran the corresponding PoCs so that we could provide evidence of exploitable vulnerabilities to vendors. Finally, we confirmed and submitted 307 exploitable vulnerabilities based on our own devices. 38 pre-installed app vulnerabilities from 15 vendors have been confirmed with 32 CVE/CNVD numbers.

Case studies. Here we give some interesting cases.

Case 1. *The security risks of "ancient times" are still alive.*

The app `com.autonavi.manu.widget` (ver. 2.4), signed by AutoNavi Software Co., set the `debuggable` value as `true`. This app is integrated into the ROM image[9]. The attackers can exploit the exposed `debuggable` functionality to access the app data and even execute arbitrary code. There are two possible reasons: (1) the developers forgot to check the `debuggable` value, (2) the supplier provided the test version of the app by accident.

Case 2. *Crypto misuses bring practical data leakage risks.*

The app `com.samsung.android.smartcallprovider` (ver. 10.0.24) signed by Samsung, used a hard-coded stringas the key for the deployed AES encryption. Since AES is a symmetric encryption algorithm, the attacker can utilize this key to decrypt the ciphertext, making the encryption operations meaningless. The developers may not have enough knowledge to invoke cryptographic APIs correctly [37]. Also, the manufacturer neglected to check on the hard-coded key existing in the pre-installed apps.

Case 3. *Exported components result in user privacy leakage.*

The app `com.vivo.weather.provider` (ver. 6.0.2.2) is a weather app developed by Vivo. It has an exported content provider, `com.vivo.weather.provider.WeatherProvider`. The attacker can steal the victim's privacy from the content URIs through the exported component without permission, such as the victim's accurate location information (pinpointing the street)[10], the cities visited in recent 100 days[11], and inferring the victim's trajectory information[12]. Based on our database, this vulnerability exists in all 24 versions of this app (ver. 1.0 to 6.0.2.2), which affects 109 ROM images of Vivo (OS ver. 4.4.2 to 11), build time from Jun 2016 to Sep 2020. This vulnerability has been confirmed by Vivo and assigned a CVE number, CVE-2021-26279.

Case 4. *Different vendors have different reactions.*

The Google app `com.android.providers.settings`(ver. 10) has a vulnerability that allows an attacker to steal the victim's privacy through a no-permission third-party app. We have reported this vulnerability to nine related vendors, of which four vendors confirmed it, four vendors ignored it, and one vendor (Google) still reviews it for nearly three months. In the confirmed vendors, one vendor (Huawei) identified this vulnerbility as high-risk, two vendors (OPPO, Vivo) identified this vulnerability as medium-risk, and one vendor (Lenovo) identified it as low-risk. Finally, two vendors assigned CVE numbers to this vulnerability, CVE-2021-22486 (Huawei) and CVE-2021-26281 (Vivo). In the ignored vendors, three

[9]Fingerprint: `OnePlus/OnePlus5T/OnePlus5T:9/PKQ1.180716.001/2002171214:user/release-keys` of OnePlus, built on Feb 17, 2020.

[10]Through `content://com.vivo.weather.provider/localweather`.

[11]Through `content://com.vivo.weather.provider/usualcity`.

[12]Through `content://com.vivo.weather.provider/alert`.

vendors (Samsung, ZTE, Oneplus) reasoned that the app was a pre-installed app of Google, and one vendor (Xiaomi) reasoned that the leaked information was insensitive. These differences are mainly due to the different customized development of the Android system by various vendors and various vendors' different vulnerability review standards.

Assessment: All vendors have potential security risks in the pre-installed apps in their ROM images. It may lead to multiple kinds of vulnerabilities. The vendors need to do a comprehensive app security test before their devices are released to markets, especially for their customized apps and third-party apps.

5 THREAT TO VALIDITY

Here we discuss several threats to the effectiveness of this work.

CVE data. Since CVE reports were submitted without standard formats, some of them may lack important information. For example, if the CVE reports do not provide the vulnerable apps' affected versions, we cannot match the corresponding pre-installed apps. This issue may be solved by collecting more vulnerability data from broader sources, like research papers, technical reports, and blogs.

Packed App. As APP Analyzer is built on static analysis tools, it cannot analyze packed apps well. Nevertheless, we found that only 0.23% (1,020/443,475) of the pre-installed apps are packed, which has minimal impact on our measurement results.

6 RELATED WORK

In this section, we review the related work on the security analysis of Android firmware. Most of the previous research focused on a particular aspect of firmware, and few works tried to launched comprehensive large-scale security measurements.

The closest works are the studies of Zheng et al. [60] and Gamba et al. [41]. Zheng et al. [60] designed and implemented DroidRay to evaluate the security of Android firmware images on both the app level and system level, and 250 firmware images were covered. However, their analysis concentrated on pre-installed malware, and the vulnerabilities they analyzed are too old to exploit in the current scenario. Gamba et al. [41] carried out a large-scale study on the ecosystem of pre-installed Android apps and the potential impact on consumers. This study covers several aspects of pre-installed apps, like the stakeholders involved in the supply chain, personally identifiable information leakage, potentially harmful behaviors. However, their analysis target only concentrated on the malicious pre-installed apps and the related privacy operations, without the firmware level security analysis that includes system-level and pre-installed app level. Compared with the above work, this paper presents a comprehensive security measurement on the Android firmware ecosystem, giving a panoramic view and the causes behind the security threats, not just the particular aspects. Meanwhile, we used a more extensive and diverse firmware dataset for analysis.

During the firmware customization process, new security hazards may be introduced. For example, Zhou et al. [61] studied the security risks introduced by the driver customization, especially the downgrade of the Linux protection level of a customized device-related file. Aafer et al. [27] detected the security configuration changes introduced by Android customization, such as permission mismatch and duplicate components declaration. On other aspects of vendor customization, Tian et al. [54] and Aafer et al. [26] focused on the security of AT commands and hanging attribute references, respectively. Possemato et al. [51] studied the Android OEM compliance and security posture during the customization process.

The issue of capability leakage in firmware is also noticed by security researchers. As an early study, Grace et al. [43] designed a tool named Woodpecker to uncover the permission leaks in Android firmware. Nevertheless, their experiment scale was small, and only eight images were analyzed. Wu et al. [57] studied the provenance, permission usage, and vulnerability distribution of pre-installed apps. Similarly, Cam et al. [32] proposed a system, uitXROM, to detect sensitive data leakage in Android firmware by analyzing relationships of pre-installed apps. More recently, Elsabagh et al. [38] presented FirmScope, an automated analysis system to uncover different types of vulnerabilities in pre-installed apps, and over two thousand firmware images were evaluated.

Besides, to the aspect of security patches, Zhang et al. [59] investigated the Android kernel patch ecosystem, revealing the relationship among different parties as well as the bottleneck in patch propagation. The study of Farhang et al. [39] also measured the patch latency from Qualcomm and Linux repositories to AOSP.

7 CONCLUSION

To obtain a panoramic view and deep understandings of the Android firmware ecosystem security, this work presents large-scale security measurements based on a record firmware dataset of 6,261 ROM images from 153 vendors. For achieving such measurements, we designed and implemented an automated analysis framework named AndScanner, containing the Crawler, ROM Parser, Patch Analyzer, and App Analyze modules. As demonstrated in this work, all the users know about their devices' security can be just the tip of the iceberg. Many new findings and the reasons behind them are discovered through our measurements.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their insightful comments. This work was partially supported by Joint Funds of the National Natural Science Foundation of China (Grant No. U1836113), National Natural Science Foundation of China (Grant No. 62002203, 92064008, and 61902148), Shandong Provincial Natural Science Foundation (Grant No. ZR2020MF055, ZR2021LZH007, ZR2020LZH002, and ZR2020QF045), and Beijing Nova Program of Science and Technology (Grant No. Z191100001119131).

REFERENCES

[1] accessed: 2021-09-03. Android Compatibility Program. https://source.android.com/compatibility/overview.
[2] accessed: 2021-09-03. Android Dumps. https://dumps.tadiphone.dev/dumps.
[3] accessed: 2021-09-03. Android Enterprise Security White Paper. https://static.googleusercontent.com/media/www.android.com/zh-us//static/2016/pdfs/enterprise/Android_Enterprise_Security_White_Paper_2019.pdf.
[4] accessed: 2021-09-03. Android Open Source Project. https://source.android.com/.
[5] accessed: 2021-09-03. Android Security Bulletins. https://source.android.google.cn/security/bulletin?hl=en.
[6] accessed: 2021-09-03. brotli. https://github.com/google/brotli.

[7] accessed: 2021-09-03. Certified Partners. https://www.android.com/certified/partners/.

[8] accessed: 2021-09-03. Compatibility Test Suite. https://source.android.com/compatibility/cts.

[9] accessed: 2021-09-03. CVE. https://cve.mitre.org/.

[10] accessed: 2021-09-03. Factory Images for Nexus and Pixel Devices. https://developers.google.com/android/images.

[11] accessed: 2021-09-03. Full OTA Images for Nexus and Pixel Devices. https://developers.google.com/android/ota.

[12] accessed: 2021-09-03. How to address WebView SSL Error Handler alerts in your apps. https://support.google.com/faqs/answer/7071387.

[13] accessed: 2021-09-03. How to fix apps containing an unsafe implementation of TrustManager. https://support.google.com/faqs/answer/6346016.

[14] accessed: 2021-09-03. How to resolve Insecure HostnameVerifier. https://support.google.com/faqs/answer/7188426.

[15] accessed: 2021-09-03. Images. https://source.android.com/devices/bootloader/images.

[16] accessed: 2021-09-03. List of supported Android devices. https://storage.googleapis.com/play_public/supported_devices.html.

[17] accessed: 2021-09-03. Network security configuration. https://developer.android.com/training/articles/security-config.

[18] accessed: 2021-09-03. NVD. https://nvd.nist.gov/.

[19] accessed: 2021-09-03. OPPO A57. https://www.coloros.com/rom/firmware?id=126.

[20] accessed: 2021-09-03. OPPO R9s. https://www.coloros.com/rom/firmware?id=125.

[21] accessed: 2021-09-03. Play Protect Certified Android devices: safe and secure. https://www.android.com/certified/.

[22] accessed: 2021-09-03. Prepare for release. https://developer.android.com/studio/publish/preparing#turn-off-logging-and-debugging.

[23] accessed: 2021-09-03. Standard partitions. https://source.android.com/devices/bootloader/partitions.

[24] accessed: 2021-09-03. Supported devices. https://support.google.com/googleplay/answer/1727131?hl=en.

[25] accessed: 2021-09-03. Treble. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html.

[26] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiao-yong Zhou, Wenliang Du, and Michael Grace. 2015. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), Denver, CO, USA, October 12-16, 2015.*

[27] Yousra Aafer, Xiao Zhang, and Wenliang Du. 2016. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10-12, 2016.*

[28] androguard. accessed: 2021-09-03. Androguard. https://github.com/androguard/androguard.

[29] Android Police Team. accessed: 2021-09-03. Android security update tracker, March 2021: Rankings for popular smartphones. https://www.androidpolice.com/2021/03/03/android-phone-security-update-tracker/.

[30] Cláudio André. 2018. Gmail Android App Insecure Network Security Configuration. https://labs.integrity.pt/articles/Gmail-Android-app-insecure-Network-Security-Configuration/index.html.

[31] anestisb. accessed: 2021-09-03. simg2img. https://github.com/anestisb/android-simg2img.

[32] Nguyen Tan Cam, Van-Hau Pham, and Tuan Nguyen. 2017. Sensitive Data Leakage Detection in Pre-Installed Applications of Custom Android Firmware. In *Proceedings of the 18th IEEE International Conference on Mobile Data Management (MDM), Daejeon, South Korea, May 29 - June 1, 2017.*

[33] Catalin Cimpanu. 2020. Android OEM patch rates have improved, with Nokia and Google leading the charge. https://www.zdnet.com/article/android-oem-patch-rates-have-improved-with-nokia-and-google-leading-the-charge/.

[34] CryptoGuardOSS. accessed: 2021-09-03. CryptoGuard. https://github.com/CryptoGuardOSS/cryptoguard.

[35] cyxx. accessed: 2021-09-03. extract_android_ota_payload. https://github.com/cyxx/extract_android_ota_payload.

[36] Android Dumps. accessed: 2021-09-03. Firmware_extractor. https://github.com/AndroidDumps/Firmware_extractor.

[37] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013.*

[38] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *Proceedings of the 29th USENIX Security Symposium (USENIX-SEC), August 12-14, 2020.*

[39] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. 2019. Hey Google, What Exactly Do Your Security Patches Tell Us? A Large-Scale Empirical Study on Android Patched Vulnerabilities. *CoRR* abs/1905.09352 (2019).

[40] FSecureLABS. accessed: 2021-09-03. Drozer. https://github.com/FSecureLABS/drozer.

[41] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2020. An Analysis of Pre-installed Android Software. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, May 18-21, 2020.*

[42] Gionee. accessed: 2021-09-03. GIONEE. https://gionee.co.in/.

[43] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 5-8, 2012.*

[44] Willem Jan Hengeveld. accessed: 2021-09-03. extfstools. https://github.com/nlitsme/extfstools.

[45] Simon Hill. 2018. What is Android fragmentation, and can Google ever fix it? https://www.digitaltrends.com/mobile/what-is-android-fragmentation-and-can-google-ever-fix-it/.

[46] IDC. 2021. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/vendor.

[47] Leagoo. accessed: 2021-09-03. Leagoo. https://www.leagoo.com/.

[48] Codrut Neagu. 2021. What is firmware? What does firmware do? https://www.digitalcitizen.life/simple-questions-what-firmware-what-does-it-do/.

[49] Karsten Nohl and Jakob Lell. 2018. Mind the Gap: Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis. In *HITB 2018.*

[50] OWASP. accessed: 2021-09-03. OWASP Mobile Security Testing Guide. https://owasp.org/www-project-mobile-security-testing-guide/.

[51] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), Virtual Event, May 23-27, 2021.*

[52] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, November 11-15, 2019.*

[53] Maddie Stone. 2019. Securing the System: A Deep Dive into Reversing Android Pre-Installed Apps. In *BlackHat 2019.*

[54] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Ruales, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. 2018. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *Proceedings of the 27th USENIX Security Symposium (USENIX-SEC), Baltimore, MD, USA, August 15-17, 2018.*

[55] Liam Tung. 2018. Android security: Your phone's patch level says you're up to date, but that may be a lie. https://www.zdnet.com/article/android-security-your-phones-patch-level-says-youre-up-to-date-but-that-may-be-a-lie/.

[56] vicky858. accessed: 2021-09-03. SplitUpdated. https://github.com/vicky858/SplitUpdated.

[57] Lei Wu, Michael C. Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. 2013. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013.*

[58] xpirt. accessed: 2021-09-03. sdat2img. https://github.com/xpirt/sdat2img.

[59] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2021. An Investigation of the Android Kernel Patch Ecosystem. In *Proceedings of the 30th USENIX Security Symposium (USENIX-SEC), Virtual Event, August 11–13, 2021.*

[60] Min Zheng, Mingshen Sun, and John C. S. Lui. 2014. DroidRay: A Security Evaluation System for Customized Android Firmwares. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS), Kyoto, Japan - June 03 - 06, 2014.*

[61] Xiao-yong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (Oakland), Berkeley, CA, USA, May 18-21, 2014.*