Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization

Andrea Possemato*†, Simone Aonzo†, Davide Balzarotti†, Yanick Fratantonio†‡
* IDEMIA
† EURECOM
‡ Cisco Talos

***Abstract*—Nowadays, more than two billions of mobile devices run Android OS. At the core of this success are the open source nature of the Android Open Source Project and vendors' ability to customize the code base and ship it on their own devices. While the possibility of customizations is beneficial to vendors, they can potentially lead to compatibility and security problems. To prevent these problems, Google developed a set of requirements that *must* be satisfied for a vendor to brand its devices as "Android," and recently introduced Project Treble as an effort to partition vendor customizations. These requirements are encoded as part of a textual document (called Compatibility Definition Document, or CDD) and various automated tests. This paper performs the first longitudinal study on Android OEM customizations. We first built a dataset of 2,907 ROMs, spanning across 42 different vendors, and covering Android versions from 1.6 to 9.0 (years 2009–2020). We then developed an analysis framework and pipeline to extract each ROM's customization layers and evaluate it across several metrics. For example, we analyze ROMs to determine whether they are compliant with respect to the various requirements and whether their customizations negatively affect the security posture of the overall device. In the process, we focus on various aspects, ranging from security hardening of binaries, SELinux policies, Android init scripts, and kernel security hardening techniques. Our results are worrisome. We found 579 over 2,907 (~20%) of the ROMs have at least one violation for the CDD related to their Android version — incredibly, 11 of them are branded by Google itself. Some of our findings suggest that vendors often go out of their way to bypass or "comment out" safety nets added by the Android security team. In other cases, we found ROMs that modify init scripts to launch at boot outdated versions (with known CVEs and public POCs) of programs as root and reachable from a remote attacker (e.g., `tcpdump`). This paper shows that Google's efforts are not enough, and we offer several recommendations on how to improve the compliance check pipelines.**

I. Introduction

Mobile devices play a fundamental role in our everyday lives. The vast majority of them, more than two and a half billion worldwide [1], run the Android operating system. A Google-led open source project called *Android Open Source Project* (AOSP) offers both the documentation and the source code needed to build custom variants of the Android operating system (Android OS from now on). These variants are usually called Android ROMs.

However, AOSP does not include all the components required to build a complete system. For instance, Google and AOSP cannot provide kernel device drivers for every hardware configuration. Therefore, third-party vendors (also known as *Original Equipment Manufacturers*, or OEMs) that wish to produce an Android-based device need to properly customize and tweak an AOSP base image according to their needs. These modifications can affect both user-space components, for instance, by including custom applications or services, and kernel-space components, such as kernel drivers.

AOSP's openness and flexibility was a determining factor for the great success of the platform, leading to its adoption by a vast number of vendors, which market devices with various hardware configurations and versions of Android. This resulted in a multitude of different variants, an aspect known as *fragmentation*. The different natures of the different devices can lead to a significant degree of customization (with respect to the baseline AOSP) that, in turn, can have a massive impact on the security of the resulting Android ROM.

In particular, we can identify two classes of security problems. The first is that these customizations may affect the security posture of the overall system (e.g., by making Google's hardening efforts vain), increase the attack surface, and in some cases, even introduce new security vulnerabilities. For instance, a recent study published by Project Zero reports several critical bugs found in such customizations [2].

The second class of problems may originate from the actual components that are affected by the OEM customizations. Indeed, customizations that modify core components of the Android OS may lead to compatibility problems and delays in the application of security patches, such as the ones released as part of the monthly Android security bulletins.

Google, who is leading the Android project, is well aware of these problems, and it has tried to counter them by working in two parallel directions.

The first is *compliance*: while AOSP is an open source project and thus it can be freely modified, an OEM that wishes to brand its devices with the "Android" label (which is a trademark of Google) needs to follow a well-defined set of rules. For example, to be Android-branded, a device needs to meet the requirements presented in the Android *Compatibility Definition Document* (CDD) [3], including any documents incorporated via reference. From a practical standpoint, the CDD is a series of technical and non-technical requirements specified in natural language. Each of the requirements has a label that indicates whether it *must* be adopted, its adoption is *strongly recommended*, or just *recommended*. A new CDD is published for each new version of Android, and, usually, requirements that are indicated as 'strongly recommended' are later marked as 'must' in the next version of the CDD. To simplify the checking for compliance with these requirements, Google also released a Compatibility Test Suite (CTS). While the CTS has the advantage of being fully automated, it only checks for a subset of the requirements specified in the CDD (this is due to the nature of some CDD requirements, which is challenging to express in a programmatic form).

A second Google-led effort to counter the security repercussions introduced by OEM customizations is *Project Treble*, a re-architecture of the Android OS, introduced in 2017 as part of Android 8.0. This reorganization aims to separate the vendor-specific components (e.g., drivers for specific chipsets and other customizations) from the core Android OS framework. The rationale behind this change is to make it easier for OEM to apply (security) patches to their customized AOSP. In fact, AOSP patches only touch AOSP-related code and do not touch the vendor-specific portion. Thus, an OEM that respects Project Treble's core principle can always cleanly apply AOSP security patches without worrying about backward compatibility and other integration problems.

Finally, with Project Treble, the test suites have also been augmented with the *Vendor Test Suite* (VTS), which helps to validate the vendor interface and ensuring forward compatibility of vendor implementations. According to Google's documentation [4], the VTS can be thought of as an analogous of the CTS, and it can be used to automate the testing of the hardware abstraction layer and OS kernel, in both legacy and current Android architectures. We note that compliance with the VTS is strictly required for any ROM that wishes to run Google's software suite, also known as *Google Mobile Services* (GMS), which includes popular software like the Google Play Store, GMail, Google Maps, and YouTube. VTS compliance is also required for a device to be branded under the 'Android One' label [5].

Some works show how vendors' customizations introduce vulnerabilities with severe security repercussions. Researchers focused on customized drivers [6] and customizations of the Android framework [7], but we still lack a complete picture of Android OEM customizations over time and that tackles different aspects of the OS security perimeter. Hence, we built a fully automated analysis pipeline tailored to the analysis of Android OEM customizations, and we used our framework to perform the first large-scale longitudinal study on Android OEM customizations. The analysis was performed on a dataset of 2,907 ROMs from 42 different OEMs. This dataset was obtained by crawling OEMs websites, which often contain direct links to ROMs, and it consists of ROMs published from the year 2010 to 2020 and covering ROMs from Android version 2.3 to version 9.0.

From a high-level perspective, our analysis focuses on two key aspects: 1) whether a given OEM complies with the various regulations imposed on Android-branded devices (e.g., CDD, CTS, VTS); and 2) whether and how the various OEM customizations affect the security posture of the entire OEM. To investigate these two aspects, our study considers a wide range of technical aspects, including customizations of the security hardening of binaries, SELinux policies, Android's `init` scripts, and kernel security hardening settings.

Our large-scale measurement allows us, for the first time, to answer several security-related questions. For instance, are Google's automated compliance checks sufficient to detect CDD and VTS violations? Do certified ROMs violate some of the requirements? Do ROM customizations follow Project Treble's principle of keeping vendor-specific changes to the vendor partition (so to ease the application of security patches)? What kind of customizations is most prevalent? Do these customizations affect the overall security posture? For what concerns the vendor-specific binaries and data, how do their security settings (e.g., hardening techniques) fare when compared to the ones adopted in the main AOSP baseline?

Sadly, the answers to these questions are often worrisome. We identified that 579 over 2,907 (~20%) Android-branded ROM violate at least one "must comply" CDD rule, while 289 (~10%) do not implement at least one "strongly recommended" suggestion.

While some of these violations may have gone unnoticed by Google because of the technical challenges involved when automatically analyzing ROM—a challenge that we nonetheless successfully overcame—some of these violations are surprisingly obvious, and even the automated CTS and VTS tests can raise warnings. This result casts some shadow on the effectiveness of the ROM certification process. Our analysis also identified violations concerning Project Treble guidelines: in particular, we found ROMs that significantly modify non-vendor partitions, thus affecting the ease of application of security patches. Even though we believe that the principle and the intent of Project Treble are valuable, its effectiveness is hampered by the lack of a strict enforcement procedure.

Finally, we identified several customizations whose security impact, regardless of whether they constitute or not a violation of the guidelines, is significant. For example, we have found that 29% of ROMs with SELinux modified their policies in a way that bypasses *never allow* specifications of the main AOSP SELinux policy: we identified cases that "commented out" *never allow* SELinux policies to compile their customized version of the policies. We also found devices shipping `init` scripts implementing invasive customizations. For instance, we found a vendor that ships a ROM with an outdated version of `tcpdump` (with a known CVE and public POC), running as root, at boot, and reachable by a remote attack. We also found several ROMs that do not use many of the hardening techniques that the Android security team has developed over the years [8].

We conclude this paper with several recommendations for Google. In particular, we identified several improvements to extend the compliance requirements that can be automatically verified, and we discuss several proposals in terms of guidelines that Google could add to its official documentation to discourage customizations that affect the security posture of customized ROMs.

In summary, this paper makes the following contributions:

- We perform the first longitudinal and large-scale analysis on 2,907 Android ROMs, over 42 OEMs and spanning over 10 versions of Android, to explore how customizations affect the Android System Security.
- Our analysis takes an in-depth look at two key aspects: *compliance*, which checks whether a certified ROM actually follows the rules, and *security posture*, which focuses on how customizations may affect the security of the overall device.
- We identified numerous certified ROMs—and thus supposed to have passed the test suites and compliant with all the requirements dictated by Google through CDD— that actually do not meet the security prerequisites.
- We highlight how vendor-specific components significantly lag behind with respect to the security posture of the main AOSP, and we uncover several techniques that, even though are not strict violations of the guidelines, create security holes in AOSP main safety nets (e.g., SELinux policies, software hardening).

In the spirit of open science, we will release the detailed results and we will make our analysis pipeline available to the community.

II. Life of a ROM

A. *What is in a ROM*

We use the term ROM to refer to a phone firmware based on the Android operating system. Devices come with a pre-installed system, called *stock ROM*, which is often provided in the form of an archive (with different compression schemes), to allow users to restore the device to factory settings. A ROM contains all the necessary software components, policies, and configurations needed by the system to boot and work properly. Among the software components present in a ROM, we find, for example, the various executables and system libraries, the pre-installed applications, all the scripts necessary for the system to be configured correctly at boot (Android Init Script), and a series of security policies (such as SELinux and SECCOMP) intended to make the system safer. All these components are organized in a set of partitions. The first partition common to all systems is the boot partition, which contains the Linux kernel image. Then, depending on the Android version used by the vendor as the base for its system, the partition layout, and the filesystem may vary. For instance, if the system is based on an Android version before 8.0 (SDK 26), all these components are likely placed inside a single `/system` partition. Otherwise, if the device is based on an Android version equal or greater than 8.0 and has been subject to the re-architecture of Project Treble, all the customizations made by the vendor are delegated to a separate `/vendor` partition. As we already explained, this separation allows for a more straightforward application of the security patches provided by Google. Unfortunately, our study shows that in practice, this is often not the case.

B. *ROM Customization*

The process of creating an Android-based system requires numerous steps. First, the vendor must decide which version of Android to use as the basis for its system. Once the version (and therefore its SDK level) has been decided, the vendor proceeds to fork the corresponding tagged branch from the official repositories of the Android Open Source Project.

A counter-intuitive fact is that a single Android version (e.g., Android 9, codename *Pie*) might have multiple tags to use as base image: for example, just for the Android Pie, Google released *47 different base images* at different points in time [9]. Hence, a vendor that bases its custom Android system on Android Pie can decide which base image to use across those 47 different versions officially provided by Google. Each of these images might differ from several aspects: a newer release might provide some fix for disclosed vulnerabilities or other usability issues, introduce new binaries and services, or change the default configuration for a specific component.

Once a vendor obtains a base image, it then applies customization and modification to the entire system, either by introducing new components (e.g., new binaries and services) or modifying core services. Changes are not limited to user-space software components only. Typically, the vendor also inserts kernel components into the system (such as drivers for custom peripherals) and can also make changes to security policies or init scripts.

When the vendor has completed the system modification process and is ready to market its device, it can decide whether it wants the device to become an *Android Google Mobile Services certified* device or to remain a generic device built on top of the AOSP. If the vendor wants to use the Android brand on its device, it must request a certification from Google. Having this certification also allows the vendor to include all Google apps within its ROM, such as GMail, or Google Maps. Depending on the type of device that the vendor wants to market, with or without a Google license, the vendor is required to pass a series of tests, which we illustrate next.

C. *Compliance Checks and Requirements*

We now present the different types of tests vendors must pass to have a device compatible with AOSP or the GMS certification by Google.

To release an *Android-compatible* device, vendors *must* comply with the guidelines defined in the *Android Compatibility Definition Document* (CDD). The CDD enumerates all the requirements that must be satisfied by a vendor to have a system compatible with a given version of Android. For each new Android release, Google maintains and publishes a new CDD, where they define the new guidelines for several aspects, like compatibility with the multimedia framework or with the hardware. Security also plays a crucial role in the CDD that, from its first edition, contains an entire chapter dedicated to the *Security Model Compatibility*.

If the vendor wants instead to obtain a Google certification and brand its device as *Android*, it must pass numerous tests aimed at analyzing and verifying first the compatibility with AOSP, but also the security of the whole system. The first class of tests is defined by the *Compatibility Test Suite*, a series of tests aimed at ensuring that the device is entirely compatible with AOSP. Many of the tests performed in this test suite verify that the requirements defined in the CDD are respected.

If the vendor wants its devices to include all the Google applications, it must also comply with the *GMS Requirements Test Suite* (GTS): once passed, these tests allow the vendor to obtain the Google license for their apps. If the vendor's system is based on a version of Android redesigned with Project Treble, the approval process requires the vendor to pass another series of tests, named *Vendor Test Suite* (VTS). The VTS consists of a set of frameworks and test cases designed to improve the robustness, reliability, and compliance of the Android system (e.g., Hardware Abstraction Layers and libraries) and low-level system software like the OS kernel. All these tests are run by the device manufacturer [10] thanks to Tradefed [11], a continuous test framework designed for running tests on Android devices. If all tests pass correctly, the device is considered compliant with the CDD and with all the security and compatibility requirements defined by Google.

III. ROM Analysis Framework

In this section, we present an overview of our ROM analysis framework and we discuss how we extract different security-relevant information, such as binary security settings, SELinux policies, `init` scripts, and kernel security settings.

A. Architecture Overview

Given a ROM as input, our framework automatically detects the compression schemes and the filesystem type, and it unpacks the ROM for the analysis. Once the ROM is unpacked, the system then proceeds by identifying the AOSP tag used by the vendor to build the firmware. This step is fundamental to perform our analysis. In fact, the process of identifying how the vendor customized a given ROM can be seen as a differential analysis of the ROM with respect to the AOSP baseline that the vendor selected when customizing its version. This phase is fundamental when trying to understand whether a vendor customization introduced an error, a misconfiguration, or a new vulnerability, or whether the problem was already present in the original AOSP code.

Once our system identifies the starting AOSP tag, it then clones and compiles the corresponding repository to build a reference image on top of which it can perform the differential analysis. This process is repeated for each ROM.

Finally, the system extracts information related to binaries and libraries (ELF), SELinux policies, init scripts, and Linux kernel configurations. Each of these components is handled by an ad-hoc plugin, which we discuss in detail in the upcoming subsections. The entire procedure takes approximately 20 minutes for each ROM.

B. Tag Identification

Finding the right base image (identified by a git tag) used by a vendor as a starting point for its customization is crucial for our work. Unfortunately, identifying the base image used by a given ROM is not always a straightforward process as there are often many different base images for each "main" version of Android. This section discusses the various techniques and heuristics we developed to pinpoint the base image used by a given ROM.

During the building process of a system image, the build system adds a large amount of information that may help recover the exact git tag forked by the vendor. However, since the vendor controls the entire building system, this information might—and, in fact, often is—removed entirely.

In case the vendor did not modify the build system, the ROM usually includes a Build ID that uniquely identifies the starting base image. The format of this identifier may change across different Android versions, and it resides in the `ro.build.id` property of the `build.prop` file. Therefore, we built a mapping between Build IDs and base images, starting from the official Android documentation [9]. This mapping shows that, for example, a Build ID equal to `NOF27B` corresponds to Android Nougat release 25 (`android-7.1.1_r25`).

However, if the vendor modified the build system and stripped this information, the identification becomes more challenging. In these cases, we adopt different strategies. First, we look at different values including the `ro.build.description` property (that may still contain the original Android build identifier) and the `ro.com.google.gmsversion` property (which, when combined with the `ro.build.version.sdk` value, can be used to pinpoint the base image). It is important to note that this value should always be present when the vendor obtained the *GMS* certification. However, there is no guarantee that the vendor obtained this certification, and we also found ROMs that contain GMS apps but that however did not include a `gmsversion` property.

If none of these pieces of information is available, we rely on the combination of two properties that are *always* included: `ro.build.version.sdk` (i.e., the Android version) and `ro.build.date.utc` (i.e., the build date). By combining these two values, we can determine the "best" candidate to be considered the base image. In particular, we first list all the AOSP `tags` associated with the target Android SDK version, and we then identify the tag with the nearest creation timestamp.

We note that a vendor cannot easily modify these two final values, because that would introduce usability problems: changing the `sdk` value might introduce undefined and unexpected behavior both from the system and the apps, while changing the `build date` might introduce issues when dealing with system updates (e.g., anti-rollback protections might use this information to avoid booting older firmware [12]).

As explained in Section IV, this process worked well in practice. Moreover, even if some errors might have occurred, our analysis is not particularly sensitive to small imprecisions.

C. Analysis of Binary Customization

We start our analysis by looking at the binaries (both ELF executables and APK's native libraries) contained in the ROM. This is particularly important because most of the critical bugs are found within these binaries components, as they are created by using unsafe languages (e.g., C, C++). Moreover, customizations have been the root cause of several recent critical 0-click bugs, e.g., those recently reported by Google Project Zero's in the (custom) Skia component of Samsung devices [2].

In this phase, we check how customizations affect three main aspects related to binaries. First, we focus on security hardening techniques: we check whether vendors introduced customizations that lower the security posture of existing AOSP binaries. Second, we check whether the vendor introduced new functionality by adding new binaries or by modifying existing ones (we check for modifications of these binaries by inspecting ELF metadata such as the symbol table). Third, we check how the security posture of new binaries compares to exiting AOSP binaries and settings.

D. Analysis of SELinux Policies

Security-Enhanced Linux (*SELinux*) is a Mandatory Access Control (MAC) system developed by the NSA and Red-Hat and publicly released in December 2000.

SELinux policies are used to define rules that a process should follow. More precisely, rules apply to *contexts* (each process belongs to a context) and can be very fine grained: in fact, every resource on the system (e.g., files, sockets) is labeled, and rules can specify access policies for each of them. For instance, a rule can indicate that a process in context X is allowed to open a network connection. SELinux follows the principle of least privilege: if no rule grants a capability C to a context X, then X does not have access to that capability.

SELinux policies became partially enforced in Android 4.3 and fully enforced in Android 5. However, during the years, SELinux has proved to be at the same time a powerful exploit mitigation [13], but also the direct cause of several critical security issues due to vendor customizations [14]. Indeed, vendors *must* customize SELinux policies as every new file (including those introduced by the vendor) need to be appropriately labeled, and new rules need to be introduced to give proper access to the right contexts. However, these customizations may also have security repercussions. For example, since base AOSP SELinux can be quite restrictive, vendors may be tempted to relax the policies and somehow circumvent the safety nets implemented by AOSP. To give an extreme example: AOSP defines several "never allow" SELinux rules, which are rules that tell the SELinux compiler "refuse to compile if a different rule is violating it": We found several ROMs with customized SELinux policies that violate base AOSP rules: *this implies that the vendor must have commented out the problematic "never allow" rule instead of redesigning their customization more safely.*

Our analysis framework first extracts all customizations to the base SELinux policy and then inspects them to identify several problematic patterns, like the one discussed above.

E. Analysis of Init Scripts

Unlike other Linux systems, Android uses its initialization process. Android init scripts are textual files with the `.rc` extension and they are written in a dedicated language, namely the *Android Init Language* [15]. During the years, this component has been subject to several attacks [16]. Most of the vulnerabilities were introduced by third-party customizations, and, most of the time, their exploitation allowed a local attacker to escalate privileges to `root`.

Init scripts are loaded at boot, just after the kernel initialization, and play a crucial role in the Android system setup and bootstrap. Default AOSP init scripts are located in the `/system/` directory, while vendors can add custom scripts in the `/vendor/` or `/odm/` folders.

Init scripts can specify the user/group the binary should be run with, the Linux capabilities that should be granted, and the SELinux context the program should be run with (by default, all init scripts run within the `init` context).

Given the potential security consequences of improper customizations, our framework includes support for the analysis of Android init script to study whether vendors customize default AOSP init scripts or add new ones and to verify if the new services are executed with appropriate user/group and Linux capabilities, and as part of a "safe" SELinux context.

Unfortunately, our experiments show that vendors often customize these scripts, and in some cases, significantly increase the attack surface and leave the device vulnerable to remotely exploitable bugs (with known CVEs).

F. Kernel Security Analysis

Kernel security has grown in importance in recent years as the number of kernel security bugs reported for Android increased almost ten times in only three years [8]. As a consequence, many kernel hardening techniques were recently introduced. These are so important that the CDD itself introduced a number of "must" requirements in this area that a vendor needs to satisfy to brand its devices as Android.

In Android ROMs, the kernel is usually provided in a binary form within the `boot.img` file. The Vendor Test Suite implements checks for some of the CDD requirements, but unfortunately, they are quite limited due to the binary-only format of the kernel.

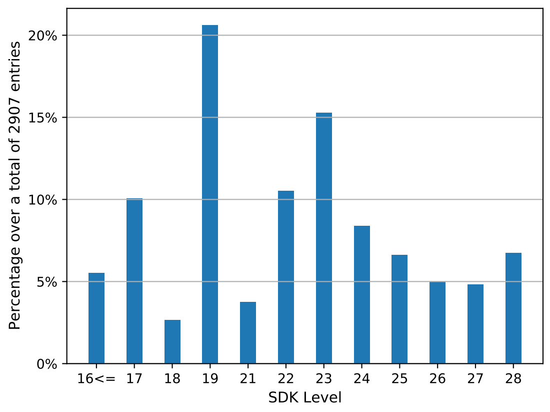

Fig. 1. SDK Distribution

To study the kernel's security, our framework includes various analyses that can extract several security-sensitive information and test for additional CDD requirements. For each kernel, the system first extracts its version, and all the information generally provided within the *Linux Kernel banner* [17]. It then attempts to extract the kernel build configuration options. If the kernel was compiled with the `CONFIG_IKCONFIG` option, the required information is easily accessible through the `.config` file. Otherwise, we created a database that maps an ELF symbol to the kernel config option. Our analysis then extracts symbols from the kernel image (using a modified version of `vmlinux-to-elf` [18]) and uses our mapping to infer several kernel config options used at compilation time. It is worrisome to note that, even though this approach does not support *all* kernel config options, it was sufficient to identify ROMs that violated several CDD "must" requirements.

IV. Dataset Characterization

To perform our longitudinal analysis, we set out to build a dataset of ROMs as comprehensive as possible. For what concerns the official Google ROMs, we downloaded them from their official website [19]. We downloaded the other vendors' ROMs from `firmwarefile.com` [20] and `stockrom.net` [21]. In total, our dataset consists of 2,907 Android ROMs, which span across 42 different vendors and cover 1,403 different device models. For what concerns the SDK distribution, our dataset covers the Android system's evolution from version 2.3.3 to version 9 (i.e., from SDK 10 to 28). The oldest image dates back to 2010, while the newest is from 2020. Figure 1 presents the distribution of our dataset in terms of SDK distribution.

According to public statistics [22], our dataset is also heterogeneous in terms of coverage of different vendors: half of our dataset is constituted by "big players" (e.g., Samsung, Huawei, LG, and Xiaomi), while the remaining ROMs belong to vendors with a market share less than 4% (e.g., Google, Lenovo, Mobicel, Motorola, Oppo, Realme, and Vivo).

Moreover, as discussed in the previous section, our customization analysis needs to compare a given ROM against its associated "base image." To this end, we also created a set of 326 ROMs by compiling *all* AOSP versions (i.e., tags) that are the base image of a least one ROM in our dataset (as discussed in Section III-B). Obviously, these last 326 ROMs are not counted in our statistics.

To identify the tag of the AOSP base image, we relied on the `ro.build.id` value for the 88% (2,566) of our dataset. For 9% (261) of the ROMs we combined instead the information from the build date (`ro.build.date.utc`) with the `ro.build.version.sdk` property values. Finally, we relied on the information contained in the `ro.com.google.gmsversion` property for 2% (59) of the ROM and the value of the `ro.build.description` for the remaining 23 ROMs. Note that all the 2,907 ROMs in our dataset contain apps from the Google Suite; thus, we can assume they all obtained the GMS certification from Google.

V. Compliance

All the ROMs in our dataset are branded Android and contain apps from the Google Suite (and thus obtained the GMS certifications). Therefore, one would expect them to be compliant with the mandatory requirements of the CDD.

This is important because system security aspects always played a crucial role in the CDD, which contained an entire chapter dedicated to the *Security Model Compatibility* since its first edition in 2009. Mandatory requirements are clearly marked as "*must*," and a failure to implement them is a clear *violation* of the CDD. Alternatively, a feature can be defined as *strongly recommended*: in this case, not implementing such a feature is not a strict violation of the CDD. This section discusses our analysis and findings.

We manually extracted all these requirements from the CDD of Android 1.6 to Android 9, as summarized in Table I. In order of appearance, the first system hardening requirement was introduced in the CDD of Android 4.3, where Google announced the support of *SELinux* for Android devices. Therefore all ROMs based on Android 4.3+ *must support and implement the SELinux Mandatory Access Control*. Then, starting from Android 7, the Security Model Compatibility section has focused mainly on kernel configuration options. Surprisingly, the CDD security requirements do not mention *user-space hardening* until Android 9, and the only user-space hardening requirement is defined only as *strongly recommended*. Since the introduction of Project Treble made updates faster and easier for OEMs to roll out to devices and introduced several tests to verify and test the OS kernel, we present our results divided before and after its introduction. This distinction can help us understand to which extent the introduction of Project Treble was able to mitigate the problem of Android customizations.

A. Kernel Configurations Compliance

As discussed in Section III-F, for each of the 2,907 ROMs, we analyzed their Linux-based kernel binary to identify potential misconfigurations in contrast with the strict requirements defined in the CDD. However, we identified that 262 ROMs in our dataset did not contain the kernel binary, and therefore we excluded them from our analysis.

For 249 of the remaining 2,645 kernels, our system was unable to extract neither their kernel configuration nor the symbols from the kernel binary. This is because those kernels were compiled without the `CONFIG_IKCONFIG` and `CONFIG_MODULES` [23] configurations. However, as described in [24], both configurations *must* be enabled for kernels targeting Android 8.0 and higher. Besides, by reading the *Core Kernel Requirements* [24] defined in the *Vendor Test Suite (VTS)* for Android 8.0 and higher, we noticed how the configuration of these 249 kernels should violate and fail the tests. Out of the 249 kernels missing these configurations, 162 corresponded to the Android version ≥ 8.0. Thus, as the first result of this analysis, we highlight how these 162 kernels *are not compliant with Android*, and these misconfigurations should have been detected by the correspondent VTS. This casts a shadow on the strictness of these requirements' enforcement, especially since some of these could have been automatically checked.

For the other 2,396 kernels, we retrieved the textual configuration from 561 kernels and the symbol table for the remaining. Identifying violations on kernels having their configuration is straightforward as the CDD precisely indicates which configuration options must be used. On the other hand, verifying violations with the only support of the kernel binary symbols is not immediate. However, we noticed how almost every kernel configuration defined in the CDD introduces a set of specific symbols, and therefore it is possible to infer a specific compilation flag based on the symbols included within the binary (Table VI, in Appendix, shows the mappings between kernel configuration and symbols).

It is important to note that since some flags are interchangeable, we conservatively mark a kernel to be *not compliant* if and only if it does not implement *any* of the available options. For example, if a kernel adopts `CONFIG_CC_STACKPROTECTOR_STRONG` rather than `CONFIG_CC_STACKPROTECTOR_REGULAR`, we *do not* mark it as not compliant since the CDD requires the vendor to implement at least one of the two.

Our analysis identified that 7.9% (190 out of 2,396) of the kernels (from 10 different vendors) violate the CDD for their specific Android version since they do not implement one or more *mandatory* security requirement.

Amongst these 162 are used in ROMs re-architectured with Project Treble, thus targeting an Android version greater or equal than 8.0. The most common violation, found on 150 kernels, relates to the absence of kernel memory protections aimed at marking sensitive memory regions and sections read-only or non-executable (which can be enabled with `CONFIG_DEBUG_RODATA` or `CONFIG_STRICT_KERNEL_RWX`).

We also identified 10% (241 out of 2,396) of the kernels (from 10 vendors) do not implement one or more strongly recommended features. This time, we noticed how 160 vendors did not enable `CONFIG_RANDOMIZE_BASE` (no Kernel Address Space Layout Randomization); hence, these kernels do not implement any randomization of their base address once loaded. Although these features are not mandatory, the Vendor Test Suites inform the vendor if any strongly recommended features are missing. Thus, even though these vendors were warned about the lack of these features, they ignored the advice and did not include them in their final product.

Table II shows the evolution of violations across different SDK levels. The table shows that the re-architecture introduced with Project Treble and the testing performed with the VTS are not

TABLE I
KNOWLEDGE BASE: THE TABLE SUMMARIZES ALL THE SYSTEM HARDENING REQUIREMENTS DEFINED IN THE ANDROID COMPATIBILITY DEFINITION DOCUMENT (BOTH THE *must* (I.E., MANDATORY) AND THE *strongly recommended*) THIRD-PARTY VENDORS SHOULD IMPLEMENT TO BRAND THEMSELVES AS "ANDROID"

SDK	**Version**	**CDD**	
		MUST	**STRONGLY RECOMMENDED**
4 - 17	1.6 - 4.2	–	–
18	4.3	**SELinux:** support Permissive Mode	–
19	4.4	**SELinux:** contexts installd, netd, and vold in Enforcing Mode	**SELinux:** other domains remain in Permissive Mode
20 - 23	5.0 - 6.1	**SELinux:** global Enforcing Mode	–
		SELinux: all domains in Enforcing Mode	
		SELinux: not modify, omit, or replace the neverallow rules present within the SELinux AOSP folder	
24 - 25	7.0 - 7.1	**Kernel:** support for seccomp-BPF support (TSYNC)	–
26 - 27	8.0 - 8.1	**Kernel:** support for CONFIG CC STACKPROTECTOR REGULAR or CONFIG CC STACKPROTECTOR STRONG	**Kernel:** support for data read-only after initialization (ro after init)
			Kernel: support for CONFIG HARDENED USERCOPY
		Kernel: support for CONFIG DEBUG RODATA or CONFIG STRICT KERNEL RWX	**Kernel:** support for CONFIG CPU SW DOMAIN PAN or CONFIG ARM64 SW TTBR0 PAN
			Kernel: support for CONFIG RANDOMIZE BASE
28	9	**Kernel:** support for CONFIG PAGE TABLE ISOLATION or CONFIG UNMAP KERNEL AT EL0	**Userspace:** do not disable CFI/IntSan on components that have it enabled
		Kernel: support for CONFIG HARDENED USERCOPY	

TABLE II
VIOLATIONS REGARDING THE KERNEL CONFIGURATION

SDK	**Version**	**# Kernel**	**# Violations CDD**	**# Strongly Recommended**
18	4.3	77	26 (33.8%)	–
19	4.4	599	3 (0.5%)	–
26	8.0	145	50 (34.5%)	70 (48.3%)
27	8.1	140	33 (23.6%)	66 (47.1%)
28	9.0	196	78 (39.8%)	101 (51.5%)
		2396	190 (7.9%)	237 (9.9%)

enough to counter the problem of customization on Android from the Kernel Security perspective.

On the contrary, it can be observed that many kernels still do not comply with the directives imposed by Google and continue to release on the market devices equipped with kernels that do not meet the mandatory security specifications. The numerous tests should have identified (and likely actually did identify) all these violations, which would be enough to mark the final ROMs as *non-compliant*.

B. SELinux Compliance

For each Android version that supports SELinux, AOSP provides a standard policy that vendors can use as a base to build and customize their SELinux configuration. As discussed in Section III-D, starting from Android 4.3, Google introduced as a strong requirement that all third-party vendors *must* adopt this new Mandatory Access Control system. The CDD mandates that third-party vendors *must support SELinux in Permissive Mode*[1]. Instead, from Android 4.4, Google started to protect few critical services with SELinux and forced the vendors to do the same: hence, vendors were required to set up SELinux in *Enforcing Mode* at least for the three domains `installd`, `netd`, and `vold`. Starting from Android 5.0, instead, vendors were required to set up SELinux in Enforcing Mode for all the domains. Moreover, from this version, vendors *must not* modify, omit, or replace some AOSP specific rules, which act as a safety net for misconfigurations. These rules are the so-called `neverallow` rules: if a custom SELinux policy directly or indirectly violates any of these rules, the SELinux toolchain would throw a compilation error, thus preventing the adoption of unsafe configurations from the beginning. With these rules, it is possible to *avoid and mitigate potential known security issues and harmful behaviors*, such as forbidding any third-party application to write to files in the `/sys` directory or preventing them from receiving and sending `uevent messages`. We note that modifying (or removing) any of these `neverallow` rules is a strict violation of the CDD.

[1]When SELinux runs in Permissive Mode, every violation is logged, but not enforced, so to provide vendors enough information for an adequate fix to the component causing the error.

To determine whether a ROM is compliant with the SELinux requirements, we proceed in two steps. First, we look at violations related to *Permissive Mode* by inspecting the SELinux policy available in the ROM (since it is possible to retrieve all the permissive domains directly from the compiled policy). Second, we look for vendors that manipulated the base policy provided in AOSP to overcome the restrictions imposed by the `neverallow` rules. For this, we retrieve the tag of used as a base image by the vendor (see Section III-B), and we compare the two sets of policies.

Out of the 2,907 ROMs, we identified 1,090 of them not containing a SELinux policy. Of these 1,090, 452 are targeting an Android version lower than 4.3, and it is thus expected that they do not have any policy.

Since SELinux must have kernel support to work, we decided to intersect the remaining ROMs with the results extracted from the previous kernel analysis (see Section V-A) and we identified how 29 lack `CONFIG_SECURITY_SELINUX`: for those, it is expected that we do not find SELinux configurations.

The remaining 609 ROMs are divided as follows: for 167 we were not able to obtain the `boot.img`, and for 91 of them we were not able to extract neither the kernel configuration nor the symbol table; thus, we cannot perform any measurement on these

TABLE III

VIOLATIONS REGARDING THE CONFIGURATION OF A PERMISSIVE DOMAIN IN THE SELINUX POLICY

SDK	Version	# ROM CDD Violations	Permissive Domains			
			Max	Min	Avg	σ
21	5.0	1/58 (1.7%)	5	5	5.0	0
22	5.1	26/251 (10.3%)	7	1	3.1	2.2
23	6.0	21/359 (5.8%)	5	1	2.0	1.3
24	7.0	11/226 (4.8%)	2	1	1.0	0.3
25	7.1	2/163 (1.2%)	1	1	1.0	0
26	8.0	21/141 (14.8%)	4	1	2.0	1.3
27	8.1	18/139 (12.9%)	1	1	1.0	0
28	9.0	8/196 (4.0%)	1	1	1.0	0
		108/1533 (7.0%)				

TABLE IV

VIOLATIONS REGARDING THE DEFINITION OF ALLOWRULES IN CONTRAST WITH A NEVERALLOW RULE DEFINED IN THE AOSP SELINUX BASE POLICY

SDK	Version	# ROM CDD Violations	Neverallow Rules Violations			
			Max	Min	Avg	σ
21	5.0	1/58 (1.7%)	8	8	8.0	0
22	5.1	20/251 (7.9%)	39	1	4.6	10.4
23	6.0	58/359 (16.1%)	121	1	3.6	15.7
24	7.0	8/226 (3.5%)	10	1	7.2	3.9
25	7.1	3/163 (1.8%)	158	1	56.3	88.1
26	8.0	121/141 (85.8%)	27	1	7.2	7.7
27	8.1	110/139 (79.1%)	25	2	7.2	7.5
28	9.0	122/196 (62.2%)	37	1	4.0	8.8
		443/1533 (28.9%)				

ROMs. For 351 ROMs, we identified that they correctly support SELinux at kernel level, but no policy has been found: we suppose these might be incremental updates, not containing the policy.

We now focus our discussion on the remaining 1,817 ROMs that define a SELinux policy. Out of them, 7% (108 ROMs) violate the CDD specification for their corresponding Android version as they still define one or more `permissive` domains. We found this violation spread across 16 different vendors. We also analyzed the distribution of these violations with respect to their SDK level to determine whether this problem only affects older versions of Android. Surprisingly, we noticed that even if Google forbids permissive domains starting from Android 5.0 (and thus from SDK 20), several ROMs are still not complaint even after *four* major releases, and after an almost complete redesign of SELinux on Android 8 [25].

Table III summarizes the results of this analysis. We divided the results before and after the introduction of Project Treble to show once more how the problem persisted even after the introduction of the new system design.

We then performed the second analysis to identify whether a vendor tampered with any of the predefined `neverallow` rules, which is a strict violation of the CDD, starting from Android 5.0. However, detecting this type of violation is not straightforward. Each Android version contains a preset of SELinux rules: the `neverallow` rules are part of this base policy. At compilation time, the SELinux policy compiler *verifies if any rules defined in the policy are in contrast with what is defined by the neverallow rules*: if a violation is identified, the compiler throws a compilation error. However, these checks are performed *only* at compilation time and are not enforced at runtime. Therefore, potentially, a third-party vendor facing a violation of a `neverallow` rule introduced by one of its customizations may be tempted to "solve" the issue by just changing or removing the `neverallow` rule that prevents the compilation of the final policy. Thus, by analyzing only the vendor policy of the final ROM is not possible to conclude whether it violates the CDD requirement.

To detect these violations, we proceed as follows: for each ROM, we first retrieved the tag used by the vendor as a base system (see Section III-B), and we save both the textual and the compiled version of the policy. Then, we identify all the differences between the compiled policies, and we collect the customizations introduced by the vendor. For each of the additional vendor-only rules, we then try to recompile the original AOSP policy with the addition of the new rule, and we check for compilation errors. In case of compilation errors, we finally check whether it is due to a *neverallow rule violation*, and if so, we mark the vendor policy as not compliant.

Out of 1,533 ROMs with a SELinux policy (and that target Android ≥ 5), we identified that 29% of them (443) violated the CDD by defining one or more rules violating one of them default neverallow rules. For all these images, from 21 unique vendors, it was possible to identify SELinux policies allowing operations that were not supposed to be available. Table IV summarizes the results of this second analysis. Also in this case, the introduction of Project Treble failed to mitigate the vendors' problems related to SELinux customizations. As can be seen from Table IV, if we consider the results for SDK level 26, 27, and 28, we see how this problem has increased dramatically, reaching peaks of 85% of the ROMs having at least one violation.

Although the Vendor Test Suites contain tests to check for SELinux violations, and specifically to identify this type of violations [26], the results we have collected show that not only these are easily bypassable by vendors, but that *vendors actually do violate them very often*, making them not compliant with the CDD, and potentially introducing security issues. We would like to note that these results do *not* imply maliciousness: we believe most of the vendors use this practice to fix compatibility issues introduced by their customizations quickly. Indeed, modifying or commenting out a `neverallow` rule is much easier than potentially re-architecting a customization to fit the requirements.

C. Binary Compliance

The last category of system hardening defined by Google is related to user-space binaries. As previously discussed, the requirements for binaries were introduced only in Android 9, and so far, they only cover two aspects: *Control Flow Integrity* (CFI) and *Integer Overflow Sanitization* (IntSan). CFI is a security mechanism that tries to prevent changes to the control flow of a compiled binary, making exploitations that require hijacking the "expected" control flow much harder. IntSan provides instead compile-time instrumentation to detect signed and unsigned arithmetic integer overflow. When an overflow is detected, the process safely aborts. Both protection systems have been gradually introduced by Google to harden the Android Media Stack component [27], [28], which has been subject to numerous attacks over the years, including Stagefright [29], which could have been prevented with these two hardening techniques.

To take advantage of these new protections, the developer must use a compiler that supports them. Officially, Google uses and supports Clang, but both features are also available on the GCC compiler.

As presented in Table I, the only requirement for the user-space binaries is defined as *strongly recommended*, and it asks vendors to not remove CFI or IntSan compiler mitigations from components that have them enabled. Thus, to identify if vendors adhere to this recommendation, we proceed as follows: for each ROM, we first identify its AOSP base image (see Section III-B). Then, we extract all binaries shared between the vendor ROM and the corresponding AOSP base image. Finally, for each of these binaries, we tested their security features: if the original binary (present in the AOSP base image) has CFI or IntSan enabled and the corresponding binary in the third-party ROM does not, we mark the ROM as not respecting the recommendation suggested in the CDD.

Since both defense mechanisms were introduced in the CDD from Android 9, we only considered the 196 ROMs with SDK ≥ 28. Among them, 85 ($43.37\,\%$) contained at least one binary that disabled CFI and 104 ($53.06\,\%$) contained at least one that disabled IntSan. In these cases, six unique vendors *lowered the security of a binary*, with respect to AOSP, thus violating the CDD recommendation.

However, these vendors did not entirely disable CFI or IntSan for all the binaries: on average, among the ROMs that have violated the recommendations, the vendors disabled CFI for 38.7% ($\sigma = 36.5$) of the binaries, while they disabled IntSan for 35.8% ($\sigma = 34.9$) of them.

VI. Additional Customizations

This section discusses our analysis of OEM customizations that, even though may not constitute a strict violation of the requirements, do negatively impact the security posture of the overall ROM.

A. New Functions in System Libraries

The vast majority of Android's core system components are still written in unsafe memory languages like C and C++ and shipped as ELF libraries. Vendors can add functionalities to such libraries, which can result in an increased attack surface. A recent (fixed in May 2020) impactful example is a bug found in Samsung's customizations of Google's Skia library [2]. The library is used to process pictures for many applications, and Samsung customized it to add support for new proprietary formats. Unfortunately, one of these functions was vulnerable to memory corruption bugs, and the exploitation of this vulnerability allowed an unauthenticated remote attacker (i.e., 0-click) to execute arbitrary code on the device.

To assess the prevalence of vendor customizations that add functionalities, given a ROM as input, we first inspect all binaries that are also found in the original AOSP (we refer to this subset of binaries with the term *Shared Libraries*), and we extract the list of exported functions that were not present in the original version. Since a vendor can use different library versions that might have been taken from another AOSP branch, we opted for the following conservative approach: we consider a ROM to have

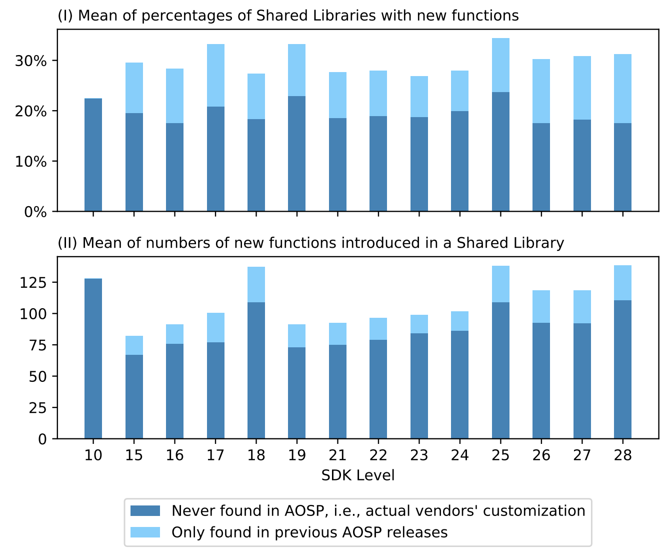

Fig. 2. Analysis of new exported functions introduced in AOSP libraries

added functionality to a given binary if it contains symbols that do not appear in *any* (i.e., older or newer) AOSP releases.

Figure 2 summarizes our findings. The two bar plots share the same x-axis (the SDK level) and report respectively the mean of the percentages of Shared Libraries in a ROM in which a vendor has introduced new functions and the numbers of new functions (when a vendor introduced at least one). The dark bar highlights the *actual* vendors' customizations, while the soft bar displays a modification found in a previous version of AOSP. There is no soft bar in the correspondence of SDK 10 because there are no ROMs older than that in our dataset.

The results show an almost constant trend of roughly 80 new functions added to 20% of the system libraries, thus vanishing Project Treble's efforts. Nevertheless, we also note that vendors are still using old AOSP functions, maybe because their code still depends on them. However, using a function that is no longer maintained in AOSP can be dangerous because it does not receive security patches.

B. Compile-time Hardening

In addition to the CDD, Google also maintains a Security Enhancements (SE) webpage [30], in which it presents the security and privacy enhancements for each Android version. While the CDD only started to discuss binary hardening in Android 9 (2018), the SE discusses this topic since Android 3 (2009). This webpage is not directly linked from the CDD, so it is not mandatory for the OEMs to implement such enhancements. However, since these aspects are security-relevant, we analyzed customizations related to these aspects as well.

We first went over all the security features reported in the SE and collected all mitigation techniques related to binary hardening: I) Stack Canaries, II) No eXecute (NX), III) Position Independent Executables (PIE), IV) Full Relocation Read-Only (Full RELRO), V) Fortify Source, and VI) the use of `setuid`/`setgid` binaries. (We do not mention CFI here because it was already discussed in the previous section.) We then compiled a list of artifacts whose

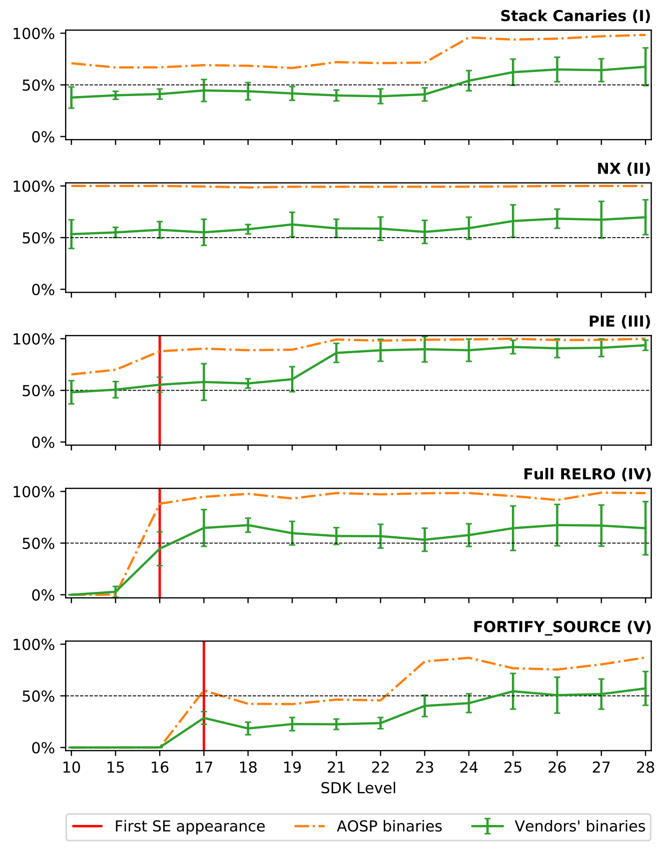

Fig. 3. Mean of percentages of binaries using a certain security feature

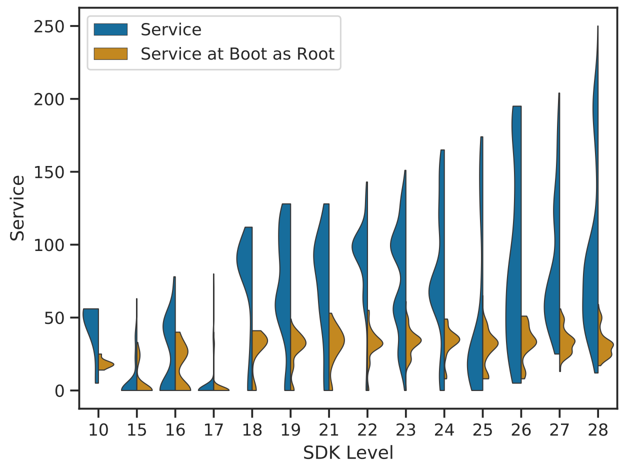

Fig. 4. Evolution of new services defined through Android Init Scripts

presence or absence can be used to infer whether an ELF binary implements or not each mitigation technique. The list is available in Table V in the Appendix, annotated with a full description of the features.

This information allowed us to compare the security-related compiler options used by vendors for their binaries with respect to the one used by the corresponding AOSP base image. First, we found a positive indication. The vendor's binaries in common with the AOSP base image have the same mitigation techniques, that is, third-party vendors do not (usually) modify the AOSP settings.

However, the results are different when we compare the binaries that are only present in the vendor's ROM (i.e., those binaries that are not present in any version of vanilla AOSP). Figure 3 presents the result of this analysis as the mean computed over the ROMs aggregated by SDK version. The vertical red line shows the point in time when the security feature was mentioned in the SE for the first time (Stack Canaries and NX have no vertical line because they were introduced even before SDK 10). The dash-dotted line represents the means of AOSP binaries, while the continuous line (supplied with standard deviation) the vendors' binaries. All graphs clearly show that the new binaries added by the vendors consistently use fewer mitigation techniques than the binaries in AOSP. At a closer look, we can also observe other interesting trends. For instance, even if stack canaries are the oldest security feature presented in the SE, it took several years for vendors to adopt them (and still today, around 40% of vendors binary lack this basic feature), probably because it slightly penalizes the performance [31]. NX adoption and Full RELRO have instead always been very common in AOSP binaries, while the gap with the vendors' binaries is still substantial.

Moreover, we found an inconsistency concerning NX adoption: the CDD never mentions NX, while the CTS contains a test to verify at run-time if NX is enabled [32]. SE webpage presents NX in Android 2.3, released in December 2010, and the test was committed in March 2011. This fact is odd because the test is checking (and thus enforcing) for a feature that is not explicitly requested.

Finally, we measured the prevalence of `setuid`/`setgid` files. Since Android 4.3 (SDK 18), the AOSP removed all `setuid` executables. Among the vendor binaries, we found that $319/447$ (71%) of the ROMs with SDK < 18 and $371/2453$ (15%) of the ROMs with SDK ≥ 18 contain at least one `setuid` executable. In particular, ROMs with SDK ≥ 18 should never contain any `setuid` executables (and in fact AOSP contains none). At a closer analysis, the binaries that appear more frequently in those ROMs are `su` (18%), `procmem` (17%), `netcfg` (16%), `procrank` (12%), and `tcpdump` (11%). Developers often use these executables for debugging purposes, but they should be removed from the final released ROM since the presence of `setuid` executables can severely affect the overall security posture of the device.

C. Android Init Script Customizations

Android OS relies on a custom init script system to start binaries at boot time. Unfortunately, this component has been subject to numerous security problems in the past, in many cases, due to changes introduced by vendors [16].

To study this aspect, our system extracts from each ROM how many *new* services it defines with respect to its corresponding AOSP base image. However, not all services are started every time. Therefore, among them, we mainly focus on those that start at system boot and that run with root user privileges.

Figure 4 summarizes our findings. For each SDK level, the figure displays the distribution of the number of new services: the left side of the violin plot covers *all* new services, while the right side only covers those that start at boot *and* with root permissions. The plot shows how, over the years, vendors have made considerable changes to init scripts and, in particular, how the total number of newly defined services is constantly growing—with some ROMs defining almost 250 additional services compared with AOSP.

To put this in terms of absolute numbers, for instance, an AOSP 8.0 (SDK 26) had, on average, 59 services defined in the init script, while an average ROM had 90, with a peak of vendors defining 195 additional services. The astonishing number of services that start as root is worrisome, and vendors are likely violating the least privilege principle. It is, in fact, much more straightforward for vendors to run a binary as root with respect to running it with less privilege, granting it only the capabilities that are strictly needed, and properly configuring SELinux policies.

Another interesting trend we observed while analyzing the init script ecosystem is how vendors customize AOSP specific services by changing or adding a *root* user as the owner of the service. Even though the numbers are not very high, we noticed how, over the years, starting from Android 4.0.3, on average, vendors customize at least one service. This customization is very hard to explain since these core services are supposed to run on the system without modification. Extending the permissions of these services might result in having an unnecessary over-privileged service. One possible cause of this change might be due to aggressive and dangerous customization that requires root privileges to work correctly.

We also cross-referenced the list of new services with the results obtained in the previous subsection, where we found binaries usually used for debugging purposes in commercial ROMs. Surprisingly, we found that some of those binaries are also automatically started at boot. For instance, we identified 18 ROMs (of 2 different vendors) configured to start `tcpdump` at boot and with root privileges. In this case, a manual investigation showed that the `tcpdump` process was configured to monitor incoming packets on all the interfaces and save the first 134 bytes of data from each packet into a log file. To make it worse, some of these ROMs use `tcpdump` version 4.9.2, which is outdated (it was released in 2017) and is affected by several CVEs [33], [34] some of which with public proof-of-concept exploits. Surprisingly, we identified these problems even in a ROM branded as Android One, and built in 2019. Listening and processing packets from untrusted sources expose the device to potential remote and local attackers, thus severely negatively affecting the entire device's security.

D. SELinux Customization

As presented in Section V-B, vendors often customize the SELinux configuration. This section discusses an in-depth analysis of the most frequent vendor SELinux-related changes and their impact on the overall system's security (independently from whether these changes are compliant with the CDD and other requirements).

SELinux plays a crucial role in the entire security of Android, and this component can also be used to introduce temporary patches to mitigate a vulnerability introduced at the software level. For example, the *vold* privilege escalation bug was properly mitigated first by a SELinux rule, before the vulnerable daemon `vold` was patched [35], [36]. Unfortunately, there might be cases in which vendors modify these policies without verifying whether the change can introduce new vulnerabilities (or make existing vulnerabilities exploitable). This is the case for Motorola that, by just introducing *one single* policy change for some of its devices, has introduced a logical bug that reverted the patch introduced to mitigate the problem on vold, allowing attackers to get root [37]. In other cases, the vulnerability can also be part of the base AOSP policy defined by Google. This is the case for *CVE-2018-9488* in which one of the default SELinux domain of AOSP was wrongly configured and allowed a local attacker to perform a privilege escalation [38].

These examples show that defining SELinux policies is a delicate and error-prone process. However, SELinux changes are necessary for the vendors. Every new process, file, and resource must be correctly labeled, and each new change introduced by a vendor must be configured accordingly. This means that each new component introduced by the vendor potentially requires introducing new domains, types, classes, and rules. In our analysis, we extracted and analyzed all vendor rules that were not present in the corresponding basic AOSP policy. We identify three different cases: 1) rules that modify a pre-existing rule to extend the permissions and operations allowed on a given resource; 2) rules that modify an exiting core policy domain but just by extending it to support new resources introduced by the vendor; 3) rules that are completely new and that operate on domains and resources that are not present in the original AOSP policy.

We now present the results of our analysis. Figure 5 shows the changes to the *allowrule* defined by a vendor for its system, while Figure 6 shows the changes affecting the definition of domains, types, and classes.

For each SDK, the first figure shows the distribution of the number of SELinux rules present in the policy. The graph combines a traditional boxplot on the left, showing the first and third quartiles, with a violin plot on the right side to show the distribution of the number of ROMs that define a given number of rules. The plot also includes two dots to indicate the average number of rules present in the correspondent AOSP policy compared to the average number of rules found in the different vendor policies. Also note that to accommodate outliers better, the Y-axis is plotted on a logarithmic scale.

We noticed how these outliers aggressively modified the default policy defined in AOSP to add a significant number of rules. For example, for SDK 27, an AOSP policy contained in average 10,000 rules, but some vendors defined a policy containing more than 232,000 rules (i.e., over a 20x increment). A similar trend also appears in the changes to the definition of domains, types, and classes, as presented in Figure 6.

These results highlight how the problem of customization has significantly affected SELinux policies and how vendors often behave very differently from one another. If we consider that even very restrictive policies with a very limited number of rules, such as those provided by AOSP, have been found to contain problems, it is difficult to foresee vendors' policies that introduce a number of rules 20 times greater than the average can be free from logical misconfiguration or even from real vulnerabilities introduced by a completely insecure rule.

Figure 7 presents a more fine-grained breakdown of the changes. In this case, for each modified rule, we checked if it applies to new

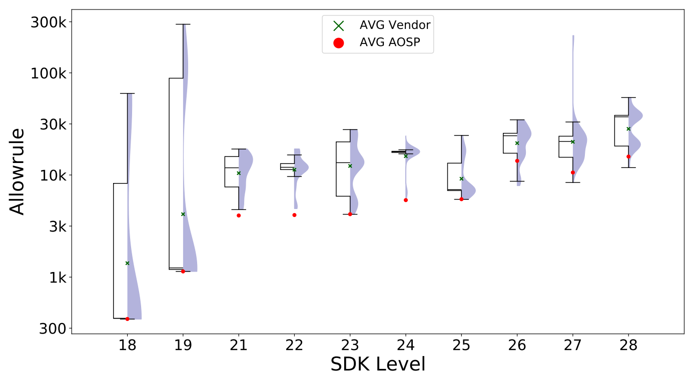

Fig. 5. Distribution of SELinux rules present in the policy (in log scale)

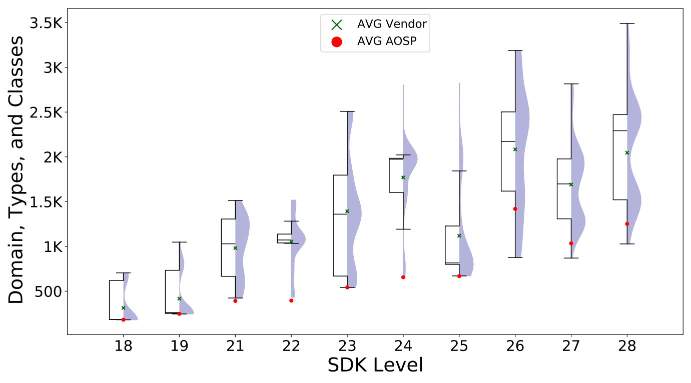

Fig. 6. Distribution of SELinux domains, types, and classes present in the policy

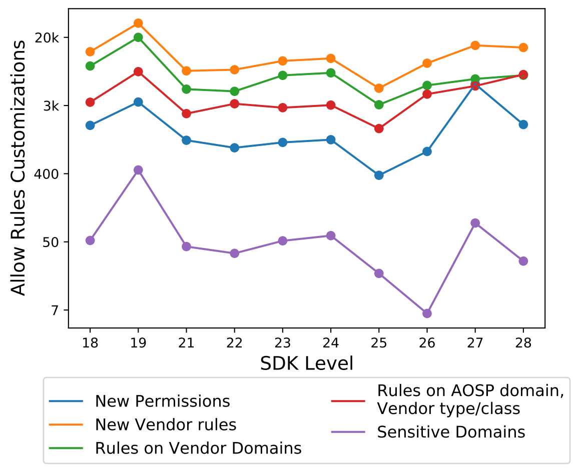

Fig. 7. Evolution and Classification of Rule Changes

domains added by the vendor, if it interests AOSP domains, if it is adding permissions to a previous rule, or if it is a modification that interests sensitive domains. The graph shows that most of the vendors' changes consist of rules that involve new domains that are not present in the original AOSP policy. These new rules are usually introduced by the vendors to configure custom components properly. We also observe a substantial number of changes to rules that affect domains shared with AOSP, but which see the introduction of classes and types that are not present in the basic policy. These numbers exemplify the problem of customization by showing how many changes the vendor makes to the initial policy configuration so that new third-party components can interact correctly with the entire system. But this also shows how intrusive vendor changes are, and how, as noted above, this dangerous trend is continuing to grow.

A more important finding emphasized by Figure 7 is the number of changes vendors made to the base policy, by extending the permissions and privileges for default AOSP domains. In principle, these rules only affect AOSP components that the vendor should not modify. However, if a vendor applies some customizations to services running on these domains, some of these modifications might raise a runtime SELinux violation since one new feature introduced by the vendor violates a rule defined in the original policy. Furthermore, this trend has seen a significant increase in versions from SDK 25 to 27. For instance, in SDK 27, we found vendors introducing more than 130,000 permission changes to the corresponding AOSP core policy.

Depending on the attack model considered, some SELinux policy changes can be more problematic than others. In particular, two domains, *isolated_app*, and *untrusted_app*, play a particular role in the security of the system, and therefore their basic policy included in AOSP is very restrictive. The *isolated_app* domain is mainly used as an additional sandbox for the Chrome renderer process, or to sandbox processes that should not have permissions of their own. Adding rules to this context could widen the attack surface for remote attacks, potentially allowing an easier sandbox escape. The *untrusted_app* domain is used instead for all third-party applications, and therefore also for potential malware that might be inadvertently installed by the user. Any change to this predefined policy could widen the attack surface for local privilege escalation attacks.

Figure 7 shows that vendors modify these domains less often, but the numbers are still not negligible (e.g., an average of 95 changes to isolated and untrusted_app were introduced in Android SDK 26).

For each domain, we now present significant dangerous changes made by vendors and discuss their impact on the overall security.

Customization *isolated_app* Domain. We identified a total of 58,776 changes to this context, 1,375 of which are unique. By manually inspecting these modifications, we identified several severe and dangerous cases. For instance, in SDK 19, we found 44 different ROMs that allowed an *isolated_app* process to perform read, write, and ioctl operations directly on kernel drivers. More recently, some devices targeting SDK 23 and 25 changed a rule that allows a process running on this domain to perform open, read, and write operations on application data files. We noticed how this rule is also violating a *neverallow* rule for this process since isolated apps should never directly open application data files themselves.

Customization *untrusted_app* Domain. Across the years, vendors have made 95,577 changes to this context, 4,228 of which are unique. Among these customizations, we found old systems (based on Android 4.4) that allowed an *untrusted_app* process to perform read and ioctl operations directly on kernel drivers. On newer policies, the risks have been reduced by removing the *ioctl* capability, while still allowing the domain to read from kernel components. Another interesting finding, affecting newer devices targeting SDK 27, is related to a rule that allows a process running on this domain to read files containing the device's MAC address. Google is restricting access to this information in many ways, including a `neverallow` rule that prevents this operation [39]. Despite this effort, we still find vendors that nullify these defensive measures by allowing, unintentionally, other applications to access this information.

VII. Related Work

There are mainly two areas of works relevant to this paper: the perils of Android customizations and SELinux policy analysis.

The Perils of Android Customizations. The problems related to Android's customizations have been analyzed in prior studies from different points of view. Aafer et al. [7] have demonstrated how vendors have introduced severe security issues within their systems by modifying security components within the Android framework. Their analysis mainly focused on the framework component, analyzing the various deltas between the XML security configurations of the various ROMs, looking for inconsistencies. Dai et al. [40] instead illustrate how the Android customizations on the framework might be a direct cause of the *patch gapping*, showing how vendors fail to roll out all of the security patches published by Google in a reasonable time. On the same topic, Daniel et al. [41] suggest that another reason updates are not provided in a reasonable time may be due to the large number of entities involved in the supply chain that have to cooperate for the patch to reach the device. Instead, Zhou et al. [6] focus their analysis on vendors' custom drivers, showing how the customized drivers are often sources of security problems and how this problem is not so widespread and common in the drivers offered by the official Android platform. Tian et al. [42] analyzed over 2,000 Android smartphone firmwares across 11 vendors to extract custom and hidden AT commands. To conclude, Wu et al. [43] and Gamba et al. [44] analyzed the customization problems analyzing the pre-installed apps on the device, showing how vendors' customizations introduce several issues for what concerns the security of the system as well as the user privacy.

SELinux Policy Analysis. The analysis of SELinux is a problem that has been addressed in many ways by previous work. Among the first works, Reshetova et al. [45] present SELint, a tool that helps OEMs overcome common challenges and avoid mistakes when writing SELinux policies. The same authors then present another tool, in [46], that can improve policy design and analysis. However, the usage of this tool requires a real device, rooted (or with an engineering build), and it is thus not possible to use it for a large scale analysis. Another relevant work is EASEAndroid [47], which presented an analytic platform for automatic SELinux policy analysis and refinement. This refinement process is automated using semi-supervised learning, and it was trained over millions of SELinux denial logs from real-world devices. Another category of work focuses on the static analysis of this component. Im et al. [48] performed the first measurement on the evolution of SELinux policies available in the official AOSP repository, proposing a new metric to measure the complexity of a given policy. Last, Hernandez et al. [49] proposed BigMac, a new policy analysis framework that works on firmware images and extract attack paths between processes and can help to identify dangerous rules.

Our work significantly differs and complements all these previous ones, for both the type of analysis performed and the components considered, as it is the first to discuss a longitudinal analysis on OEM customizations, their *compliance* aspect, and details about system binaries, libraries, SELinux policies, Android init scripts, and user- and kernel-space hardening techniques.

VIII. Concluding Remarks

In this work, we have focused on the four main components that are responsible for the security of the Android OS: SELinux configurations, system binaries hardening, init scripts, and the Android Linux kernel. However, over the years, due to changes introduced by vendors, these same components have also been a source of severe vulnerabilities. To address this issue, Google has introduced a number of requirements (e.g., CDD) and many automated routines to check for a subset of them (e.g., CTS and VTS). Google also recently re-architectured the Android OS with Project Treble, in an attempt to disentangle vendors' customizations from AOSP.

This paper discusses the first longitudinal, large-scale analysis on a dataset of 2,907 ROMs from 42 different vendors: our results are worrisome. About 20% of ROMs in our dataset did not meet at least one of the security requirements imposed by Google, and among them, to our great surprise, 11 ROMs are branded by Google itself. We observed $190/2{,}396$ (~8%), from 10 different vendors, including Google, contain non-compliant kernels, while $443/1{,}533$ (~29%) violate the SELinux policies requirements. We also show how vendors often configure over-privileged services, introduce dangerous and over-permissive changes to SELinux policy, and do not use compiler-level mitigations. Finally, our results show that these problems did not improve over time and with new versions of Android.

The first insight is that the current set of regulations and checks is clearly insufficient. The same goes for Project Treble: while the engineering effort is admirable, it is clear that its impact was not enough to mitigate the aforementioned problems. The current testing and verification procedure, which is fundamentally based on trust since the tests are executed by the vendors themselves, is not effective at catching problems and it is very often violated. We believe that the vast majority of vendors act in good faith, but some changes suggest intentional attempts to circumvent, for practicality reasons, Google's safety nets (e.g., SELinux `neverallow` rules).

Looking forward, we believe more checks should be automated and that the existing ones should be more accurate. An automatic framework like the one we presented in this paper, which is already able to identify CDD violations that are not detected by the existing test suites, can be used as a basis for these automated analyses.

The second part of our paper shows that there are several areas of customizations that, even if they do not violate any strict requirement, are the root cause for severe security problems. While the CDD is a great starting point, we believe it should be significantly extended to prevent vendors from customizing their ROMs in ways that go against many well-established security practices and principles.

In conclusion, we hope this paper inspires future works and analysis in the important area of OEM customizations, and that Google adopts a stronger stance on OEM customizations that favor performance and ease of development rather than security.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive feedback. We would also like to thank Slasti Mormanti and Betty Sebright for their support and insights.

This work was partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs).

References

[1] S. Cuthbertson, "Sharing what's new in Android Q." https://www.blog.google/products/android/android-q-io/, 2019.

[2] "Issue 2002: Samsung Android multiple interactionless RCEs and other remote access issues in Qmage image codec built into Skia." https://bugs.chromium.org/p/project-zero/issues/detail?id=2002. Accessed December 22, 2020.

[3] "Android compatibility definition document." https://source.android.com/compatibility/cdd. Accessed December 22, 2020.

[4] "Vendor Test Suite (VTS) & Infrastructure." https://source.android.com/compatibility/vts. Accessed December 22, 2020.

[5] "Android ONE." https://www.android.com/one/. Accessed December 22, 2020.

[6] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy*, pp. 409–423, 2014.

[7] 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)*, pp. 1153–1168, 2016.

[8] N. Kralevich, "Honey, i shrunk the attack surface," *Black Hat*, 2017.

[9] "Android Codenames, Tags, and Build Numbers." https://source.android.com/setup/start/build-numbers#source-code-tags-and-builds. Accessed December 22, 2020.

[10] "Android Security: Taming the Complex Ecosystem." https://wisec19.fiu.edu/wp-content/uploads/wisec2019-keynote.pdf. Accessed December 22, 2020.

[11] "Trade Federation Overview." https://source.android.com/devices/tech/test_infra/tradefed. Accessed December 22, 2020.

[12] "Verifying Boot - Rollback protection." https://source.android.com/security/verifiedboot/verified-boot. Accessed December 22, 2020.

[13] "Android 4.4.3 Patch Finally Closes Up An Ancient Vulnerability, Shuts Down Several Serious Security Exploits." https://www.androidpolice.com/2014/06/04/android-4-4-3-patch-finally-closes-ancient-vulnerability-shuts-several-serious-security-exploits/. Accessed December 22, 2020.

[14] "CVE-2018-9488." https://nvd.nist.gov/vuln/detail/CVE-2018-9488. Accessed December 22, 2020.

[15] "Android Init Language." https://android.googlesource.com/platform/system/core/+/master/init/README.md. Accessed December 22, 2020.

[16] "Practical Android Exploitation." http://theroot.ninja/PAE.pdf. Accessed December 22, 2020.

[17] "Linux Kernel Banner." https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/init/version.c. Accessed December 22, 2020.

[18] "vmlinux-to-elf." https://github.com/marin-m/vmlinux-to-elf. Accessed December 22, 2020.

[19] "Factory images for nexus and pixel devices." https://developers.google.com/android/images. Accessed December 22, 2020.

[20] "Firmware file." https://firmwarefile.com/. Accessed December 22, 2020.

[21] "Stock rom." https://www.stockrom.net/. Accessed December 22, 2020.

[22] "Mobile vendor market share worldwide." https://gs.statcounter.com/vendor-market-share/mobile/worldwide/#monthly-201003-202007. Accessed December 22, 2020.

[23] "Loadable Kernel Modules." https://source.android.com/devices/architecture/kernel/loadable-kernel-modules. Accessed December 22, 2020.

[24] "Core Kernel Requirements." https://source.android.com/devices/architecture/kernel/core-kernel-reqs. Accessed December 22, 2020.

[25] "SELinux for Android 8.0." https://source.android.com/security/selinux/images/SELinux_Treble.pdf. Accessed December 22, 2020.

[26] "SELinuxNeverallowTestFrame.py." https://android.googlesource.com/platform/cts/+/refs/heads/master/tools/selinux/SELinuxNeverallowTestFrame.py. Accessed December 22, 2020.

[27] "Hardening the media stack." https://android-developers.googleblog.com/2016/05/hardening-media-stack.html. Accessed December 22, 2020.

[28] "Control Flow Integrity." https://source.android.com/devices/tech/debug/cfi. Accessed December 22, 2020.

[29] "Experts Found a Unicorn in the Heart of Android." https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/. Accessed December 22, 2020.

[30] "Security Enhancements." https://source.android.com/security/enhancements. Accessed December 22, 2020.

[31] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, 2015.

[32] "Add Test to Verify NX is Enabled." https://android-review.googlesource.com/c/platform/cts/+/21776. Accessed December 22, 2020.

[33] "Tcpdump Public CVE List." https://www.tcpdump.org/public-cve-list.txt. Accessed December 22, 2020.

[34] "Tcpdump Common Vulnerabilities." https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=tcpdump. Accessed December 22, 2020.

[35] H. Meng, V. Thing, Y. Cheng, Z. Dai, and L. Zhang, "A survey of Android exploits in the wild," *Computers & Security*, vol. 76, pp. 71–91, 07 2018.

[36] "Android Vulnerabilities: vold asec." https://androidvulnerabilities.org/vulnerabilities/vold_asec. Accessed December 22, 2020.

[37] "Root 4.4.X - Pie for Motorola devices." https://forum.xda-developers.com/moto-x/orig-development/root-4-4-x-pie-motorola-devices-t2771623. Accessed December 22, 2020.

[38] "OATmeal on the Universal Cereal Bus: Exploiting Android phones over USB." https://googleprojectzero.blogspot.com/2018/09/oatmeal-on-universal-cereal-bus.html. Accessed December 22, 2020.

[39] "Android 6.0 Changes - Access to Hardware Identifier." https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-hardware-id. Accessed December 22, 2020.

[40] "BScout: Direct Whole Patch Presence Test for Java Executables," in *29th USENIX Security Symposium (USENIX Security 20)*, (Boston, MA), USENIX Association, Aug. 2020.

[41] D. Thomas, A. Beresford, and A. Rice, "Security metrics for the android ecosystem," pp. 87–98, 10 2015.

[42] D. Tian, G. Hernandez, J. Choi, V. Frost, C. Ruales, K. Butler, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, and M. Grace, "ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 351–366, USENIX Association, 2018.

[43] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," pp. 623–634, 11 2013.

[44] 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)*, pp. 1039–1055, 2020.

[45] E. Reshetova, F. Bonazzi, and N. Asokan, "Selint: An seandroid policy analysis tool," pp. 47–58, 01 2017.

[46] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan, "Characterizing seandroid policies in the wild," pp. 482–489, 01 2016.

[47] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab, "Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 351–366, USENIX Association, Aug. 2015.

[48] B. Im, A. Chen, and D. S. Wallach, "An historical analysis of the seandroid policy evolution," *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[49] D. J. T. Grant Hernandez, 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)*, 2020.

APPENDIX

A. Compile-time Hardening

This section of the appendix is dedicated to a more detailed description of mitigation techniques considered in Section VI-B. For each technique we provide a short description and we explain which artifacts we considered to detect if and ELF file implements it or not. We report a final summary in Table V.

I) Stack Canaries. Stack canaries, introduced in Android 1.5, work by placing a random integer (canary) in memory just before the stack return pointer. In order to overwrite the return pointer (and thus take control of the execution flow), stack-based buffer overflows attacks must also overwrite the canary value. Before a function returns, the stack canary integrity is checked using the function `__stack_chk_fail` (or `__intel_security_cookie` in an alternative implementation), and if it appears to be modified due to an overwrite, the program exits immediately. Thus, we checked for the presence of the aforementioned function among the binary's symbols. We highlight how the stack protector works in two configurations. The first one protects the buffer only if it is greater than a certain size (depending on the architecture), while the second one (named "strong") protects buffers even if they are one byte size. We assumed, as safe assumption, that real world binaries have at least one buffer that can be protected by this compiler defense mechanism. Even though this assumption might sometimes fail, we believe that the numbers of binaries without a buffer to protect is negligible and it is not going to affect the overall results of our measurement.

II) No eXecute (NX). NX marks certain areas of the program as not executable. NX can be implemented both via software or hardware (almost all modern processors uses it). In our analysis, we checked if the `GNU_STACK` segment of the binaries, which tells the system whether the stack should be executable or not.

III) Position Independent Executables (PIE). The code of a PIE binary can be placed into random locations in memory, and it executes properly regardless of its absolute address. PIE works in tandem with Address Space Layout Randomization (ASLR). ASLR randomly arranges the address space positions of key data areas of a process (e.g., the base of the executable, stack, heap, and libraries). If the executable is position independent, the location of the executable code within the process is also randomized, making it more difficult for an attacker to predict target addresses. As of Android 4.0 (SDK 15), the kernel gained support for ASLR, but Android still lacked userspace support. The Android 4.1 (SDK 16) release introduced support for full ASLR by enabling heap randomization and adding linker support for PIE. Android 5 (SDK 21) is the latest step forwards, as non-PIE executable support was dropped, and all processes now have full ASLR. The third graph of Figure 3 reflects the history flow: the growth since its introduction (SDK 16), and a second increase when the non-PIE executable support was dropped in (SDK 21). PIE is the security enhancement with the greater adoption because, after Android 5, the linker does not load non-PIE executables. A PIE ELF file is of the type `ET_DYN`, and its `.dynamic` section contains the `DT_DEBUG` tag.

IV) Full Relocation Read-Only (RELRO). A dynamically linked ELF binary uses a look-up table called Global Offset Table (GOT) to dynamically resolve functions located in shared libraries. The dynamic linker defers function-call resolution to the point when the function is called rather than at load time. This technique is known as lazy binding, and it needs that the GOT lives in a predefined place and is writable. Hence, if an attacker finds a bug allowing them to write a few bytes (as many as the length of a valid address), they can overwrite a GOT entry. If a GOT entry is properly overwritten, the attacker can hijack a library call to their malicious code. However, the immediate binding is a valid countermeasure: the linker can resolve all the dynamically linked functions at the beginning of execution and make the GOT read-only. This mitigation is known as *Full RELRO*, and it appears in the SE of Android 4.1. If an ELF implements the Full RELRO, it has the `GNU_RELRO` segment and its `.dynamic` section contains the `DT_BIND_NOW` tag. The `GNU_RELRO` segment indicates the memory region which should be made read-only after relocation is done, while the `.dynamic` section contains an array of tags. The `DT_BIND_NOW` tag indicates the linker that all relocations must be processed before returning control to the program, i.e., using immediate binding.

V) `FORTIFY_SOURCE`. This is a macro (available in both GCC and Clang) that provides lightweight checks for detecting buffer overflows in various dangerous functions, like `memcpy`. Some of the checks can be performed at compile time while other checks take place at run-time and result in a run-time error if the check fails. `FORTIFY_SOURCE` works in two phases: first, it tries to computes the number of bytes of the destination buffer used in a dangerous function. If it succeed, it replaces the dangerous functions with their secure `_chk` counterpart (e.g., `memcpy` $\rightarrow$ `__memcpy_chk`) adding as new argument the size of the buffer. If an attacker tries to copy more bytes, the `_chk` function detects the overflow and the program's execution is stopped. If the first step fails, the compiler cannot harden a function (e.g., it might fail with dynamic memory allocated buffers). For dynamically linked executables, the `libc` contains the implementation of the `_chk` functions. Therefore, we first checked whether the `libc` supports `FORTIFY_SOURCE`, that is, the `libc` contains at least one `_chk` function among its exported symbols. If yes, for each binary, we check if it contains at least one `_chk` function among its imported symbols.

VI) `setuid/setgid`. These are a special type of file permissions that permit users to run specific executables with temporarily elevated privileges, to perform a specific task.

B. Kernel Configuration Mappings

Table VI contains, for each of the configurations defined in the CDD, the correspondent *Kernel Symbol* introduced while enabling the specific configuration. We noticed how, some kernel configurations might map to the same symbol, while other kernel configurations, depending on the version of the kernel, might change the symbol used. For these configurations, we rely on regular expressions to identify valid symbols. For 3 configurations, we were not able to identify any symbol for the mapping.

TABLE V
MITIGATION TECHNIQUES

SDK	Version	Enhancement	Artifact
3	1.5	Stack Canaries	`__stack_chk_fail` function symbol, or `__intel_security_cookie` function symbol
9	2.3	No eXecute (NX)	`GNU_STACK` segment `RW-`
16	4.1	Position Independent Executables (PIE)	ELF type `ET_DYN`, and `.dynamic` section with `DT_DEBUG` tag
		Full Read-only Relocations (RELRO)	`GNU_RELRO` segment, and `.dynamic` section with `DT_BIND_NOW` tag
17	4.2	FORTIFY_SOURCE	`*_chk` function symbols, and `*_chk` exported function in libc
18	4.3	No setuid/setgid programs	setuid/setguid bit in file's permission

TABLE VI
MAPPINGS KERNEL CONFIGURATION TO ELF SYMBOLS

Kernel Configuration	Kernel Symbol
CONFIG_SECURITY_SELINUX	Symbol contains `selinux`
CONFIG_SECCOMP	Symbol contains `seccomp`
CONFIG_CC_STACKPROTECTOR_REGULAR	`__stack_chk_fail`
CONFIG_CC_STACKPROTECTOR_STRONG	`__stack_chk_guard`
CONFIG_DEBUG_RODATA	`rodata_enabled, set_debug_rodata, __setup_set_debug_rodata`
CONFIG_STRICT_KERNEL_RWX	`mark_readonly`
CONFIG_HARDENED_USERCOPY	`__check_heap_object, __check_object_size`
CONFIG_ARM64_SW_TTBR0_PAN	`reserved_ttbr0`
CONFIG_RANDOMIZE_BASE	Symbol contains `kaslr`
CONFIG_PAGE_TABLE_ISOLATION	`tlb_flush_mmu_tlbonly`
CONFIG_UNMAP_KERNEL_AT_EL0	`__initcall_map_entry_trampoline1`
CONFIG_HARDEN_BRANCH_PREDICTOR	`__nospectre_v2`
CONFIG_SHADOW_CALL_STACK	`init_shadow_call_stack`
CONFIG_SECURITY_DMESG_RESTRICT	`dmesg_restrict`
CONFIG_SECURITY_KPTR_RESTRICT	`kptr_restrict`
CONFIG_ARM64_PAN	`cpu_enable_pan`
CONFIG_CFI_CLANG	Symbol contains `__cfi_*`
CONFIG_DEFAULT_MMAP_MIN_ADDR	No symbol mapping found, variable
CONFIG_CPU_SW_DOMAIN_PAN	No symbol mapping, inline assembly
CONFIG_LTO_CLANG	No symbol mapping found