Automating the Quantitative Analysis of Reproducibility for Build Artifacts derived from the Android Open Source Project

Manuel Pöll
manuel.poell.mail@gmail.com
Johannes Kepler University Linz
Institute of Networks and Security
Linz, Austria

Michael Roland
michael.roland@ins.jku.at
Johannes Kepler University Linz
Institute of Networks and Security
Linz, Austria

ABSTRACT

This work proposes a modular automation toolchain to analyze current state and over-time changes of reproducibility of build artifacts derived from the Android Open Source Project (AOSP). While perfect bit-by-bit equality of binary artifacts would be a desirable goal to permit independent verification if binary build artifacts really are the result of building a specific state of source code, this form of reproducibility is often not (yet) achievable in practice. Certain complexities in the Android ecosystem make assessment of production firmware images particularly difficult. To overcome this, we introduce "accountable builds" as a form of reproducibility that allows for legitimate deviations from 100 percent bit-by-bit equality. Using our framework that builds AOSP in its native build system, automatically compares artifacts, and computes difference scores, we perform a detailed analysis of differences, identify typical accountable changes, and analyze current major issues leading to non-reproducibility and non-accountability. We find that pure AOSP itself builds mostly reproducible and that Project Treble helped through its separation of concerns. However, we also discover that Google's published firmware images deviate from the claimed codebase (partially due to side-effects of Project Mainline).

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; *Software configuration management and version control systems*; Development frameworks and environments; Open source model; • **Security and privacy** → **Mobile platform security**; *Intrusion/anomaly detection and malware mitigation.*

KEYWORDS

Deterministic build, Over-time measurement comparison

ACM Reference Format:
Manuel Pöll and Michael Roland. 2022. Automating the Quantitative Analysis of Reproducibility for Build Artifacts derived from the Android Open Source Project. In *WiSec '22: 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks, May 16–19, 2022, San Antonio, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3507657.3528537

WiSec '22, May 16–19, 2022, San Antonio, TX, USA

ACM ISBN 978-1-4503-9216-7/22/05…$15.00
https://doi.org/10.1145/3507657.3528537

1 INTRODUCTION

Android is the single most widely adopted mobile operating system in use today (with a market share of more than 70 percent as of 2021 and continuously held for the past years [45]). Its core is based on open source software, the Android Open Source Project (AOSP). AOSP contains all the core components of a fully-functioning mobile operating system distribution on which mobile handset manufacturers and providers of after-market firmware (e.g. the LineageOS distribution) base their own customizations and additions.

Open source software in general is increasingly influential in numerous application domains. Originating in the Linux community [37] and, for a long time, a very common form of software development by mainly hobbyists and academia, it has found widespread adoption in enterprise environments of all sizes, including large multinational enterprises [39]. Open source software is seen as both a potential benefit and also a potential threat to software quality and security [39]: While enterprise open source software is perceived as a chance to increase quality and security, and to benefit from latest innovations and code-reuse, particularly community-driven open source software is often considered a risk in terms of security and quality. On the one hand, a key motivation behind open source software is the establishment of trust. Any interested party may freely inspect the source code and may assure itself that a program is free of malicious components. On the other hand, vulnerabilities in open source libraries and particularly the issues associated with managing them are seen as problematic, cf. [15, 34, 38]. Recent attacks on software dependencies and the software supply chain in general (e.g. Dependency Confusion [9] and SolarWinds/SUNSPOT/SUPERNOVA/SUNBURST [14]) suggest that not only open source libraries but the whole software supply chain management poses a significant security risk. This risk, introduced by the software supply chain, is not new, cf. [19, 32, 36, 44]. Already in 1974, Karger and Schell [30] mentioned the threat of code injection by malicious compilers. Practically demonstrating this issue 10 years later, Thompson [47] concluded that we rely on putting ultimate trust into all the intermediate steps (and involved parties) on the path between the source code and the resulting executable binary files. As a result, even if open source software permits self-assurance about non-maliciousness of the source code, this trust does not automatically propagate to binary artifacts.

One approach to reducing the attack surface imposed by maliciously acting software providers and parties involved in the build process is the concept of reproducible builds. Reproducible builds aim to make the software build process deterministic. Any interested party can inspect the source code, configurations and build toolchain, can create their own binary artifacts, and can use these

to verify that they match the officially provided artifacts bit-by-bit (cf. the concept of *diverse double-compiling* attributed to Henry Spencer by Wheeler [49]). While the vast majority of users do not compile their software themselves, they can still benefit from independent verification of the correspondence between source code and resulting binary artifacts. Even if this correspondence is verified by only a small number of independent parties, this provides additional trust to the overall user base. Also, if users (or organizations) would be willing to compile their own software, this is sometimes infeasible for device firmware due to secure boot environments rejecting self-signed firmware binaries. Therefore, reproducible builds are a logical next step for open source projects to extend the trust from their source code to the final binary artifacts.

Debian [17] is probably among the most prominent projects that aim for reproducibility. With the Reproducible Builds project [42], a whole initiative to drive reproducible builds has been founded. Besides supply chain security, reproducible builds have also proven as an important measure to achieve reproducibility of scientific results (e.g. in high-performance computing with GNU Guix [13]).

In the segment of mobile software and operating systems, there are a few security-focused, AOSP-based projects that claim reproducibility (e.g. GrapheneOS [24], CopperheadOS [12], and Guardian Project [25]). Moreover, F-Droid is a dedicated app catalog for Free and Open Source Software (FOSS) that supports reproducible builds for Android apps [46]. While AOSP does not aim for reproducibility with their current build system, there are a few projects trying to make AOSP reproducible with improved build systems (e.g. AOSP Build [26] and robotnix [20]). However, using such a customized build environment only allows to compare results based on these alternative environments. Consequently, this does not help towards the goal of verifying the correspondence between the AOSP source code and the actual binaries already shipped by mobile handset manufacturers and (after-market) firmware providers. Even with the announced migration to the Bazel build system [28], AOSP has a long way ahead of becoming fully reproducible.

It is, therefore, interesting to quantitatively and qualitatively assess to what extent AOSP already provides reproducibility and how this reproducibility allows to establish trust that published firmware images stem from a specific, untainted version of the official AOSP source code. Moreover, where reproducibility is not achieved, it is interesting to identify the root causes of differences.

Beside the official build system not (yet) aiming for full reproducibility, the heterogeneous Android ecosystem poses several additional challenges: The tremendous amount of different build targets results in various generic and device-specific firmware images and image formats. Moreover, AOSP only provides the core components, while OEMs (including Google) enrich the resulting firmware images with closed-source functionality (like the Google Apps) creating expected discrepancies between the overall binary artifacts (even if the core components are untainted). Finally, firmware images are signed with keys that must be kept secret. While this is well-studied and does not pose a significant issue, firmware images themselves contain signed binaries leading to multiple layers of signatures.

In this paper, we measure the state of reproducibility of AOSP against publicly available reference images such as the generic system images (GSI). Further, we analyze how this reproducibility can be used to assess the trustworthiness of claims that specific device firmware images stem from a certain, untainted codebase. We create an automated toolchain for running AOSP builds in their native build system, for automating artifact comparison, and for analysis of remaining differences. This permits a quantitative assessment of changes in reproducibility over time and an independent verification of the mapping between official source code and official binary artifacts. As we found that there are certain complexities in the Android ecosystem that make perfect reproducibility unpractical, particularly in the assessment of production firmware images, we introduce "accountable builds" as a form of reproducibility that allows for legitimate deviations from 100 percent bit-by-bit equality. Finally, we analyze the current major issues of AOSP leading to non-reproducibility and non-accountability.

2 REPRODUCIBLE BUILDS

The gap between programs in their source code form and their compiled binary form leaves room for manipulation. There is no intrinsic guarantee that artifacts distributed by someone, claiming these stem from some unmodified source code, really are the result of building that specific source code without any (potentially malicious) modifications. An obvious solution to establish your own trust in the mapping between source code and binary form, would be to compile all software yourself—at least for open source software and under the assumption that the build toolchain itself is benign. However, that is not a practical solution for most users. Reproducible builds are a way towards bridging this gap, as they allow verification of that mapping for existing binary artifacts.

The typical textbook definition of *reproducibility* mandates exact bit-by-bit equality between all the artifacts produced from the same source code in the same build environment using the same build instructions [8, 31, 40]. Based on this requirement, verifying if two artifacts stem from the same reproducibly building source code is straight forward: They can simply be compared bit-by-bit. This makes automation of the comparison trivial and also eases independent validation: A party that wants to contribute results of their own compilation does not need to publish full binary artifacts, but can simply publish a cryptographic checksum over the generated untainted artifacts to provide a trust anchor for independent verification by others. Similarly, users can build and use their own binary artifacts from source and can then verify them against a simple hash value provided by the publisher of the source code without the need to obtain the whole pre-built artifacts.

2.1 Deterministic Build System

A make-or-break factor for reproducibility is the build environment. This includes not only all the different tools used as part of the build process, but also the state and configuration that these tools are used in. Particularly for huge projects like Android, there is a diverse set of tools involved in the build process of monolithic binary artifacts (i.e. the final firmware images). These include several compiler toolchains for compiling and linking source code in different languages, tools to assemble application and firmware package files, and build automation tooling (e.g. Soong and Make) that combines all the other tools into one huge build system.

An important property that all the involved build systems need to contribute is determinism. A *deterministic build system* ensures

that stable inputs (source code, configuration) lead to stable outputs (identical artifacts), while minimizing effects of the environment a build is performed in [7, 11]. Even in a deterministic build system, non-stable inputs from configuration and environmental differences will cause variations in outputs. Hence, uncontrollable parameters in the build environment prevent determinism.

For instance, artifacts may include timestamps and other metadata that describe the build environment itself (e.g. begin/end of compilation, file metadata in file system images and other containers, absolute paths, hostnames, usernames). Best-practice for a deterministic build system would be to completely get rid of all such information [10]. While often only used for convenience, existing specifications of data formats (and tools based on them) may mandate presence of such metadata though. Hence, a deterministic build system needs to apply an appropriate normalization strategy to such metadata. Data that is not essential for the functionality of an artifact may simply be stripped in post-processing or not be added at all. If omission is not desirable, values must be derived in a deterministic fashion, e.g. by solely relying on data from source code management (version control system). Again, such values may then be used by the build tools directly or patched through a post-processing step.

2.2 Accountable Builds

In practice, it is not always possible to achieve full bit-by-bit equality. For those cases, it is important to distinguish between accountable differences, which have their origin in a specific, explainable deficiency that can be held accountable for, and unexplainable differences. If two builds of the same build target differ only by fully explainable differences, we refer to this as an *accountable build*. While an accountable build is weaker than a reproducible one, it is a good step towards a fully reproducible build system.

One reason for such differences can be that (parts of) the build system are not yet deterministic. This is usually fixable by patching and updating the build tooling. Nevertheless, it is important to measure the degree of reproducibility, and to classify issues into explainable and unexplainable differences, even when a build system does not yet provide full determinism.

Another reason for failing full reproducibility is parts of the build environment that are expected to differ between the official source of binary artifacts and someone who tries to reproduce building them. This is, for instance, the case when verifying the claim that some firmware image published by a third party stems from a specific source code version. One such unavoidable difference may come from code signing. Obviously, someone trying to reproduce a build must not have access to the official signing keys used for the published firmware images. Hence, resulting signatures generated for artifacts must be different.

As long as one considers the signature (and potentially associated certificate chains) as auxiliary metadata and not part of the binary artifact itself, this does not threaten reproducibility and, consequently, should not impact the trust gained from an independent verification of such build artifacts. However, while it would be simple to strip a single signature before comparison for a single executable, this task is a lot more complex when it comes to whole operating system distributions bundled into firmware images.

Looking specifically into Android, it is not as simple as a single detachable signature and a firmware image. Instead, each application package (Android Package, APK) and, lately, also each upgradable system module package (Android Pony EXpress, APEX) is signed individually. Signing such package files means that certificate files are added and that manifest files listing packaged files with their signature values change. Such changes are, again, accountable differences and should, in most cases, be automatically excluded from a build comparison. However, since code signing is an integral part of the Android permission system [33], changes of signing keys (and their associated certificates and signatures) also propagate to other areas.

Since critical permissions are tied to signatures, public keys and certificates can be part of permission policy definitions (e.g. embedded in SELinux policy files). Simply stripping all public keys and certificates from a policy before comparison may result in undiscovered security issues like an additional key injected by a malicious party to gain permissions based on their own signing key.

Also, update mechanisms for application packages and the firmware image itself are tied to signatures. For APK and APEX files, this has an interesting side-effect seemingly introduced by Project Mainline [43]: Certain applications available as part of AOSP are included into firmware images with a different package name. Specifically the AOSP package name prefix "com.android" is changed to the Google prefix "com.google.android" while supposedly keeping the source code identical. This seems to have the practical reason that Google can ship those packages with their own signature updatable through Play Store while keeping versions directly built from AOSP unaffected and conflict-free. Here, conflict-free means that updates published by Google through Play Store would not be identified as updates for AOSP versions (due to the different package name) and would, consequently, not result in signature-mismatches. Therefore, as long as the rest of the binary artifact (except for the signature/certificate and the package name) match, such a deviation may still be considered an accountable difference.

Besides Google's Mainline effort, firmware images for Android devices are usually assembled from AOSP and other components. Such additional components are expected in the Android ecosystem since device manufacturers often want to (or even have to) rely on closed-source components and also want to enhance beyond the core AOSP functionality. A number of these components is provided by chipset vendors and original design manufacturers to provide platform-specific low-level device drivers and software to install/update peripheral firmware. Since these components are often not offered as FOSS, device OEMs can, at best, provide them as standalone pre-built binary blobs for reproducing their device firmware. Another share is made up by components shipped by the OEM to enrich user-experience and branding over plain AOSP. Such additions are expected since manufacturers want to offer a value-gain over other AOSP-based devices. Even though functionality embedded in such binaries cannot be verified based on public source code, we consider such additions acceptable in an accountable build as long as they are clearly distinct from AOSP, do not modify the AOSP codebase itself, and do not change the security and privacy guarantees provided by the platform. Nevertheless, we would prefer these components to be reproducible FOSS as well.

3 RELATED WORK

There has been extensive work in defining the requirements for reproducible builds. Specifically, the Reproducible Builds project [7, 11] put a significant effort towards defining reproducibility and how to create a deterministic build environment. While their focus lies on Debian, there are similar efforts for other FOSS projects.

There are two actively developed projects (AOSP Build and robotnix) that create new or enhanced build environments focused on improving reproducibly building AOSP. Our automation framework differs from their approach in that we explicitly aim to measure reproducibility and comparability based on the unmodified build system that is also used for official firmware builds. Nevertheless, we have embedded the robotnix build flow into our framework (though we consider that out-of-scope for this paper).

With their 2020 announcement to migrate to the Bazel build system [28], Google made "*correct and reproducible (hermetic) AOSP builds*" an explicit goal for AOSP for the first time. Our framework can help with continuously monitoring that effort by comparing reproducibility metrics before, during, and after that transition.

The tool diffoscope [41] (from the Reproducible Builds project) performs recursive, context-aware comparison ("diff-ing") of archive files and generates reports in the form of line-by-line differences. Such reports form the basis for our metrics.

With regard to measuring reproducibility, Ren et al. [40] create RepLoc, a framework to automatically localize reproducibility issues. Their work focuses on (but is not exclusively limited to) building packages in the Debian distribution. RepLoc leverages diffoscope to identify differences between two build instances of the same source code. Based on these difference reports and on domain-specific knowledge about common reproducibility issues in Debian, they estimate the source files most likely responsible for breaking reproducibility. In contrast, the goal of our analysis framework is to give a quantitative estimate of improvements in reproducibility/comparability over time while specifically focusing on the special aspects of the Android ecosystem.

Even before the Reproducible Builds project and the efforts in Debian, reproducibility had been understood as an important property in software configuration management and build systems. For instance, Heydon et al. [27] mention repeatable builds as a valuable benefit of the Vesta SCM system. Muniswamy-Reddy et al. [35] discuss data provenance as a source for automated generation of build steps to reproduce artifacts.

With regard to software supply chain security, there is various orthogonal work. For instance, Jämthagen et al. [29] present a novel approach to create hidden functionality at the source code level. They specifically target false trust in deterministic builds, as any malicious functionality that makes it into the source code repository while not being identified as such, would (obviously) not be discoverable through independent deterministic builds.

de Carné de Carnavalet and Mannan [16] analyze verifiability of official TrueCrypt binaries by manually recreating a build environment that potentially resembles the original through trial-and-error.

Torres-Arias et al. [48] develop a framework to cryptographically guarantee continuous integrity of the whole software supply chain from source code to artifacts at the end user. We partially share this goal as we try to establish trust that a firmware image published by a vendor stems from a claimed source code version. However, we do not require the parties involved in the build process to collaborate by providing authenticated metadata on their actions and only rely on independently rebuilding the binary artifacts. This is similar to Gitian [18], which aims at providing a deterministic build environment to create trusted binaries that are verified by multiple independent builders.

4 AUTOMATING THE ANALYSIS

Building AOSP involves several steps, starting with the preparation of a build environment, checkout of source code, and finally the actual build process [1]. We created a wrapper around the Android build system to automate the entire process of setting up the build environment, fetching and building AOSP, and comparing artifacts with reference builds. This permits us to assess the current state of reproducibility when matching the official AOSP build instructions as closely as possible, to easily reproduce such an analysis, and to perform a long-term analysis of rebuilt firmware images against officially provided firmware images over time. We make the whole environment, which we call "***S**imple **O**pinionated **A**OSP builds by an external **P**arty*" (*SOAP*), publicly available at https://github.com/mobilesec/reproducible-builds-aosp.

4.1 Tooling and Architecture

A majority of the steps of the official AOSP build instructions [1] are commands meant for execution in a Bash compatible shell on a Linux host system (specifically Ubuntu). Since our environment should be as close as possible to the official instructions, we opted to use shell scripts for implementing our automation environment.

We created an automation environment consisting of several small, composable shell scripts. Each script automates a specific smaller task (i.e. a step in the aforementioned process) creating a modularized framework that allows for simple exchange of individual components. A top-level script executes all steps in proper sequence. This top-level script also allows for continued, automated long-term rebuild comparison (we use Jenkins). The whole automation framework is packaged in a Docker container based on Ubuntu 18.04 (or Ubuntu 14.04 for older build targets).

After a successful build, our framework performs an automated analysis to compare the (re-)built artifacts to reference builds, and derives quantitative metrics and detailed difference reports. At its core, this process relies on diffoscope to perform recursive diffing of the firmware packages. This includes unpacking of various archive formats and transformation of binary formats into human-readable representations for visualizing differences. In addition, our framework performs a set of pre-processing steps to eliminate certain expected, accountable differences and to handle Android-specific container formats that diffoscope does not support yet. A set of post-processing steps on the output of diffoscope derives quantitative metrics from difference reports that form the basis for analysis of over-time reproducibility and for continuous assessment of the trustworthiness of claims that official firmware image releases are built from specific source code versions. This analysis process is packaged in a separate Docker container for clean isolation from the build environment.

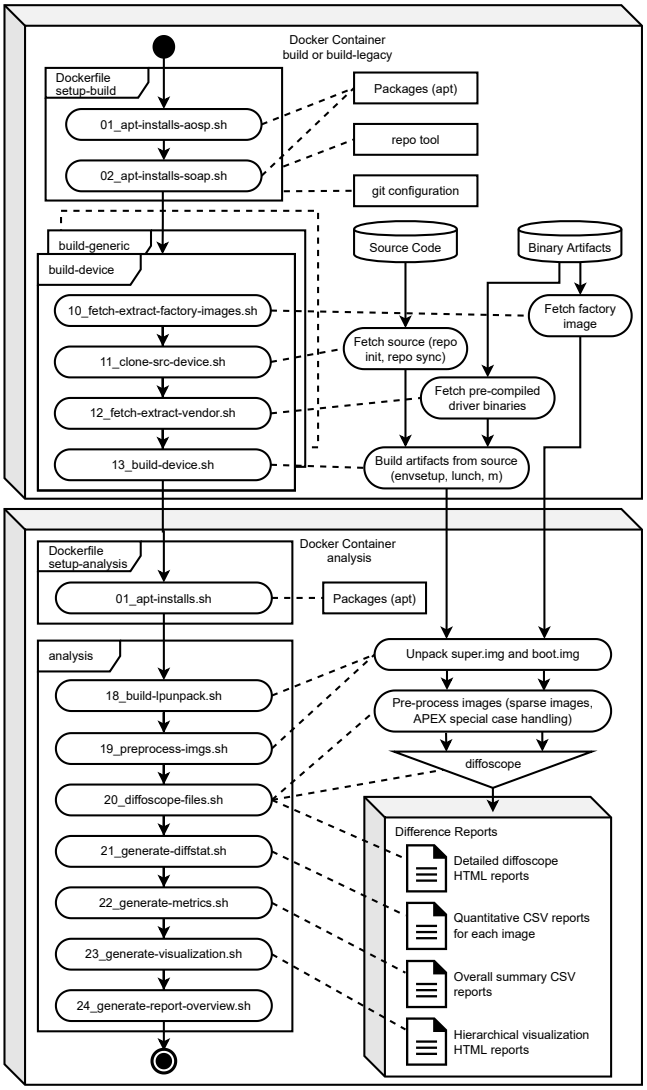

Figure 1: Automation framework architecture.

The overall architecture is depicted in Fig. 1. The left side of this figure shows the modules of our framework split into a build stage and an analysis stage (each packaged in a Docker container). The right side shows the corresponding automation steps for setup/preparation, fetching of source code and binary reference artifacts, build and comparison stages (including their pre- and post-processing), and the final comparison outputs.

Our environment currently performs automated comparison of reference builds sourced from (1) the generic system images (GSI, cf. [3]) as provided by the Android Continuous Integration (CI) dashboard [21], and (2) the official factory images for Nexus and Pixel devices [23] provided by Google. These sources have been chosen under the following assumptions:

(1) GSI builds are, by definition [3], "*[...] considered a pure Android implementation with unmodified Android Open Source Project (AOSP) code [...]*" Therefore, they are well-suited as ground truth for comparison against our rebuilds in order to measure the degree of reproducibility of pure AOSP.

(2) Pixel (formerly Nexus) are considered Google's flagship devices. They have a timely update schedule and cover the latest innovations in AOSP. A documented, exact mapping [6] between build IDs and source code Git tags allows us to infer an exact source revision for any given Pixel/Nexus factory image [23]. Moreover, AOSP contains the necessary build configuration (i.e. device-specific build targets) for these devices. Google also provides ready-made binary blobs of proprietary, closed-source components of their firmware that are, due to legal reasons, not part of AOSP.

    However, we acknowledge that device-specific build targets in AOSP ("`aosp_<codename>`") are not exactly matching the build targets used by Google for the official factory images ("`<codename>`"). As the latter are not publicly available, we are limited to a best-effort comparison under the expectation that these build targets are similar enough to warrant comparability, even though the build instruction are (likely) not identical. While this does not permit to measure reproducibility in a classical sense, we expect them to be sufficiently close for assessing the mapping between untainted AOSP source code and OEM firmware images.

While out-of-scope for this paper, our modular environment could easily be adapted to analyze other build targets from other sources as long as they rely on the AOSP build system, the necessary build configuration is provided, and a well-defined mapping between firmware images and source code exists. Even other build systems or changes to the current AOSP build system could be integrated at the price of the additional effort for integrating deviating steps in the form of new automation modules.

4.2 Deviations from AOSP Build Instructions

Although our goal was to fully abide by the official build instructions [1], our tests on Ubuntu 18.04 and Debian 10 revealed additional dependencies for the AOSP build environment. The tool "repo" (used to fetch the AOSP source code repository) has a dependency on Python 2. Even though the tool is written for Python 3, the shebang (`#!/usr/bin/env python`) of repo exclusively resolves to Python 2 on Ubuntu (as the `python` command is not provided by Python 3 packages). Once started in Python 2, repo immediately restarts itself in a Python 3 interpreter. As Python 2 is no longer installed by default on Ubuntu 18.04, we install the APT package `python` in addition to the official list of dependencies. Moreover, older Android versions (8.0 and below) do not ship their own JDK and building them requires the installation of OpenJDK 8. Furthermore, a GSI build uncovered xxd (a hexdump tool) as an additional dependency not mentioned by the official build instructions.

While readily installed on Ubuntu 18.04, we also install the packages `rsync` and `libncurses5`. This allows our toolchain to also run on Debian 10 while not having any impact on systems where these packages already ship pre-installed.

The AOSP build system also uses several undocumented environment variables (e.g. `BUILD_DATETIME`) that allow to make the build output more deterministic. We rely on these variables and fill them with corresponding values from the official firmware images.

4.3 Challenges and Potential Solutions

During the implementation of our automation tooling, we encountered challenges inherent to the AOSP build process and related to the use of diffoscope for analysis that required unique solutions.

4.3.1 Sparse Images. File system images can become quite large, especially due to zero-padding (for alignment or to match partition sizes) or due to duplicate data blocks (because files contain identical code or data structures). In order to mitigate problems when transferring or flashing such large image files, Android has its own sparse image format [4] used for Google's factory images [23].

As of today, diffoscope (version 202) is not capable of handling such Android sparse images directly. Therefore, we opted to convert these images to regular file system images before passing them on to diffoscope for further processing. For this, we rely on the tool `simg2img` provided by AOSP. This tool is included in AOSP in source code form and is built during the AOSP build process.

4.3.2 Project Mainline APEX Files. Starting with Android 10, as part of their Project Mainline effort, Google ships certain components of AOSP in the form of easily and independently upgradable packages. While the source code of these APEX package files is (or should be) part of AOSP, a side effect of ensuring seamless and conflict-free upgradability is that the package prefix of most APEX files changes from "com.android" (in AOSP) to "com.google.android" (in device factory images). Since this also changes their file names, diffoscope no longer considers these to be the same (or comparable) files across the compared artifacts.

We consider the pure change of package name as an accountable difference (cf. section 2.2) as long as this is the only modification and both packages are built from the same source code. Therefore, we want to get a similarity score despite the different package names. To achieve this, we opted to exclude APEX files from the analysis of the overall firmware image and perform an additional comparison step where we extract these files from the outer image, unify their names to the prefix "com.android", and then pass them on to diffoscope for separate analysis.

4.3.3 Dynamic Partitions. With Android 10, Google introduced *super images* containing dynamic partitions. Such a partition may bundle any of the read-only mounted partitions used from within the Android/Linux system. This allows for seamless changes in the device partition layout through over-the-air updates [2]. A `super.img` encapsulates the images and partitioning information for several other partitions, most notably the system partition (containing the Android framework; otherwise in a file `system.img`), the vendor partition (containing components not publishable with AOSP; otherwise in `vendor.img`), and the product partition (containing OEM-specific components to make the vendor partition consist of only SoC-specific components; otherwise in `product.img`).

The GSI build targets output such a super image. Therefore, the Android CI dashboard offers only the `super.img` build artifact while omitting the classical `system.img` and `vendor.img` files. Device builds currently use the separate partitions approach. As we want to maintain easily comparable results between our evaluation of GSI builds and device builds, decomposing the super image into the individual partition images is necessary. AOSP provides the tool lpunpack for this task. This tool is not automatically built from the standard build targets and needs to be explicitly built using the command "`mm -j $(nproc) lpunpack`".

4.4 Trade-Offs

4.4.1 Embedded Signatures and Certificates in Device Builds. A core concept of the Android platform security model [33] is that not only whole partitions, but each individual application component is digitally signed. Consequently, each APK file and each APEX file also has its own signature that is added at the end of the build process. Obviously, the secret signing keys must not be shared. Therefore, the certificates and signatures embedded into published firmware images must differ from those embedded into our own builds. While this is essential for device builds, GSI builds follow a different strategy. As they are not considered production software, they are signed with public test keys (private key published) instead.

For device build comparison, deviating signatures are considered an accountable difference. Hence, it is necessary to exclude the relevant certificate and signature files from comparison. For APK and APEX files, the signature is located in `META-INF/CERT.RSA`. APEX files additionally contain the signer public key in a file (`apex_pubkey`). These are simply excluded from our difference reports. The same applies to certificates for the platform signing key and for over-the-air updating that are directly embedded into the file system of the system partition. These are `releasekey.x509.pem` (or `testkey.x509.pem` for our builds) for platform signing and `update-payload-key.pub.pem` for OTA updates.

Besides the signature itself, the signing scheme of APK/APEX files also embeds a digest of the signer certificate into the signed Java archive manifest files (`META-INF/CERT.SF` and `META-INF/MANIFEST.MF`). Since we consider the remaining portions of these files (digests over all signed files within the APK/APEX) important for reproducibility, we tolerate these changes to propagate into difference reports but eliminate them from quantitative change metrics for device builds. The same applies to the comparison of ZIP archive metadata for these files performed by diffoscope through the tool zipinfo.

Another difference due to signing is related to SELinux. The policies contain permissions based on platform signatures. As a consequence, the file `system/etc/selinux/plat_mac_permissions.xml` is accountably different. Due to the sensitivity of this file and the possibility to miss additional certificates injected into it, we opted to tolerate this as a false positive to also propagate into our reports while eliminating the expected number of changes from quantitative metrics.

4.4.2 Noise Reduction in ELF Binary Comparison. diffoscope performs a rich comparison of ELF (Executable and Linkable Format) files, showing differences in headers and individual sections through the help of the tools readelf and objdump. Resulting diff-reports can become unproportionally large even for only minor changes as even a small offset shift causes the entire compared hexdumps to show as different. Through manual analysis, we found that, in all cases, differences in ELF files also show up as differences in the ELF headers (mainly as changes in offset and size fields). Based on this observation, we have opted to exclude detailed comparison of ELF files (symbol table, relocation table, disassembly, and raw section hexdumps) and solely rely on header comparison. While

there is, admittedly, a small possibility that tiny changes could slip through, we consider this an acceptable trade-off to significantly reduce noise in analysis reports.

4.5 Output Format

An Android installation consists of several partitions that are written to flash storage. Each of these partitions contains file systems or data for a specific purpose, such as the main system partition, partitions for SoC, OEM and product specific files, a boot image containing the Linux kernel together with an initial ramdisk, etc. Therefore, file system (or raw) images of these partitions are the binary artifacts of the AOSP build process.

After the build process finishes, our analysis framework performs an analysis (comparison and pre-/post-processing) and stores the resulting difference reports. Specifically, we create a list of artifacts for both our own builds and the pre-built images that we use as our compare-targets. We consider only those artifacts that exist in the pre-built images for further inspection. For each of these files we perform a recursive difference analysis through diffoscope and generate several reports:

- A detailed HTML report showing difference listings for all artifacts that exhibit variations.
- CSV reports summarizing the number of change lines for all artifacts via the tool diffstat.
- In a post-processing step, these CSV reports are cleaned from expected accountable changes that we deliberately let slip through into the difference reports (cf. section 4.4), which we refer to as "diff score".
- The individual CSV reports of each artifact are further aggregated into a single change summary report. Besides accumulated change lines, the individual CSV reports are also used as the basis to calculate a "weight score" (as relative amount of changes, see section 5).
- The final quantitative metrics are also visualized in a hierarchical treemap for improved navigation through detailed difference reports (not further used in this paper though).

5 QUANTITATIVE CHANGES OVER TIME

An assessment of reproducibility over time requires stable metrics that quantify the state of differences. Existing tools primarily target localizing the source of differences (cf. [40]). For instance, diffoscope provides (indirectly through unified diffs) a change score in terms of change lines between two artifacts. After elimination of expected changes as discussed in section 4.4, we refer to this as "diff score" (DS). These change lines are not necessarily the result of directly comparing two files, as the notion of a "line" usually only applies to text files and not to other binary artifacts. Change lines provided by diffoscope may, instead, be the result of comparing high-level reports generated by analysis tools that translate or summarize binary artifacts to abstract information (e.g. zipinfo, apktool, readelf). As a result, this metric does not necessarily have a relationship to the file size of individual components or the amount of change lines reported for other artifact components. Nevertheless, many accountable changes (such as side-effects of the signing scheme) have a stable impact on change line reports, and can easily be observed in and excluded from them.

Since any tiny difference in the binary artifacts could mean that (potentially malicious) functionality was added, it does not make much sense to reflect maliciousness in a quantitative difference metric. Another meaningful quantity that a reproducibility score could reflect is the relative amount of artifact bytes affected by changes. In a first attempt towards creating a reproducibility score that not only allows comparison and localization of changes within a single artifact, but also allows to compare changes in reproducibility of a source code repository and a specific build target over time, we define a "weight score" (WS). This weight score is calculated from the accumulated size of files that include changes (based on their diff score) or exist exclusively in the reference artifact, divided by the overall accumulated size of files (in the reference artifact).

5.1 Generic System Image (GSI) Builds

In a first step, we aim to measure the state of reproducibility of AOSP. For comparison against our own rebuilds, we rely on the GSI builds as publicly available reference binary artifacts generated by an external source from pure AOSP. These are available through the Android CI dashboard. To observe reproducibility over time, we follow a monthly cadence, taking the first successful build after the first of each month from the "release" branches of GSI (`aosp-<version>-gsi`, where version is `pie` for Android 9, `android10` for Android 10, etc.) for the latest major Android release at that time. Following that scheme, the first build artifact available through the CI dashboard has build ID 5854032 (September 2019) and is the last build (given our monthly cadence) of Android 9 before the release of Android 10 (October 2019). For older builds on the CI dashboard, the firmware images were removed from the available build results. While many builds are marked with the "archived" flag to indicate this removal, the dashboard also shows older successful builds without that flag[1]. This suggests that there might be an issue with the dashboard that causes either improper flagging or accidental removal of older GSI firmware images. We did not receive an official statement from Google on this.

While we prefer to base our comparison on externally built reference images to, hopefully, eliminate potential side-effects of our own build framework, we could overcome the unavailability of older GSI reference builds by comparing build artifacts generated by two instances of our own framework. However, the results seen for Android 9 to 12, seem to show sufficient reproducibility to warrant the comparison of device builds performed in the remaining sections. Hence, we leave analysis of older versions to future work.

Fig. 2 shows the resulting DS and WS over time for each partition image in the GSI build. Note that 0 is (falsely) visualized on the logarithmic DS scale as an extra Y-axis tick slightly below 1 (10^0) to distinguish the value 0 from absent measurements.

The main interesting artifact is `system.img`. Our results show that its contents are, in fact, bit-by-bit identical for most Android 10 builds. However, the ext2 file system image itself is not bit-by-bit identical. Android 11 shifted this issue towards APEX files which contain an ext2 file system in `apex_payload.img`. Again, the contents of the APEX payload is reproducible, but the containing file

[1] Build ID 5771215 (https://ci.android.com/builds/branches/aosp-pie-gsi/grid?head=5771215) is marked as archived, while build ID 5403672 (https://ci.android.com/builds/branches/aosp-pie-gsi/grid?head=5403672) is not, but still lacks the firmware images.

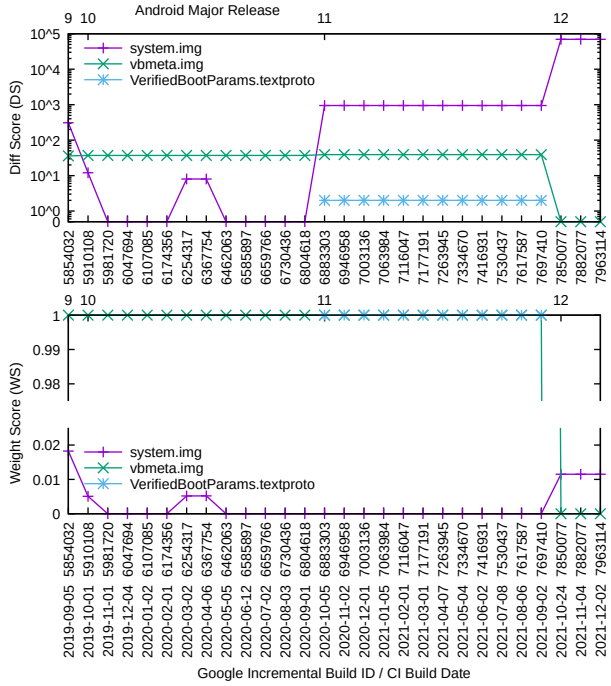

Figure 2: DS and WS metrics over time for GSI builds.

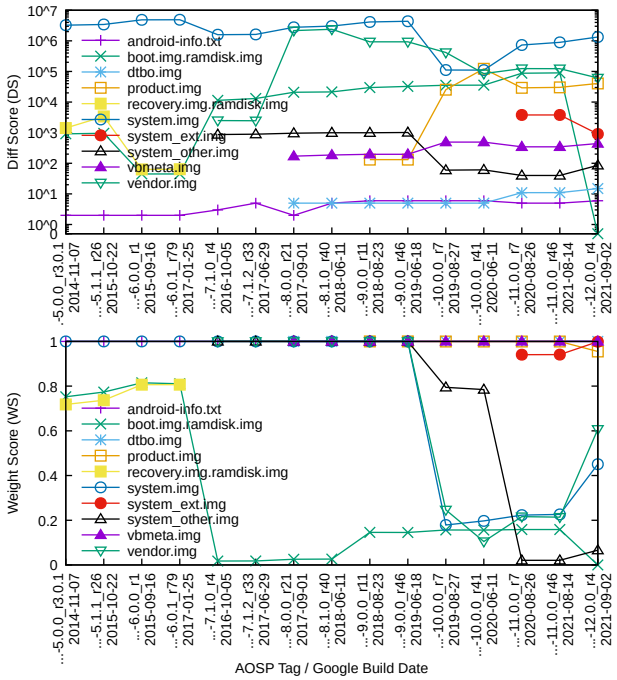

Figure 3: DS and WS metrics over time for device builds.

system image is not. Subsequently, signatures and related metadata of APEX files differ. Android 9, 10 and 12 exhibit differences in native libraries (see section 6.2.1). However, Android 12 resolved the ext2 image issue, making APEX files reproducible, and further improved overall reproducibility by making `vbmeta.img` reproducible and by eliminating `VerifiedBootParams.textproto`.

5.2 Pixel/Nexus Device Builds

Given the result that recent pure AOSP is already reproducible to a significant extent, we analyze if this reproducibility state helps in validating that concrete device firmware images stem from a claimed codebase. Specifically, we use our analysis framework to compare rebuilds of each major release of Android from version 5 to 12. As reference build for our comparison, we use the first and last build of each major release for the Google flagship device (Pixel/Nexus) that has been released alongside that major version (e.g. Pixel 6 Pro for Android 12, see appendix A).

Fig. 3 shows the results of this comparison for each partition image (analogous to the degree of reproducibility of GSI builds). As these official firmware images use slightly different build targets (cf. section 4.1), our measurement results do not reflect reproducibility in a classical sense. Nevertheless, we believe that such a comparison permits a useful assessment of the mapping between supposedly untainted AOSP source code and corresponding firmware images, and also allows to identify issues in the current build and system layout that inhibit validation of that mapping.

As seen in the figure, starting with Android 8, the Treble project (a major overhaul of the Android architecture) introduced additional firmware images for better separation of concerns. This overhaul caused no drastic changes in the measured metrics for most firmware components (particularly not for the `system.img` that is expected to primarily stem from AOSP). Only the `vendor.img`, into which Treble shifted many proprietary components, incurred an (expected) sharp increase of differences.

Android 10 caused another big shakeup of the measured metrics. This marks the start of Project Mainline (cf. section 4.3.2). Clear improvements can be seen mainly for both `system.img` and `vendor.img`, with `system.img` being arguably the most relevant artifact for comparison to AOSP. Unfortunately, Android 12 introduces a regression for `system.img` (analogous to our observation for reproducibility against GSI builds) mainly caused by non-reproducible native shared libraries and ODEX files. Overall, the comparison of factory images against our builds shows reduced differences for the most essential images over time, which we mainly attribute to the positive effects of Project Treble outsourcing expected differences from non-AOSP components to separate partition images.

Another partition image that experienced significant change of differences across observed builds is the ramdisk image. Several additional graphics files were added in Android 7, significantly increasing the overall file size. However, native binaries that previously contained differences became bit-identical in Android 7. Removal of bootloader info message graphics (cf. section 6.3.1) finally made this image fully reproducible in Android 12.

6 QUALITATIVE ANALYSIS

In order to verify the quantitative metrics and the overall output of our analysis framework, and to identify the main causes for differences between official release build artifacts and our (expectedly) sufficiently close rebuilds, we manually inspect the differences identified in the quantitative comparison. We further discuss the implications of these differences and classify them as accountable differences (that may justify exclusion from quantitative

analysis) and unaccountable differences (that prevent perfect reproducibility or comparability in the case of device builds).

All examples illustrated in this section are based on difference reports for the following states of AOSP:

(1) GSI build ID 7963114[2] in branch `aosp-android12-gsi` (built on 2021-12-02) for build type "userdebug",
(2) source repository tag `android-12.0.0_r4`[3] (equivalent to security patch level 2021-10-05) for the device build flow for the Pixel 6 Pro (codename "raven"),
(3) source repository tag `android-11.0.0_r46`[4] (equivalent to security patch level 2021-10-01) for the device build flow for the Pixel 5 (codename "redfin").

Findings from these specific builds were also cross-checked against results obtained from our toolchain for other revisions of the AOSP source code to reduce the possibility of outliers leading to wrong conclusions. However, due to the nature of comparison of closely, but not exactly matching build targets, the difference reports for device build comparison have substantial differences between major Android releases. To determine improvements (or regressions) between major version steps, we also analyze the changes between builds of Android 11 and 12.

6.1 Accountable Differences in Device Builds

Accountable builds may entail artifact differences that can be explained and are tolerable due to specific circumstances. Besides differences that must obviously be excluded from comparison, there also exist accountable differences that are technically fixable, but have strong and valid reasons to be maintained.

6.1.1 Property Files. Several partitions feature one or more property files that record build and configuration properties (e.g. `/system/build.prop` in `system.img` and `/build.prop` in `vendor.img`). Several of these properties differ, accountable to corporate policies or comparison of deviating device build targets, and reflect valuable information for debugging and backtracking, for instance the brand and manufacturer names ("google" vs. "Android", reflected in multiple properties) and the exact build target in case of device builds. However, we see no technical requirements that would warrant these differences. Therefore, despite being accountable, these differences are not excluded from our framework reports. Moreover, we observed several additional properties in these files (classified as unaccountable changes, see section 6.2.4).

6.1.2 APEX/APK File and Package Names. For most APEX files, the name changes between AOSP and the official device factory image (cf. section 4.3.2). This change is accountable, as long as no other changes to APEX content occur.

Similar to this, Google also ships several APK files of AOSP components with their own package name [43]. Besides the different package name, these application components also ship with a different file name (usually, but not exclusively, with "Google" added as prefix or suffix). For instance, `CaptivePortalLoginGoogle.apk` and `GooglePackageInstaller.apk` replace their AOSP variants `CaptivePortalLogin.apk` and `PackageInstaller.apk`.

Overall, we were able to discover 3 such cases in `/system/app`, 5 in `/system/priv-app` and 9 in APEX files. While these changes are, as with APEX files, accountable, we do not exclude file name differences from our framework reports since there is no clear, well-defined and stable naming convention for these changes.

6.1.3 License Attribution. The file `NOTICE.xml.gz` on various partitions lists all installed files on the partition with corresponding license information. Therefore, missing or added files and changes in file names have a direct impact on this file. We consider these differences to be accountable since they are a knock-on effect.

6.2 Unaccountable Differences

Beyond accountable differences, we also discovered issues that were not justifiable with any good reason. This section highlights the most noteworthy differences, where we found clear patterns.

6.2.1 Differences in Native Binaries in GSI Builds. The Android 12 GSI build comparison showed differences in ELF files; specifically in 6 shared libraries (all related to the Bluetooth stack, e.g. `libbluetooth.so`, `audio.hearing_aid.default.so`). All of these differences appear to be alignment issues. Notably, there are no additional/missing entries in the relocation and symbol tables, and only slightly altered locations for the same entries.

6.2.2 Inconsistent Build Type of Vendor Partition. In device builds, the build properties of `vendor.img` indicate build type "userdebug" (in the property `ro.vendor.build.type`) instead of build type "user", which was specified during the invocation of the `lunch` command to initialize the build target. Further inspection revealed that the whole `vendor.img` is not actually built on our build environment. Instead, it is simply copied from the pre-compiled blobs of driver binaries provided by Google for their Pixel and Nexus devices [22]. Besides the two files being identical, this is further confirmed by other metadata in the build properties file that clearly indicate a build at Google infrastructure from around the same time as the full factory image. Notably, this file is different from the `vendor.img` packaged into the corresponding full firmware image. We are not aware of any official source for the pre-built `vendor.img` for build type "user" that matches the official firmware (other than extraction from the pre-built firmware image itself, which is prohibited by Google's terms and conditions [23]). Also, we did not find any mechanism that would initiate a re-packaging (or similar) of the vendor partition during the AOSP build process.

6.2.3 Differences in APEX/APK Content. After normalization of APEX names (cf. section 6.1.2), we observed 3 additional APEX files exclusive to the Google factory image and 1 exclusive to our `android-12.0.0_r4` device build (cf. appendix B). Except one, all remaining comparable APEX files also showed unaccountable content differences, clearly indicating that our build is based on different source code. Most (but not all) of them also clearly indicate this through a different version code. For instance, the repository tag and, consequently, our build artifact contains version code 319999900 for `com.android.conscrypt.apex`, while the factory

[2] https://android.ins.jku.at/soap/7963114_aosp_x86_64-userdebug_Google__7963114_aosp_x86_64-userdebug_docker-Ubuntu18.04/summary.html
[3] https://android.ins.jku.at/soap/android-12.0.0_r4_raven-user_Google__android-12.0.0_r4_aosp_raven-user_docker-Ubuntu18.04/summary.html
[4] https://android.ins.jku.at/soap/android-11.0.0_r46_redfin-user_Google__android-11.0.0_r46_aosp_redfin-user_docker-Ubuntu18.04/summary.html

firmware image contains version 310727000 of the equivalent APEX We could locate this version in a standalone version bump commit (without tag/branch) of the AOSP repository. The version code observed in our builds seems to be used in the development branch to assure that artifacts always have a higher version number than actual releases. While the main design goal of APEX files is seamless upgradeability independent of the overall system partition, we believe that if firmware images are claimed to stem from a specific repository tag, that tag should mark the exact version provided in the published build. Appendix B provides a more detailed analysis.

We observed a similar situation for renamed Google APKs packaged directly into `system.img` (without APEX encapsulation). Aside from their different file name, comparison revealed different version codes, additional manifest entries and resources, and a Google APK targeting multiple CPU architectures (where our version only targets the specific device platform).

6.2.4 Additional Entries in Property Files. The property files in `system.img` of device builds contain a significant number of additional entries in our AOSP builds (61 for tag `android-12.0.0_r4`) that have no matching (or even comparable) entries in the Google factory images. All affected properties are placed in a section of the property files that is prefixed with the comment "ADDITIONAL_BUILD_PROPERTIES".

6.2.5 Differences in Native Binaries. The device builds contain several native binaries that differ from our AOSP builds. This concerns mainly shared libraries (133 files, e.g. `libandroid_runtime.so`) but also executable files (22 files, e.g. `adbd`) in `system.img` and associated APEX files for tag `android-12.0.0_r4`. As opposed to our observation for GSI builds, we found a wide spectrum of differences ranging from changes in debug information to more complex changes (often including additions/removals in the relocation and symbols tables, and even differences in strings embedded into binaries). Given the variety of differences, we assume that at least some of them are a result of building a different source code than what was used for the official build artifacts. While this is obviously the case for Mainline APEX and APK files with version or name mismatches, it is unclear why this also affects other files in `system.img`.

6.3 Evolution over Major Releases

6.3.1 Resolved Image Rendering Issues. Before Android 12, the bootloader utilized info messages encoded in PNG images. Rendered during the build process, these files showed significant bit differences in the Android 11 build while no differences were noticeable in manual visual inspection. Pixel-by-pixel image comparison revealed minimal differences around the borders of font shapes (see Fig. 4), potentially caused by font rendering. Android 12 no longer contains these files and, consequently, eliminated this issue.

6.3.2 System Partition and SoC Vendor Files. SoC vendor related files have existed in a dedicated `vendor.img` since Android 7. However, some SoC related files are tightly coupled with AOSP components [5]. In the past (Android 10 and before), these proprietary files were embedded directly into `system.img`. Starting with Android 11, such components reside in a dedicated `system_ext.img`, no longer cluttering the core `system.img`.

Figure 4: Pixel delta (marked in red) for a bootloader image containing a multilingual info message.

7 CONCLUSIONS

We propose a modular automation framework to analyze current state of reproducibility of AOSP, to measure over-time improvements, and to measure comparability between AOSP and official pre-built firmware images. The latter is particularly important for assessing trustworthiness in situations where users have to rely on pre-built firmware images that they cannot recreate themselves (e.g. due to signature requirements enforced by a bootloader). Our framework builds AOSP from source in its native build system, automatically compares artifacts to reference firmware images, and derives trivial difference metrics that permit comparison.

A weighted change score based on differences in relation to artifact file sizes allows meaningful comparison of relative reproducibility between different revisions of a source code repository, even for complex artifacts as long as the artifacts have comparable structure and contents. This weight score is used to continuously measure changes in reproducibility of AOSP over time.

By comparing against existing, publicly available GSI build artifacts, we found that pure AOSP is already very close to full reproducibility. Although the introduction of additional Mainline modules in Android 11 initially had a negative impact on reproducibility due to non-determinism in file system image generation, this issue has been resolved in Android 12. Unfortunately, Android 12 also introduced new reproducibility issues for native libraries.

Given the good state of reproducibility of AOSP, we evaluated how well Google's Pixel/Nexus factory images can be matched to their claimed codebase for rebuilds of Android 5 to 12. We introduce accountable builds as a form of sufficiently-close rebuilds of firmware images that allow neglection of expected differences (such as different signatures or results of comparison against only a partial codebase) when comparing to production release artifacts.

We found that the separation of concerns introduced by Project Treble helped to continuously get `system.img` closer to AOSP (particularly kicking in with Android 10) while shifting deviations to other partitions. While Project Mainline significantly improves Android security through modularization of upgradable components, it turns out Mainline modules shipped inside firmware images do not match the repository tags identifying those firmware image releases. This significantly inhibits comparability of Mainline modules and not only affects APEX files, but also APK files directly embedded into `system.img`. As with GSI builds, Android 12 introduces a regression with regard to reproducibility of native binaries.

Logical next steps for future work are the continuation of reproducibility analysis for upcoming GSI builds (and potentially also older Android versions), and the extension of our comparison effort to Android device firmware images from other OEMs. Any given OEM factory image could be used as reference for comparison, provided we have a build recipe defining the device-specific build target and the corresponding state of the AOSP source code that the OEM based their customization on.

ACKNOWLEDGMENTS

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by ONCE (FFG grant FO999887054 in the program "IKT der Zukunft") and the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry for Digital and Economic Affairs (BMDW), the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

REFERENCES

[1] Android Open Source Project 2020. *Android Developer Codelab.* Android Open Source Project. https://source.android.com/setup/start

[2] Android Open Source Project 2020. *Dynamic Partitions.* Android Open Source Project. https://source.android.com/devices/tech/ota/dynamic_partitions

[3] Android Open Source Project 2020. *Generic System Images.* Android Open Source Project. https://source.android.com/setup/build/gsi

[4] Android Open Source Project 2020. *Images (in Architecture: Bootloader).* Android Open Source Project. https://source.android.com/devices/bootloader/images

[5] Android Open Source Project 2022. *Android Shared System Image.* Android Open Source Project. https://source.android.com/devices/bootloader/partitions/shared-system-image

[6] Android Open Source Project 2022. *Codenames, Tags, and Build Numbers.* Android Open Source Project. https://source.android.com/setup/start/build-numbers#source-code-tags-and-builds

[7] Aspiration 2015. *Open Technology Fund Community Lab: Reproducible Builds Summit* (Athens, Greece, Dec. 1–3, 2015). Aspiration, San Francisco, CA, USA. https://reproducible-builds.org/files/AspirationOTFCommunityLabReproducibleBuildsSummitReport.pdf

[8] Aspiration 2016. *Reproducible Builds Summit II* (Berlin, Germany, Dec. 13–15, 2016). Aspiration, San Francisco, CA, USA. https://reproducible-builds.org/files/ReproducibleBuildsSummitIIReport.pdf

[9] Alex Birsan. 2021. *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies.* medium.com. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[10] Jérémy Bobbio, Juri Dispan, Paul Gevers, Georg Koppen, Chris Lamb, Holger Levse, and Niko Tyni. 2020. *Reproducible Builds Documentation: Timestamps.* reproducible-builds.org. https://reproducible-builds.org/docs/timestamps/

[11] Jérémy Bobbio, Paul Gevers, Georg Koppen, Chris Lamb, and Peter Wu. 2020. *Reproducible Builds Documentation: Deterministic build systems.* reproducible-builds.org. https://reproducible-builds.org/docs/deterministic-build-systems/

[12] Copperhead 2022. *Building CopperheadOS.* Copperhead. https://copperhead.co/android/docs/building/

[13] Ludovic Courtès and Ricardo Wurmus. 2015. Reproducible and User-Controlled Software Environments in HPC with Guix. In *Euro-Par 2015: Parallel Processing Workshops* (Vienna, Austria) *(LNCS, Vol. 9523).* Springer, Cham, 579–591. https://doi.org/10.1007/978-3-319-27308-2_47

[14] Cybersecurity and Infrastructure Security Agency (CISA). 2020. *Advanced Persistent Threat Compromise of Government Agencies, Critical Infrastructure, and Private Sector Organizations.* Alert AA20-352A. National Cyber Awareness System. https://us-cert.cisa.gov/ncas/alerts/aa20-352a

[15] Stanislav Dashevskyi, Archim D. Brucker, and Fabio Massacci. 2019. A Screening Test for Disclosed Vulnerabilities in FOSS Components. *IEEE Transactions on Software Engineering* 45, 10 (Oct. 2019), 945–966. https://doi.org/10.1109/TSE.2018.2816033

[16] Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. Challenges and Implications of Verifiable Builds for Security-Critical Open-Source Software. In *ACSAC '14: Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, LA, USA). ACM, New York, NY, USA, 16–25. https://doi.org/10.1145/2664243.2664288

[17] Debian Wiki 2020. *ReproducibleBuilds.* Debian Wiki. https://wiki.debian.org/ReproducibleBuilds

[18] devrandom 2022. *Gitian: a secure software distribution method.* devrandom. https://gitian.org/

[19] Robert J. Ellison, John B. Goodenough, Charles B. Weinstock, and Carol Woody. 2010. *Evaluating and Mitigating Software Supply Chain Security Risks.* Technical Note CMU/SEI-2010-TN-016. Carnegie Mellon University, Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/technicalnote/2010_004_001_15176.pdf

[20] Daniel Fullmer. 2022. *robotnix - Build Android (AOSP) using Nix.* GitHub Project. https://github.com/danielfullmer/robotnix

[21] Google 2020. *Android Continuous Integration dashboard.* Google. https://ci.android.com

[22] Google Developers 2021. *Driver Binaries for Nexus and Pixel Devices.* Google Developers. https://developers.google.com/android/drivers

[23] Google Developers 2021. *Factory Images for Nexus and Pixel Devices.* Google Developers. https://developers.google.com/android/images

[24] GrapheneOS 2022. *Build.* GrapheneOS. https://grapheneos.org/build#reproducible-builds

[25] Guardian Project 2022. *Services.* Guardian Project. https://guardianproject.info/services/

[26] hashbang. 2022. *AOSP Build.* GitHub Project. https://github.com/hashbang/aosp-build

[27] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. 2001. *The Vesta Approach to Software Configuration Management.* Research Report SRC-RR-168. Compaq Systems Research Center, Palo Alto, CA, USA. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-168.pdf

[28] Joe Hicks. 2020. *Welcome Android Open Source Project (AOSP) to the Bazel ecosystem.* Google Developers. https://developers.googleblog.com/2020/11/welcome-android-open-source-project.html

[29] Christopher Jämthagen, Patrik Lantz, and Martin Hell. 2016. Exploiting Trust in Deterministic Builds. In *Computer Safety, Reliability, and Security* (35th International Conference, SAFECOMP 2016, Trondheim, Norway) *(LNCS, Vol. 9922).* Springer, Cham, 238–249. https://doi.org/10.1007/978-3-319-45477-1_19

[30] Paul A. Karger and Roger R. Schell. 1974. *Multics Security Evaluation: Vulnerability Analysis.* Technical Report ESD-TR-74-193, Vol. II. Electronics Systems Division (AFSC), L. G. Hanscom AFB, MA, USA. https://csrc.nist.gov/publications/history/karg74.pdf

[31] Chris Lamb, Clemens Lang, and Valerie R. Young. 2019. *Reproducible Builds Documentation: Definition.* reproducible-builds.org. https://reproducible-builds.org/docs/definition/

[32] Elias Levy. 2003. Poisoning the software supply chain. *IEEE Security & Privacy* 1, 3 (June 2003), 70–73. https://doi.org/10.1109/MSECP.2003.1203227

[33] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Transactions on Privacy and Security* 24, 3, Article 19 (April 2021), 35 pages. https://doi.org/10.1145/3448609

[34] Alyssa Miller, Simon Maple, Ron Powell, and Vincent Danen. 2020. *The state of open source security report.* Report. Snyk Ltd. https://info.snyk.io/sooss-report-2020

[35] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-Aware Storage Systems. In *Annual Tech '06: 2006 USENIX Annual Technical Conference.* USENIX Association, Boston, MA, USA, 43–56. https://www.usenix.org/legacy/events/usenix06/tech/muniswamy-reddy.html

[36] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (17th International Conference, DIMVA 2020, Lisbon, Portugal) *(LNCS, Vol. 12223).* Springer, Cham, 23–43. https://doi.org/10.1007/978-3-030-52683-2_2

[37] Open Source Initiative 2018. *History of the OSI.* Open Source Initiative. https://opensource.org/history

[38] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empir Software Eng* 25 (Sept. 2020), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

[39] Red Hat, Inc. 2021. *The state of Enterprise Open Source.* A Red Hat Report. Red Hat, Inc. https://www.redhat.com/en/enterprise-open-source-report/2021

[40] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *ICSE '18: Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden). ACM, New York, NY, USA, 71–81. https://doi.org/10.1145/3180155.3180224

[41] Reproducible Builds project 2022. *diffoscope: in-depth comparison of files, archives, and directories.* Reproducible Builds project. https://diffoscope.org/

[42] Reproducible Builds project 2022. *Reproducible Builds — a set of software development practices that create an independently-verifiable path from source to binary code.* Reproducible Builds project. https://reproducible-builds.org/

[43] Aamir Siddiqui. 2020. *Everything you need to know about Android's Project Mainline.* XDA Developers. https://www.xda-developers.com/android-project-mainline-modules-explanation/

[44] Stacy Simpson. 2008. *Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today.* Rep. SAFECode. https://safecode.org/publication/SAFECode_Dev_Practices1108.pdf

[45] StatCounter Global Stats 2021. *Mobile Operating System Market Share Worldwide – Jan 2021 - Dec 2021.* StatCounter Global Stats. https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201201-202112
[46] Hans-Christoph Steiner, Michael Pöhn, Tobias Groza, Andreas Schildbach, and kitsunyan. 2020. *Reproducible Builds.* F-Droid Ltd. https://www.f-droid.org/en/docs/Reproducible_Builds/
[47] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (1984), 761–763. https://doi.org/10.1145/358198.358210
[48] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19).* USENIX Association, Santa Clara, CA, USA, 1393–1410. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias
[49] David A. Wheeler. 2005. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05).* IEEE, Tucson, AZ, USA, 33–45. https://doi.org/10.1109/CSAC.2005.17

A ANDROID VERSIONS, SELECTED FLAGSHIP DEVICES, AND BUILDS

A.1 Android 5 (Lollipop)

The selected flagship device for Android 5 (Lollipop) is the Nexus 6 (codename "shamu"). The first available firmware build for that device has build ID LRX21O (mapping to tag `android-5.0.0_r3.0.1`). The last firmware build before the next major release has build ID LMY48Y (mapping to tag `android-5.1.1_r26`).

A.2 Android 6 (Marshmallow)

The selected flagship device for Android 6 (Marshmallow) is the Nexus 6 (codename "shamu"). The first available firmware build for that device has build ID MRA58K (mapping to tag `android-6.0.0_r1`). The last firmware build before the next major release has build ID MOB31T (mapping to tag `android-6.0.1_r79`). Actually, the flagship device would be Nexus 6P (codename "angler"). However, the Android 6 driver binaries are not available from Google for that device.

A.3 Android 7 (Nougat)

The selected flagship device for Android 7 (Nougat) is the Pixel XL (codename "marlin"). The first available firmware build for that device has build ID NDE63P (mapping to tag `android-7.1.0_r4`). The last firmware build before the next major release has build ID NZH54D (mapping to tag `android-7.1.2_r33`).

A.4 Android 8 (Oreo)

The selected flagship device for Android 8 (Oreo) is the Pixel 2 XL (codename "taimen"). The first available firmware build for that device has build ID OPD1.170816.010 (mapping to tag `android-8.0.0_r21`). The last firmware build before the next major release has build ID OPM4.171019.021.R1 (mapping to tag `android-8.1.0_r40`).

A.5 Android 9 (Pie)

The selected flagship device for Android 9 (Pie) is the Pixel 3 XL (codename "crosshatch"). The first available firmware build for that device has build ID PD1A.180720.030 (mapping to tag `android-9.0.0_r11`). The last firmware build before the next major release has build ID PQ3A.190801.002 (mapping to tag `android-9.0.0_r46`).

A.6 Android 10

The selected flagship device for Android 10 is the Pixel 4 XL (codename "coral"). The first available firmware build for that device has build ID QD1A.190821.007 (mapping to tag `android-10.0.0_r7`). The last firmware build before the next major release has build ID QQ3A.200805.001 (mapping to tag `android-10.0.0_r41`).

A.7 Android 11

The selected flagship device for Android 11 is the Pixel 5 (codename "redfin"). The first available firmware build for that device has build ID RD1A.200810.0207 (mapping to tag `android-11.0.0_r7`). The last firmware build before the next major release has build ID RQ3A.211001.001 (mapping to tag `android-11.0.0_r46`).

A.8 Android 12

The selected flagship device for Android 12 is the Pixel 6 Pro (codename "raven"). The first available firmware build for that device has build ID SD1A.210817.015.A4 (mapping to tag `android-12.0.0_r4`).

B DETAILED ANALYSIS OF APEX DIFFERENCES

Table 1 shows a detailed comparison of all APEX files that were encountered in the device build for tag `android-12.0.0_r4`. APEX files where file name or version code match between the official build and our own build are marked in the corresponding "Match" columns.

The Google factory image contains 25 APEX files while our build contains only 23. 4 of these APEX files have identical names and 18 could be matched through normalization (according to section 4.3.2). Note that our build of AOSP generates several compressed APEX files (file extension `.capex`) while the factory image contains the uncompressed form. As the Google documentation explains[5], APEX compression minimizes storage impact and requires an additional ZIP file wrapping, but has no influence on the actual functionality.

We found that APEX files with identical name also have the identical version code "1". As discussed in section 6.2.3, most of the APEX files that were subject to name normalization, have their version code set to "xx9999900" in AOSP to assure that development artifacts always have a higher version number than actual releases. The first two digits ("xx") of the version code appear to encode the Android API level ("31" for Android 12). Curiously, `com.android.mediaprovider.apex` is an outlier to this schema, using "3199999***10***" in our build instead. Also, `com.(google.)android.appsearch.apex` has version code "300000000" in both images.

Overall, we cannot infer any correlation between version (mis-) matches and the amount of changes in an APEX file. However, we found that even APEX files with identical file names and identical version code contain differences.

[5] https://source.android.com/devices/tech/ota/apex#compressed-apex

Table 1: APEX file comparison for repository tag `android-12.0.0_r4`.

Google Build		Our Build		Match			
File Name (FN)	Version (V)	File Name (FN)	Version (V)	FN	V	DS	WS
com.google.android.adbd.apex	310727002	com.android.adbd.capex	319999999			100162	0.748
com.android.apex.cts.shim.apex	1	com.android.apex.cts.shim.apex	1	X	X	0	0.000
com.google.android.appsearch.apex	300000000	com.android.appsearch.apex	300000000		X	90	0.010
com.google.android.art.apex	310733000	com.android.art.capex	319999900			23319	0.641
com.google.android.cellbroadcast.apex	310733000	com.android.cellbroadcast.capex	319999900			5	0.995
com.google.android.conscrypt.apex	310727000	com.android.conscrypt.apex	319999900			181	0.525
com.google.android.extservices.apex	310727000	com.android.extservices.capex	319999900			62	0.986
com.android.i18n.apex	1	com.android.i18n.apex	1	X	X	3	0.003
com.google.android.ipsec.apex	310727000	com.android.ipsec.capex	319999900			1	0.000
com.google.android.media.apex	310731000	com.android.media.capex	319999900			150	0.743
com.google.android.media.swcodec.apex	310727000	com.android.media.swcodec.capex	319999900			894	0.994
com.google.android.mediaprovider.apex	310731000	com.android.mediaprovider.capex	319999910			49935	0.967
com.google.android.neuralnetworks.apex	310727000	com.android.neuralnetworks.capex	319999900			30	0.575
com.google.android.os.statsd.apex	310727000	com.android.os.statsd.apex	319999900			137	0.784
com.google.android.permission.apex	310733000	com.android.permission.capex	319999900			241185	0.988
com.google.android.resolv.apex	310733000	com.android.resolv.capex	319999900			82	0.975
com.android.runtime.apex	1	com.android.runtime.apex	1	X	X	141	0.487
com.google.android.scheduling.apex	310727000	com.android.scheduling.apex	319999900			100	0.023
com.google.android.sdkext.apex	310729000	com.android.sdkext.apex	319999900			133	0.918
com.google.android.telephony.apex	1						
com.google.android.tethering.apex	310733000	com.android.tethering.capex	319999900			20	0.442
com.google.android.tzdata3.apex	310727000						
		com.android.tzdata.apex	319999900				
com.android.vndk.current.apex	1	com.android.vndk.current.apex	1	X	X	466	0.078
com.google.android.wifi.apex	310733000	com.android.wifi.capex	319999900			73	0.837
com.google.mainline.primary.libs.apex	310005800						

DS Diff score based on sum of line differences in diffoscope reports. WS Weight score based on relative file size of changed files.

C REPRODUCING THE REPRODUCIBILITY ANALYSIS

We believe that for an analysis of reproducibility, it is also important that the analysis itself is replicable. Therefore, besides our analysis framework, we also provide the necessary scripts to recreate our analysis results in a repository at https://github.com/mobilesec/reproducible-builds-aosp-wisec. These scripts create the analysis reports, the time-series figures, and the APEX comparison table shown in this paper by passing the relevant source code state and reference artifacts to our analysis framework.

Users need to fulfill the following baseline requirements in order to run the entry-point script:

(1) Start with a Linux distribution with Docker support.
(2) Ensure there is at least 750 GB of free storage[6].
(3) Install Docker (see https://docs.docker.com/engine/install/).
(4) The user account used to run our scripts must be able to execute Docker commands without superuser privileges[7].
(5) Install Gnuplot[8], which we use to generate the figures summarizing the quantitative data.
(6) Install Git[9].
(7) Check out the Git repository https://github.com/mobilesec/reproducible-builds-aosp-wisec and switch to that folder.
(8) Optionally, create a working directory and provide the absolute path via the environment variable `RB_AOSP_BASE`. If the variable is unset, our framework defaults to `${HOME}/aosp` (which is created automatically if it does not exist).

Once these manual setup steps are complete, one can simply execute the entry-point script via "`./run-wisec-2022.sh`". This script then performs the following tasks to replicate the jobs that were run on our Jenkins instance:

(1) As a safeguard, all of the above preconditions are verified and further execution is refused if any of them is not met.
(2) Our framework repository is cloned and the version used for this paper is checked out.
(3) The Docker images are built. This is done to customize the image with the current (non-root) user.

[6]We recommend a higher value than Google recommends for building AOSP because we build and analyze 43 versions of AOSP and also need storage for the reference artifacts.
[7]See https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user
[8]See http://www.gnuplot.info/download.html
[9]See https://git-scm.com/download/linux

(4) Then, the build and analysis pipelines (Docker containers) are run, parameterized for each of the 28 GSI builds and 15 device builds analyzed in this paper. Note that this takes a considerable amount of time, even on powerful hardware.
(5) Finally, metrics are extracted to generate the DS/WS columns of the APEX comparison table and the time-series data for the figures. That data is then processed by gnuplot to generate the figures presented in this paper.

The detailed reports created by our framework can be found in `${RB_AOSP_BASE}/diff/`, each subfolder contains the report for one build. The figures visualizing the time-series data reside in `${RB_AOSP_BASE}/figure/`. The LaTeX code for the APEX comparison table resides in `${RB_AOSP_BASE}/apex-table/`.