JOHANNES KEPLER
UNIVERSITY LINZ

Manuel Pöll,
Michael Roland
Institute of
Networks and Security

@ michael.roland@ins.jku.at
https://www.digidow.eu/

July 2021

Analyzing the Reproducibility of System Image Builds from the Android Open Source Project

Technical Report

Christian Doppler Laboratory for
Private Digital Authentication in the Physical World

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World, funded by the Christian Doppler Forschungsgesellschaft, 3 Banken IT GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH.

INSTITUTE
OF NETWORKS
AND SECURITY

DIGiDOW

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Contents

Abstract

This work proposes a modular automation toolchain to analyze the current state and measure over-time improvements of reproducibility of 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. In fact, binary artifacts may require to be designed in a way that makes it impossible to simply detach all sources of non-determinism and all non-reproducible build inputs (such as private signing keys). We introduce "accountable builds" as a form of reproducibility that allows such legitimate deviations from 100 percent bit-by-bit equality. Based on our framework that builds AOSP with its native build system, automatically compares artifacts, and computes difference scores, we perform a detailed analysis of discovered differences, identify typical accountable changes, and analyze current major issues that lead to non-reproducibility. While we find that AOSP currently builds neither fully reproducible nor fully accountable, we derive a trivial weighted change metric to continuously monitor changes in reproducibility over time.

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 [32]). 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 [24] 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 [27]. Open source software is seen as both a potential benefit and also a potential threat to software quality and security [27]: 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. [10, 21, 26]. Recent attacks on software dependencies and the software supply chain in general (e.g. Dependency Confusion [5] and SUNSPOT/SUPERNOVA/SUNBURST [9]) 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. [11, 19, 23, 31]. Already in 1974, Karger and Schell [17] mentioned the threat of code injection by malicious compilers. Practically demonstrating this issue 10 years later, Thompson [34] 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 other third 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 artifacts 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 [36]). While the vast majority of users do not compile their software themselves, they can still profit from independent verification of the correspondence between source code and resulting binary artifacts. Even when this correspondance is verified by only a small number of independent parties, this provides additional trust to the overall user base. 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[1] is probably among the most prominant projects that aim for reproducibility. With the Reproducible Builds project[2], 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 [8]).

In the segment of mobile software and operating systems, there are a few security-focused projects based on AOSP that claim reproducibility (e.g. GrapheneOS[3], CopperheadOS[4], and Guardian Project[5]). Moreover, there is F-Droid as a dedicated app catalogue for Free and Open Source Software (FOSS) targeting the Android platform that supports reproducible builds for Android apps [33]. While AOSP itself does not aim for reproducibility with their current build system[6], there are already a few projects trying to make AOSP reproducible with improved build systems (e.g. AOSP Build[7] and robotnix[8]). 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 correspondance between the AOSP source code and the actual binaries shipped by mobile handset manufacturers and (after-market) firmware providers. Even with Google's announced migration to the Bazel build system [15], 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 to identify the root causes of differences. The Android operating system poses additional challenges to this: The Android ecosystem is heterogenous. There is a huge amount of different build targets resulting into 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 must remain untainted). Finally, firmware im-

[1]https://wiki.debian.org/ReproducibleBuilds
[2]https://reproducible-builds.org/
[3]https://grapheneos.org/build#reproducible-builds
[4]https://copperhead.co/android/docs/building/
[5]https://guardianproject.info/services/
[6]Note that during the course of our work, a new announcement by Google [15] in late 2020 declared a significant change: AOSP will migrate to the Bazel build system, also making "*correct and reproducible (hermetic) AOSP builds*" an explicit goal.
[7]https://github.com/hashbang/aosp-build
[8]https://github.com/danielfullmer/robotnix

ages are signed with keys that must be kept secret. While this itself does not pose a significant issue and is well-studied, firmware images themselves contain signed binaries leading to multiple layers of signatures.

In this paper, we measure the state of reproducibility of AOSP against reference images such as the generic system images and specific device images. We create an automated toolchain for running AOSP builds in their native build system and to automate artifact comparison and analysis of remaining differences. This permits an independent verification of reproducibility and also a quantitative assessment of changes in reproducibility over time. As we found that there are certain complexities in the Android ecosystem that make perfect reproducibility unpractical, 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 non-reproducibility issues of AOSP.

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, at least for open source software and under the assumption that the build toolchain itself is benign, would be to compile all software yourself. 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 [18, 28, 29]. 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 the 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

The troublesome aspect 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 the effects of the environment a build is performed in [7, 25]. However, even in a deterministic build system, non-stable inputs from configuration and environmental differences will cause variations in the outputs. Particularly, uncontrollable parameters in the build environment make determinism difficult.

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 [6]. While often only used for convenience, existing specifications of data formats (and tools based on them) may mandate presence of such metadata though. 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. Even then, 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 nevertheless 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. Such unavoidable differences may come, for instance, from code signing. Obviously, someone trying to reproduce a build must not have access to the official signing keys. 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. 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 [20], 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). In such cases, simply stripping all public keys and certificates from the 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 [30]: 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 keeping the source code (supposedly) 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 functionality of AOSP. 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 and 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 and do not modify the AOSP codebase itself. 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, 25] 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.

For AOSP, there are two actively developed projects (AOSP Build and robotnix) that create new or enhanced build enviroments focused on improving reproducibly building AOSP. Our automation framework differs from their approach in that we explicitly aim to measure reproducibility of the unmodified build system. In fact, we have also put effort in embedding the robotnix build flow into our framework (though we consider that off-scope for this paper).

With their 2020 announcement to migrate to the Bazel build system [15], 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.

Another contribution by the Reproducible Builds project is `diffoscope`[9]. This tool performs recursive, context-aware comparison ("diff-ing") of archive files and generates comparison reports in the form of line-by-line differences and summarized change statistics. We heavily rely on this tool to generate change reports as the basis for our metrics.

With regard to measuring reproducibility, Ren, Jiang, Xuan, and Yang [28] created 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 over time while specifically focusing on the special aspects of the Android ecosystem.

With regard to software supply chain security, there is various ortogonal work. For instance, Torres-Arias, Afzali, Kuppusamy, Curtmola, and Cappos [35] developed a framework to cryptographically guarantee continuous integrity of the whole software supply chain from source code to the artifacts at their end user. Jämthagen, Lantz, and Hell [16] 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.

4. Automating the Analysis

In order to assess the current state of reproducibility when matching the official build instructions as closely as possible, to allow easy reproduction of such an analysis, and to perform a long-term analysis of changes in reproducibility of AOSP over time, we create a system to automate the build and analysis process.

Building AOSP involves several steps, starting with the preparation of a build environment, checkout of the source code, and finally the actual build process [1]. We created a wrapper around the Android build system that automates the entire process of setting up the build environment, fetching and building AOSP, and comparing artifacts with published reference builds. We

[9]https://diffoscope.org/

make the whole environment, which we call "*Simple Opinionated AOSP builds by an external Party*" (*SOAP*), publicly available at https://github.com/mobilesec/reproducible-builds-aosp.

In this section we give a brief introduction and frame the motivation for its design.

4.1 Tooling and Architecture

A majority of the steps of the official AOSP build instructions [1] are shell commands meant for execution in a Bash compatible shell on a Linux host system (specifically Ubuntu LTS). Since our environment should be as close as possible to the official instructions, we opted to use shell scripts for implementing our automation environment. Moreover, additional steps, like download and extraction of driver binaries, can be automated with shell scripting too.

We created an automation environment consisting of several small, composable shell scripts, each responsible for performing a step in the aforementioned process. Each script automates a specific smaller task creating a modularized framework that allows for simple exchange of individual components. A top-level script can be used to execute all steps in proper sequence. This top-level script also allows for continued, automated long-term analysis of reproducibility through, e.g., a Jenkins build server.

After a successful build, our framework performs an automated analysis to compare the build artifacts to official reference builds, and derives quantitative metrics and detailed difference reports. At its core, this process relies on `diffoscope` to perform recursive diff-ing 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. Moreover, 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 improvement of overall reproducibility.

The overall architecture is depicted in Fig. 1. The left side of this figure shows the modules of our framework. The right side shows the corresponding automation steps for setup/preparation, fetching of source code and binary 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

- the generic system images (GSI, cf. [2]) as provided by the Android Continuous Integration (CI) dashboard [12] and
- the official factory images for Nexus and Pixel devices [13] provided by Google.

These sources have been chosen under the following assumptions:

- GSI builds are, by definition [2], "*[...] considered a pure Android implementation with unmodified Android Open Source Project (AOSP) code [...]*" Therefore, they are pure builds of AOSP and best-suited as compare-targets for independent verification of AOSP reproducibility.

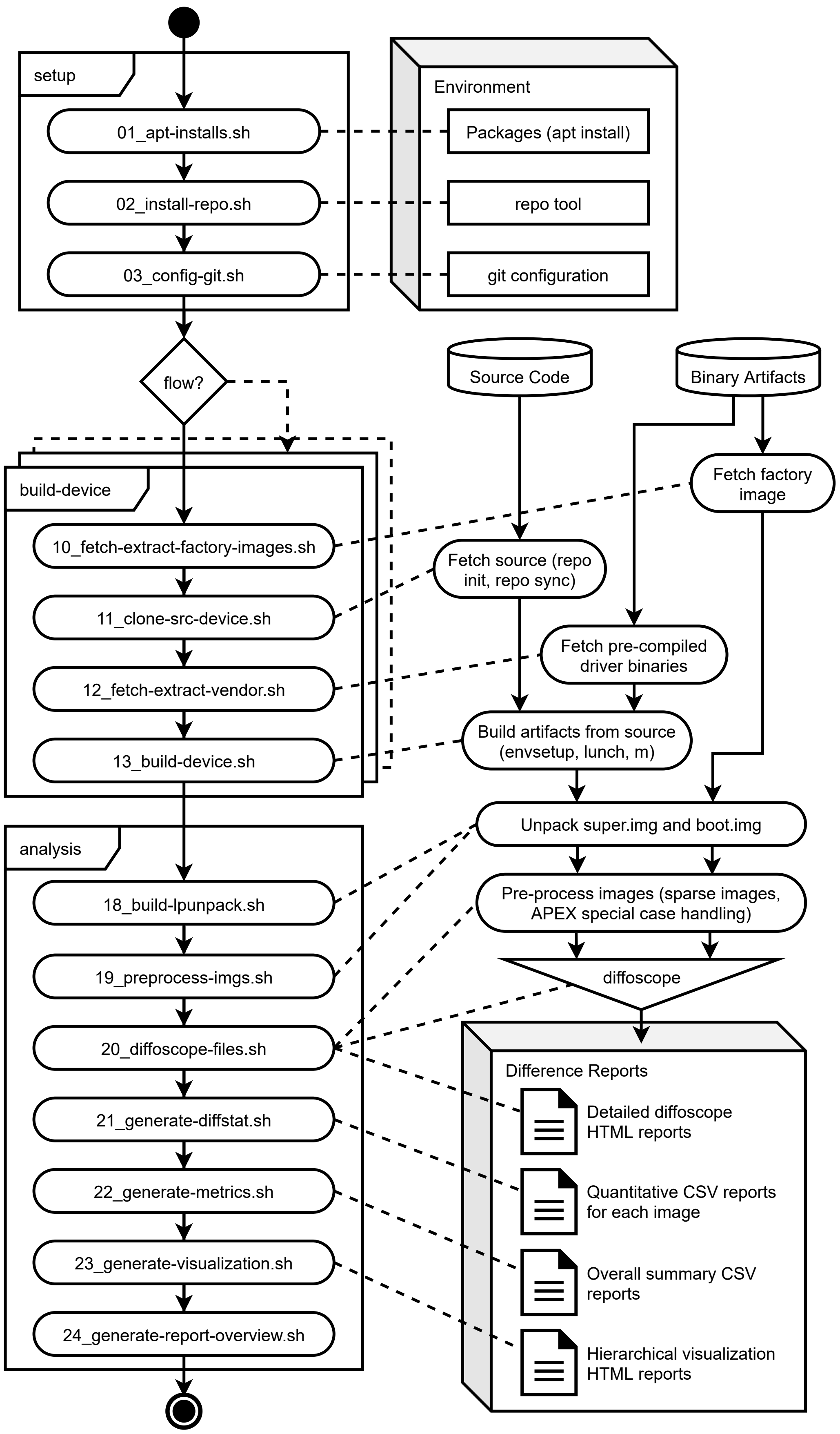
setup
01_apt-installs.sh
02_install-repo.sh
03_config-git.sh
Environment
Packages (apt install)
repo tool
git configuration
flow?
Source Code
Binary Artifacts
build-device
10_fetch-extract-factory-images.sh
11_clone-src-device.sh
12_fetch-extract-vendor.sh
13_build-device.sh
Fetch factory image
Fetch source (repo init, repo sync)
Fetch pre-compiled driver binaries
Build artifacts from source (envsetup, lunch, m)
analysis
18_build-lpunpack.sh
19_preprocess-imgs.sh
20_diffoscope-files.sh
21_generate-diffstat.sh
22_generate-metrics.sh
23_generate-visualization.sh
24_generate-report-overview.sh
Unpack super.img and boot.img
Pre-process images (sparse images, APEX special case handling)
diffoscope
Difference Reports
Detailed diffoscope HTML reports
Quantitative CSV reports for each image
Overall summary CSV reports
Hierarchical visualization HTML reports

Figure 1: Automation framework architecture (left) with corresponding automated processing pipeline (right).

- Pixel (and formerly Nexus) devices are considered Google's flagship devices. They have a timely update schedule and typically cover the latest innovations in AOSP. Moreover, AOSP contains the necessary build configuration (i.e. build targets) for these devices. Google also provides ready-made binary blobs of proprietary, closed-source components that are included into their firmware and not part of AOSP due to various legal reasons. Therefore, we assume that this would be the OEM firmware that is closest to AOSP while still revealing potential reproducibility gaps (in terms of accountable and unaccountable differences) in the transformation phase from AOSP to an actual OEM firmware image.

While out-of-scope for this paper, our modular environment could easily be adapted to analyse other build targets from other sources as long as they rely on the AOSP build system. For other build systems or changes to the current AOSP build system, the additional effort for integrating the necessary steps in the form of new automation modules would be required.

4.2 Deviations from AOSP Build Instructions

Although our goal was to abide by the official build instructions [1], our tests on Ubuntu 18.04 and on 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 itself is written for Python 3, the shebang (`#!/usr/bin/env python`) of `repo` exclusively resolves to Python 2 on Ubuntu. Once started in Python 2, `repo` would then restart itself in the 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.

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. The following are the main issues that needed to be addressed.

4.3.1 Sparse Images

File system images can become quite large, especially due to zero-padding to fulfill alignment requirements 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 [3] used for Google's factory images [13].

As of today, `diffoscope` (version 168) 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 itself is included in AOSP as source code 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 (cf. section 2.2). While the source code of these APEX package files is (or should be) part of AOSP, a side effect of ensuring seemless and conflict-free upgradability is that the package prefix of these APEX files changes from “com.android” (in AOSP) to “com.google.android” (in actual 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 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 solve this issue, 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 [4]. 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

A core concept of the Android platform security model [20] is that not only whole partitions, but each individual application component is digitally signed.

Consequenty, each APK file and each APEX file also has its own signature (cf. section 2.2) that is applied 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.

As a result of this accountable difference, 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 a separate file (`apex_pubkey`) with the signer public key. 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 files) important for reproducibility, we tolerate these changes to propagate into difference reports but eliminate them from quantitative change metrics. The same applies to the comparison of ZIP archive metadata for these files performed by `diffoscope` through the help of `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 both 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 AOSP build process finishes, our analysis framework performs an analysis (comparison and pre-/post-processing) and stores the resulting difference reports as results. 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.
- Measurement of differences is provided as CSV reports summarizing the number of change lines for each artifact ("diff score").
- 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).
- 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" that describes the relative amount of changes with regard to the overall artifact size (see section 6 for details).
- In addition, a summary report of only major differences (for the diff score and the weight score) is derived from the summary CSV report by eliminating unproportionally high (and partially accountable) noise that was identified through a qualitative analysis of individual results (see next sections for the rational behind these simplifications).
- 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. 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 build artifacts and our builds, we manually inspect the results of our comparison. We further discuss implications of these differences and classify them as accountable differences that potentially justify exclusion from quantitative analysis or as unaccountable differences that prevent (full) reproducibility.

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

- Source repository tag `android-11.0.0_r31`[10] (equivalent to security patch level 2021-02-05) for the device build flow for the device codenamed "crosshatch",
- build ID `7101486`[11] (built on 2021-01-26) for the "`eng`" build type for the GSI build flow, and
- build ID `7179179`[12] (built on 2021-03-02) for the "`userdebug`" build type for the GSI build flow.

[10]https://android.ins.jku.at/soap/android-11.0.0_r31_crosshatch-user_Google___android-11.0.0_r31_aosp_crosshatch-user_Ubuntu18.04/summary.html

[11]https://android.ins.jku.at/soap/7101486_aosp_x86_64-eng_Google___7101486_aosp_x86_64-eng_Ubuntu18.04/summary.html

[12]https://android.ins.jku.at/soap/7179179_aosp_x86_64-userdebug_Google___7179179_aosp_x86_64-userdebug_Ubuntu18.04/summary.html

Findings from these specific builds were also cross-checked against other builds from runs of our toolchain against earlier versions of the AOSP source code to reduce the possibility of outliers leading to inaccurate results.

5.1 Accountable Differences

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 reproducibility analysis, there also exist accountable differences where it would technically be possible to fix them, but there exist strong and valid reasons to maintain the differences.

5.1.1 Property Files

Several partitions feature one or more property files that record build and configuration properties (e.g. `/prop.default` in the ramdisk (`initrd.img`), `/system/build.prop` in `system.img`, `/build.prop` and `/odm/etc/build.prop` in `vendor.img`). Several of these properties are accountable to corporate policies 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, which we classified into unaccountable changes (see section 5.2.5).

5.1.2 Project Mainline APK Files

Similar to the treatment of APEX files (cf. section 4.3.2), as part of Project Mainline, Google also ships several APK files of AOSP components with their own package name [30]. Besides the different package name, these application components also ship with a different file name (usually, but not exclusively, with the additional prefix or suffix "Google").

For instance, the files `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 8 in APEX files. While these changes are, as with the 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. However, we observed a trend in alignment of the naming of these files (e.g. `GoogleDocumentsUIPrebuilt` on Android 10 factory images became `DocumentsUIGoogle` in Android 11 where the AOSP version is `DocumentsUI`). This suggests that there is ongoing work towards a unified naming convention.

5.1.3 Image Rendering

The bootloader in the device builds utilizes simple info messages encoded in PNG image files (found under `/res/images` in the ramdisk `initrd.img`).

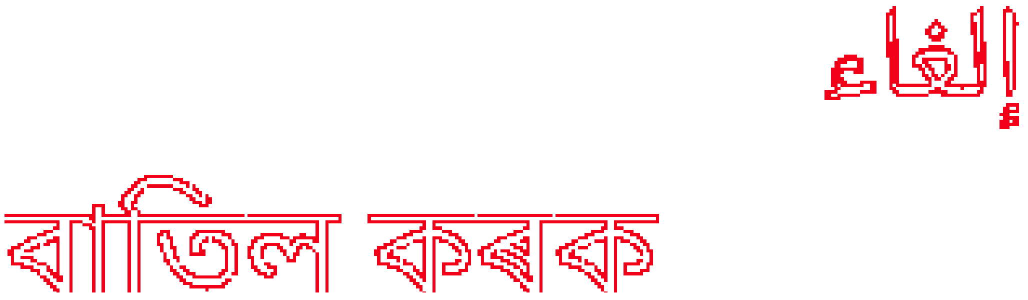

Figure 2: Different pixels (marked in red) for the image `cancel_wipe_data_text.png` containing a bootloader info message in different languages.

These files, rendered during the build process, showed significant differences in terms of differing lines in their hexdump. Visual comparison showed no differences of the actual image. A pixel-by-pixel comparison revealed that there are minimal differences around the borders of font shapes (see Fig. 2). These differences mostly occur on non-roman scripts and seem to stem from font rendering. This clearly makes sense given that rendering of text is even a well-known means for system fingerprinting (cf. [22] for an example of how this may uniquely identify graphics adapters). While such differences are accountable to the different hardware used to build the firmware images, we do not immediately exclude these differences from our framework reports.

The main reasoning behind this decision is that other (more significant) differences could go unobserved by complete exclusion. Nevertheless, we do consider these differences "noise" that we exclude from the summary of major differences.

5.1.4 License Attribution

The file `NOTICE.xml.gz` found on various partitions lists all installed files on the partition with corresponding license information. Therefore, missing or added files and even changes in file names have a direct impact on this file. This is an accountable propagation of other changes that are already reflected in the reports. As a consequence, this difference is included in the reports but excluded from the summary of major differences (analogous to treatment of deviations in image rendering).

5.2 Unaccountable Differences

Beyond differences found to be accountable to organizational or unavoidable technical factors, we also discovered reproducibility breaking issues that were not justifiable with any good reason.

5.2.1 Inconsistent Build Type of Vendor Partition

In device builds, the build properties of the `vendor.img` indicate the "`userdebug`" build type (in the property `ro.vendor.build.type`) instead of the "`user`" build type specified during the invocation of the `lunch` command (which is the command that initializes the build target). Further inspection revealed that the whole `vendor.img` file 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 [14]. Besides the two files being identical, this is further confirmed by other metadata in the build properties file that clearly indicates 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 location for retrieving the pre-built `vendor.img` for the "`user`" build type that matches the official firmware (other than extraction from the pre-built firmware image itself, which is prohibited by Google's terms and conditions [13]). Also, we did not find any mechanism that would initiate a re-packaging (or similar) of the vendor partition during the AOSP build process.

5.2.2 Version Mismatch for APEX Files

Some of the APEX files in Google's firmware images have higher version numbers than their counterpart in our device builds from AOSP. For the analyzed build, this concerns 4 of 19 APEX files (specifically `conscrypt.apex`, `media.apex`, `media.swcodec.apex`, and `resolv.apex`). Besides the mismatch of the version number, file contents also differ.

For instance, our analyzed repository tag `android-11.0.0_r31` contains the version code 300000000 for `com.android.conscrypt.apex` which matches the value in our build artifact. Its counterpart in the official factory firmware image has the version code 300900703, which we could not find in the AOSP repositories at all. The closest neighboring version code that we found was 300900700 in tag `android-mainline-11.0.0_r1`.

While the main design goal of APEX files is seamless upgradeability independent of the overall system partition, this does not justify packaging a different version directly into a firmware image that is claimed to stem from a specific tag in the AOSP source code repository. Otherwise, reproducibility cannot be achieved.

5.2.3 Version Mismatch for Java Libraries

The device builds contain 4 JAR files that exhibit a different Dalvik dexer version number ("2.1.7-r1" in the official factory images vs. "2.1.7-r3" in our builds). As all of these files are embedded inside APEX files, we assume that this has a similar root cause as the version code mismatch in other APEX files.

5.2.4 SoC Vendor Files in System Partition

Despite Google's approach to split vendor files into separate partitions, there are several files that seemingly originate from the SoC vendor (in the analyzed case from Qualcomm) embedded into the `system.img` of the factory image provided by Google. These files are neither part of AOSP nor provided through the pre-compiled blobs of driver binaries [14]. Consequently, they are missing in our device builds of AOSP. These files consist of 7 APK files (e.g. `qcrilmsgtunnel.apk` and `CNEService.apk`, `atfwd.apk`, `uimremoteserver.apk`) and corresponding permission files, a shell script `move_time_data.sh`, and 8 pairs of ODEX/VDEX files (e.g. `QtiTelephonyServicelibrary.odex`) that we could clearly attribute to Qualcomm based on license boilerplates, file names, and other identifiers.

We were unable to identify any good reason why these files are not part of the vendor binary blob or why they are embedded directly into the system image. However, this seems to be a well-known problem in the AOSP user community since there is even a tool (`android-prepare-vendor`[13]) to assemble such files from the original factory images (despite being prohibited by Google's terms and conditions [13]).

5.2.5 Additional Entries in Property Files

The property files in `system.img` and `initrd.img` of device builds contain a significant number of additional entries in our AOSP builds (101 in our evaluation target) that have no matching (or even comparable) entry in the Google factory images. All affected property entries are placed in a secition of the property files that is prefixed with the comment "`ADDITIONAL_BUILD_PROPERTIES`".

5.2.6 Differences in Natives Binaries

The device builds contain several native binaries that differ from our AOSP builds. This concerns mainly shared libraries (e.g. `libcrypto.so`) but also a few executable files (e.g. `adbd`) in APEX files (18 instances). We observed a wide spectrum of differences ranging from only changes in debug information to more complex changes (often including additions to/removals from the relocation and symbols tables). Given the wide 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. This is supported by differences in strings embedded into the binaries. We assume that this may have a similar root cause as the version code mismatches.

The GSI builds also show differences in ELF files compared to our builds. Specifically, we found different shared libraries, optimized ART files (OAT) and optimized Dalvik executable files in the system image (17 instances, e.g. `libbluetooth.so`, `services.odex`) and in the `art.debug` APEX file (20 instances, e.g. `libartd.so`, `boot.oat`) in the GSI builds for the "`eng`" build type.

Contrary to the findings for device builds, differences in the "`eng`" GSI builds appear to be merely alignment issues. Notably, there are no additional/missing entries in the relocation and symbol tables and only slightly altered locations for the same entries.

5.2.7 Missing Camera and Image Processing Libraries

Our build of the `system.img` in the GSI build flow for type "`userdebug`" is missing 19 libraries related to the camera hardware abstraction layer (e.g. `android.frameworks.cameraservice.common@2.0.so`) and image formats/-color encoding (e.g. `libyuv.so`).

5.2.8 Further Differences

Besides the particularly noteworthy differences above, where we found clear patterns, there are also numerous further differences. An in-depth analysis is beyond the scope of this paper.

[13]https://github.com/AOSPAlliance/android-prepare-vendor

6. Quantitative Changes Over Time

An assessment of reproducibility over time requires stable metrics that quantify the state of reproducibility. Existing tools primarily target localizing the source of differences (cf. [28]). For instance, `diffoscope` provides (indirectly through unified diffs) a change score in terms of change lines between two artifacts (in the following called "diff score", short "DS"). In fact, 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" (short "WS"). This weight score is calculated from the accumulated size of files that include changes (based on their diff score after eliminating expected changes as discussed in section 4.4) or exist exclusively in the reference version, divided by the overall accumulated size of files (in the reference source). We further derive a major diff score (MDS) and a major weight score (MWS) based on observations about accountable differences with unproportionally high impact as discussed in section 5.1.

A detailed breakdown of these reproducibility metrics for the three states of AOSP from the qualitative analysis (cf. section 5) can be found in appendix A.

To assess changes in reproducibility of AOSP over time, we perform builds and measurements using our analysis framework on a monthly basis for device builds and bi-monthly for GSI builds. One build per month was chosen based on the monthly interval of AOSP security patch level releases for device builds. This specifically means that device builds are based on AOSP source code repository tags `android-11.0.0rX` (where *X* is the last release number for each security patch level included in the releases published by Google for the Pixel 3 XL ("crosshatch") between September 2020 and March 2021). Resulting difference scores over time for each partition image in these device builds are shown in Fig. 3.

For the GSI builds, the state of the AOSP source code repository mapping to builds available on the 13th and 30th of each month in the same time frame were chosen. As Google changed the GSI build type from "`eng`" to "`userdebug`" by the end of January 2021, we additionally included the last build of the "`eng`" build type (build ID `7101486` on 2021-01-26). The resulting difference scores over time for each partition image in these GSI builds are shown in Fig. 4.

Based on these measurements, we currently cannot infer a clear trend of changes in reproducibility over time. In fact, the weight score of differences for device builds seems to stay rather constant. Interestingly, DS slightly increased for `system.img` within the last few builds (from `r17` to `r23`) causing a significant increase in MDS. The change in system.img is mainly due to additional changes in a single APEX file and, consequently, only has a low impact on WS.

During the same time, the change metric of initrd.img also increased due to additional differences in the SELinux policy. At the same time, the WS for product.img decreased due to the addition of theming files, most of which were found to be reproducible.

For generic builds, we initially saw a slight improvement in reproducibility of the system image with the switch of build types. This is due the absence of alignment differences in native binaries (cf. section 5.2.6) in the "userdebug" build type. Interestingly, build 7142650 for that type shows an increase of WS caused by Dalvik executables containing identical byte code but header changes that suggest differences in intermediate compilation stages between source and binary code. Future builds may reveal if this is a new issue in "userdebug".

Further, when comparing DS, MDS and WS, it can be observed that a significant portion of DS is caused by changes to the file NOTICE.xml.gz, which is filtered from the MDS and does not have any impact on WS due to its relatively low file size. Notably, builds between 2020-12-13 and 2021-01-13 do not contain such differences, indicating that both builds of system.img contained exactly the same list of files. Indeed, before these builds, the differences were caused by SELinux policy files missing from our builds. This has since been fixed. Later builds show other additional files contained in our builds that are missing from the official builds.

Finally, it can be observed that the scores of vbmeta (which is treated as an atomic unit, so WS is either 0 or 1) follow those of vendor. As the purpose this vbmeta image is to exclude system.img from verified boot, vbmeta shows as identical exactly when vendor is identical, which was the case for the last GSI "eng" builds.

Overall, it could be observed that the trivial WS gives stable results that allow comparison of artifact differences across multiple revisions of a source respository. Moreover, artifacts of similar structure and content lead to similarity in results for different types of builds (e.g. system.img across device and GSI builds).

7. Conclusions

We propose a modular automation framework to analyze the current state and to measure over-time improvements of reproducibility of AOSP with respect to official pre-built firmware images. The framework builds AOSP from source with its native build system, automatically compares artifacts to official firmware images, and derives trivial difference metrics that permit comparison of reproducibility state across different revisions of the source repository.

We found that current AOSP builds are neither fully reproducible nor fully accountable. Device builds revealed that Google even uses a codebase for some components that deviates from the officially announced repository tags (particularly for Project Mainline components). Nevertheless, we see that a larger number of individual components of build artifacts is already fully reproducible, including exactly those Project Mainline components.

We derived a trivial weighted change score based on differences in relation to artifact file sizes. Opposed to well-established difference metrics, this weight

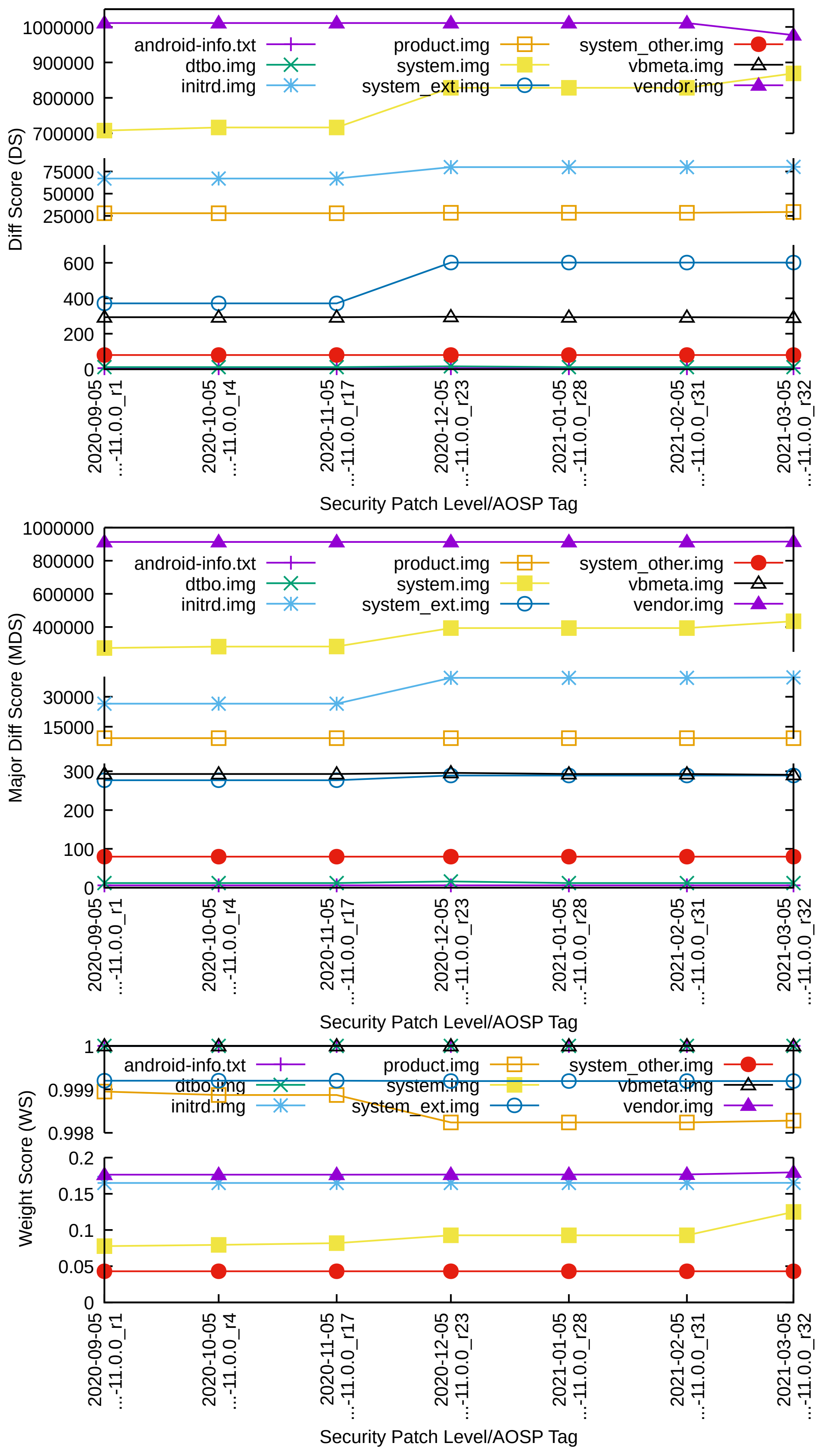

Figure 3: Difference scores over time for device builds.

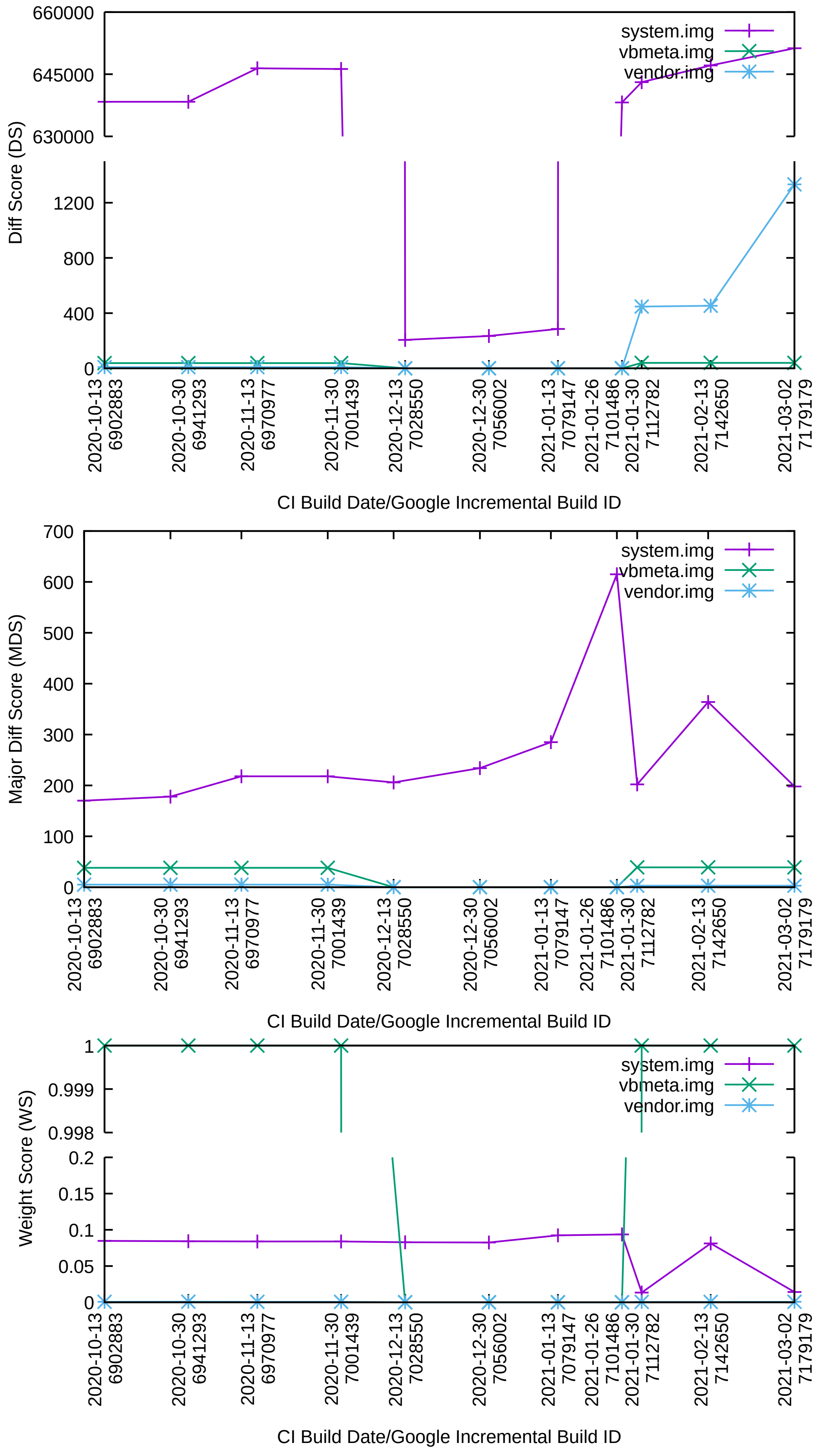

Figure 4: Difference scores over time for GSI builds.

score 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 over time.

While the goal of this work was to create a simple, yet comparable metric, future research will be necessary to further evaluate robustness and reliability of such change metrics through experiments that deliberately break certain aspects of reproducibility. For future work, it may also be interesting to design metrics that are sensitive to, potentially malicious, small code changes.

References

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

[2] 2020. Generic System Images. Android Open Source Project. (November 2020). https://source.android.com/setup/build/gsi.

[3] 2020. Images (in Architecture: Bootloader). Android Open Source Project. (August 2020). https://source.android.com/devices/bootloader/images.

[4] 2020. Dynamic Partitions. Android Open Source Project. (September 2020). https://source.android.com/devices/tech/ota/dynamic_partitions.

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

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

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

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

[9] 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, (December 2020). https://us-cert.cisa.gov/ncas/alerts/aa20-352a.

[10] 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, (October 2019), 945–966. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2816033.

[11] 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, (May 2010). https://resources.sei.cmu.edu/asset_files/technicalnote/2010_004_001_15176.pdf.

[12] 2020. Android Continuous Integration dashboard. Google. https://ci.android.com.

[13] 2021. Factory Images for Nexus and Pixel Devices. Google Developers. (March 2021). https://developers.google.com/android/images.

[14] 2021. Driver Binaries for Nexus and Pixel Devices. Google Developers. (March 2021). https://developers.google.com/android/drivers.

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

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

[17] 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, (June 1974), 156 pages. https://csrc.nist.gov/publications/history/karg74.pdf.

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

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

[20] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2020. The Android Platform Security Model. (December 2020). arXiv: 1904.05572v2 [cs.CR].

[21] Alyssa Miller, Simon Maple, Ron Powell, and Vincent Danen. 2020. The state of open source security report. Report. Snyk Ltd., (June 2020), 58 pages. https://info.snyk.io/sooss-report-2020.

[22] Keaton Mowery and Hovav Shacham. 2012. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of the Workshop on Web 2.0 Security and Privacy*. W2SP 2012, San Francisco, CA, USA, (May 2012), 12 pages. https://hovav.net/ucsd/dist/canvas.pdf.

[23] 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* (*LNCS*, volume 12223). Springer, 17th International Conference, DIMVA 2020, Lisbon, Portugal, pp. 23–43. DOI: 10.1007/978-3-030-52683-2_2.

[24] 2018. History of the OSI. Open Source Initiative. (October 2018). https://opensource.org/history.

[25] Aspiration. *Open Technology Fund Community Lab: Reproducible Builds Summit*. Athens, Greece, (December 2015). Aspiration. https://reproducible-builds.org/files/AspirationOTFCommunityLabReproducibleBuildsSummitReport.pdf.

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

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

[28] 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*. ACM, Gothenburg, Sweden, (May 2018), pp. 71–81. DOI: 10.1145/3180155.3180224.

[29] Aspiration. *Reproducible Builds Summit II*. Berlin, Germany, (December 2016). Aspiration. https://reproducible-builds.org/files/ReproducibleBuildsSummitIIReport.pdf.

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

[31] Stacy Simpson. 2008. Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today. Report. SAFECode, (October 2008). https://safecode.org/publication/SAFECode_Dev_Practices1108.pdf.

[32] 2021. Mobile Operating System Market Share Worldwide – Jan 2021 - Jan 2021. StatCounter Global Stats. https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201201-202101.

[33] Hans-Christoph Steiner, Michael Pöhn, Tobias Groza, Andreas Schildbach, and kitsunyan. 2020. Reproducible Builds. F-Droid Ltd. (December 2020). https://www.f-droid.org/en/docs/Reproducible_Builds/.

[34] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM*, 27, 8, 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210.

[35] 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, (August 2019), pp. 1393–1410. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[36] David A. Wheeler. 2005. Countering trusting trust through diverse double-compiling. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, Tucson, AZ, USA, (December 2005), pp. 33–45. DOI: 10.1109/CSAC.2005.17.

Appendix A. Detailed Quantitative Results

Table 1 provides a breakdown of the diff score and weight score reproducibility metrics for the three states of AOSP from the qualitative analysis (cf. section 5). Omitted cells indicate that an artifact component is not part of that type of build. Most notably, APEX files show as fully reproducible in GSI builds, but still exhibit significant differences in device builds due to different versions included in official releases by Google.

Table 1: Breakdown of reproducibility metrics per image component of the artifact for three states of AOSP.

	Device `android-11.0.0_r31`				Generic `7101486`, `eng`				Generic `7179179`, `userdebug`			
File	DS	MDS	WS	MWS	DS	MDS	WS	MWS	DS	MDS	WS	MWS
`android-info.txt`	6	6	1.000	1.000	0	0	0	0	0	0	0	0
`bootimg.cfg`	0	0	0	0	—	—	—	—	—	—	—	—
`cache.img`	—	—	—	—	0	0	0	0	0	0	0	0
`dtbo.img`	12	12	1.000	1.000	—	—	—	—	—	—	—	—
`initrd.img`	79970	39440	0.165	0.029	—	—	—	—	—	—	—	—
`product.img`	28494	9301	0.998	0.998	—	—	—	—	—	—	—	—
`ramdisk-debug.img`	—	—	—	—	0	0	0	0	0	0	0	0
`ramdisk.img`	—	—	—	—	0	0	0	0	0	0	0	0
`super_empty.img`	0	0	0	0	0	0	0	0	0	0	0	0
`system_ext.img`	601	289	0.999	0.999	—	—	—	—	—	—	—	—
`system.img`	828486	393633	0.093	0.092	638188	615	0.094	0.093	651269	198	0.014	0.014
….`adbd.apex`	415	415	0.488	0.488	0	0	0	0	0	0	0	0
….`apex.cts.shim.apex`	0	0	0	0	0	0	0	0	0	0	0	0
….`art.apex`	—	—	—	—	—	—	—	—	0	0	0	0
….`art.debug.apex`	—	—	—	—	397	397	0.674	0.674	—	—	—	—
….`art.release.apex`	98	98	0.044	0.044	—	—	—	—	—	—	—	—
….`cellbroadcast.apex`	200	200	0.994	0.994	0	0	0	0	0	0	0	0
….`conscrypt.apex`	235	235	0.465	0.465	0	0	0	0	0	0	0	0
….`cronet.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`extservices.apex`	160	160	0.990	0.990	0	0	0	0	0	0	0	0
….`i18n.apex`	1	1	0	0	0	0	0	0	0	0	0	0
….`i18n.apex`	1	1	0	0	0	0	0	0	0	0	0	0
….`ipsec.apex`	109	109	0.911	0.911	0	0	0	0	0	0	0	0
….`media.apex`	439	439	0.621	0.621	0	0	0	0	0	0	0	0

Continutation of Table 1.

File	Device `android-11.0.0_r31`				Generic `7101486`, `eng`				Generic `7179179`, `userdebug`			
	DS	MDS	WS	MWS	DS	MDS	WS	MWS	DS	MDS	WS	MWS
….`mediaprovider.apex`	61308	61308	0.987	0.987	0	0	0	0	0	0	0	0
….`media.swcodec.apex`	338	338	0.258	0.258	0	0	0	0	0	0	0	0
….`neuralnetworks.apex`	221	221	0.992	0.992	0	0	0	0	0	0	0	0
….`os.statsd.apex`	166	166	0.793	0.793	0	0	0	0	0	0	0	0
….`permission.apex`	283089	283089	0.995	0.995	0	0	0	0	0	0	0	0
….`resolv.apex`	217	217	0.374	0.374	0	0	0	0	0	0	0	0
….`runtime.apex`	70	70	0.001	0.001	0	0	0	0	0	0	0	0
….`sdkext.apex`	187	187	0.062	0.062	0	0	0	0	0	0	0	0
….`tethering.apex`	5996	5996	0.953	0.953	0	0	0	0	0	0	0	0
….`tzdata.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`vndk.current.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`vndk.v28.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`vndk.v29.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`vndk.v30.apex`	—	—	—	—	0	0	0	0	0	0	0	0
….`wifi.apex`	40184	40184	0.981	0.981	0	0	0	0	0	0	0	0
`system_other.img`	80	80	0.043	0.043	—	—	—	—	—	—	—	—
`vbmeta.img`	293	293	1.000	1.000	0	0	0	0	39	39	1.000	1.000
`vendor.img`	1010970	913831	0.177	0.176	0	0	0	0	1333	3	0.001	0
`zImage`	0	0	0	0	—	—	—	—	—	—	—	—
Total	2777398	1750518	0.565	0.564	1276376	1230	0.091	0.090	1303910	438	0.014	0.013

DS	Diff score based on sum of line differences in `diffoscope` reports.
MDS	Major diff score based on sum of line differences in `diffoscope` reports excluding unproportional accountable “noise”.
WS	Weight score based on relative file size of changed files.
MWS	Major weight score based on relative file size of changed files excluding unproportional accountable “noise”.