JOHANNES KEPLER
UNIVERSITY LINZ

Author
Manuel Pöll
11707301

Submission
Institute of Networks and Security

Thesis Supervisor
Dr. **Michael Roland**

September 23, 2020

AN INVESTIGATION INTO REPRODUCIBLE BUILDS FOR AOSP

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

Reproducible builds enable the creation of bit identical artifacts by performing a fully deterministic build process. This is especially desireable for any open source project, including *Android Open Source Project (AOSP)*. Initially we cover reproducible builds in general and give an overview of the problem space and typical solutions. Moving forward we present *Simple Opinionated AOSP builds by an external Party (SOAP)*, a simple suite of shell scripts used to perform AOSP builds and compare the resulting artifacts against Google references. This is utulized to create a detailed report of the differences. The qualitative part of this report attempts to find insight into the origin of differences, while the quantitative provides a quick summary.

Zusammenfassung

Reproducible Builds ermöglichen die Erstellung von bit-identischen Artefakten über einen voll-deterministischen Build-Prozess. Im Falle von Open Source Projekten, wie dem *Android Open Source Project (AOSP)*, ist dies besonders erstrebenswert. Zunächst behandeln wir Reproducible Builds im Allgemeinen und betrachten typische Problemstellungen und deren Lösung. Weiters führen wir *Simple Opinionated AOSP builds by an external Party (SOAP)* ein, eine einfache Sammlung von Shell-Skripten welche AOSP Builds durchführt und einen Vergleich gegen Google Referenzen erstellt. Dies nutzen wir um einen ausführlichen Bericht zu erstellen. Der qualitative Aspekt versucht sich an der Ursachenfindung, während der quantitative einen schnellen Überblick verschaft.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

1 Introduction

1.1 Motivation

Open source software is increasingly influential in numerous application domains. Originally a very common form of software development by hobbyists, it has found widespread adoption in companies of all sizes, including large multinational enterprises [27]. Typical usage areas include basic infrastructure (Linux, Docker, etc.), databases (PostgreSQL, MySQL, etc.) and application development across all device platforms.

Android is the single most widely adopted mobile operating system in use today [16, 17] and is also based on open source software, namely the *Android Open Source Project*, short *AOSP*. While many of the prominent Google applications users interact with on a regular basis are not included in the AOSP, it does contain all building components of a fully functioning mobile operating system. This is well illustrated by "custom ROMs" created by various hobbyist[1], including the widely known LineageOS Android distribution[2].

Reproducible builds aim to make the software build process deterministic and thus allow to verify the correspondence between source code and binary artifacts. One key motivation of open source is the establishment of trust. Since anyone may look at the source code, every interested party can inspect code for non-maliciousness on their own[3]. Therefore reproducible builds are a logical next step for open source projects to extend the trust from their source code to the binary artifacts. This trust provides assurance to the user base of an application that binary artifacts were not tampered with[4].

[1] `https://forum.xda-developers.com/`
[2] `https://lineageos.org/`
[3] How widespread such inspections are is another topic entirely and not in scope of this work
[4] It should be noted that large scale software distribution in itself is a complicated topic that involves not only the author of an application, but might also include, but is not limited to: a package maintainer, a software distribution channel, a package manager, the operating system, etc. Actual verifiable trust in installed software generally required trust in all of the aforementioned parties [18, 28]

1.2 Problem Statement

While there are various Android distributions (commonly also called "custom ROMs"), their adoption seems negligible compared to the overall number of Android installations[5]. Additionally only a subset of third party Android distribution users compile their own images, while many opt to download them from Android distribution creators. Thus the following observation holds for the vast majority of Android users: The full trust of the operating system is contingent on the full trust of Google and the device OEM (or even anonymous individuals on platforms like XDA Developers) and there is little to no possibility to verify the correspondence between the binary artifacts running on their device and the source code in AOSP by an independent third party.

Anyone, especially the vast majority of stock Android users, running the operating system via binary artifacts that they did not compile themself can benefit from the increased trust of reproducible builds. Since AOSP currently does not declare reproducibility as an official goal and we are not aware of prior investigations into this subject, we intend to research basic questions on this matter. Based on the above context we formulated the following research questions:

RQ-R Can AOSP/Android be built in a reproducible manner?

- RQ-R-D If not, can all differences be accounted for, i.e. can AOSP/Android be built in an accountable manner?[6]
- RQ-R-CBT What is the number of differences among different build targets (specific devices/ generic system images)?
- RQ-R-COT How does the number of differences change over time?

RQ-BT How can we automate the build process of AOSP and the subsequent analysis for differences in the binary artifacts?

1.3 Project Goals

This bachelor thesis and the related project (*Simple Opinionated AOSP builds by an external Party*, short *SOAP*) aim to build AOSP in a reproducible manner and identify differences to the reference builds provided by Google. We are aware of the following sources of reference builds by Google:

[5] It is difficult to acquire hard numbers on the usage of Android distributions since many advertise strong privacy feature (e.g. no usage statistics). Usage statistics by LineageOS indicate at least 1.7 million users [15] and can be seen a very conservative lower boundary.

[6] See section 2.2 for a definition of accountable builds

- Factory images [12] for phones by Google,
- *Generic system images (GSI)* as provided by the Android CI dashboard [2].

A selection (in regard to version and build target) of specific builds from these sources were made and act as basis of our investigation (RQ-R). Note that some differences are expected and likely unavoidable (c.f. section 2.2 for details, e.g. cryptographic signing keys), all of these instances will be documented. We will attempt to find explanations for all uncovered differences, thus enabling us to declare AOSP as an accountable build (RQ-R-D).

An auxiliary goal of this investigation is the creation of tooling that makes both the build of AOSP and the subsequent analysis process (to determine differences in binary artifacts) simple. This tooling will be released as open source, enabling verification of our process. Furthermore it should allow third parties (even without the technical knowhow) to create their own Android images that match, or come as close as possible to, ours (RQ-BT). Uncovered diffs between official builds and ours should be made accessible via a web interface (helping with RQ-R-D) on `https://android.ins.jku.at/`. Furthermore we intend to aggregate these diffs (number of added/modified/deleted lines). Such a quantitative analysis allows a big picture view of trends on this subject (RQ-R-CBT, RQ-R-COT).

1.4 Outline of the Bachelor Thesis

Initially Chapter 2 covers the general topic of reproducible builds as it is understood in the industry with a focus on the definitions and ground work from the Reproducible Builds initiative [20]. This also delves into some cryptography basics required in later sections. Chapter 3 introduces SOAP, its design choices with underlying rationale, noteworthy aspects, known shortcomings, and gives usage instructions for the two supported build flows. Essentially this section has the same goals as a tool paper. The results of several builds, most notably the uncovered differences against the reference builds, are analyzed and interpreted in Chapter 4. Finally, we conclude this bachelor thesis with Chapter 5 by providing conclusions and an outlook towards future research areas.

2 Reproducible Builds in General

The typical textbook definition of *reproducibility* mandates exact bit equality between two builds of the same *build target* [20]. This is very much a desirable goal since it enables the usage of cryptographic hash functions and thus makes the comparison for reproducibility purposes straight forward and easily automatable. While we could download the full reference build and compare it with ours directly bit-for-bit, the usage of a current (i.e. not broken) cryptographic hash function [14] provides the same functionality while only requiring the download of a tiny fixed size hash value. The check for reproducibility works as follows:

1. Download cryptographic hash over secure channel from a trusted party,
2. Compile your own build for the same build target,
3. Perform cryptographic hashing function on the binary artifacts and compare against known good reference.

Note that the second step involves performing your own build and hence checks for reproducibility. Replacing the second step with the download of a compiled binary, one arrives at the common pattern of integrity verification during transit. This ensures that a binary is not modified between its creation and the receiving end user. While some provide these hashes with the expectation that users compute and compare these manually (e.g. when downloading a Windows executable), this process is done by most software stores or similar, like the Debian package manager [8], automatically. A further extension of this is code signing (c.f. section 2.2.3 for details) which uses a digital certificate to identify the signing party. Code signing is important for its own reasons, but is orthogonal to the reproducibility checking procedure described above and should not be confused with it.

The goal of reproducibility has to be viewed in the context of rising software complexity in general. While academics have long since established the research area of software complexity metrics [29, 31], practitioners keep inventing new, novel programming languages along with their own suites of tooling, notably including build tools[1]. *Meta-build systems*, or more commonly called build script generators[2] add another layer of complexity onto lower level primary build systems. Huge software projects, like Android utilize several programming languages and face the challenge of

[1] E.g. `cargo build` for Rust; `go build` for Go; and even new ones, like Ninja for old, venerable languages like C/C++/Java/...

[2] CMake, GNU autotools, GYP, Meson and similar tools

integrating these. Especially for open source projects it's vital that the build process (most notably build environment prerequisites, build instructions and resource demand) stays within reasonable boundaries to ensure accessibility for current and prospective contributors.

One possible "solution" to the challenge of reproducibility, at least for open source software, is to compile all software for oneself. As previously established (c.f. section 1.1) reproducibility is about bridging the gap between source code and binary artifacts, thus performing this task oneself does create the required trust, but represents more of a workaround than a true solution. Certain projects, like the Gentoo Linux distribution, are focused around this approach[3]. The convenience and predominance of binary distribution for software cannot be overstated and thus mandate a true solution to this problem. Even if a user opts to employ this solution, a malicious build toolchain can still result in compromised binary artifacts.

Additionally it should be noted that reproducibility is also beneficial for closed source software. Various library dependency management systems feature a concept of lock files, e.g. `package-lock.json` for npm and `Cargo.lock` for Cargo, describing the exact versions of libraries (even transitive ones) that the dependency system installed. This information should be committed into a source code repository and will be used for subsequent (re-)installation, thus enabling precisely matching dependencies and should enable reproducible builds. While it is helpful to track the exact versions of dependencies (c.f. libraries/frameworks in section 2.1) it is primarily a responsibility of the build system to perform a reproducible build. One gain that is especially noteworthy in this context is the traceability that is implied with reproducible builds, since this is an important aspect for audits (a regular and often mandatory process for companies; even if their software is closed source from the perspective of the general public).

Note that it does not matter if the application itself and/or one or more of the dependencies is closed source. While the previous examples for dependency systems are prominently known for open source libraries, that is not a requirement for this mechanism. Consider that npm allows the creation of private repositories that keep your own libraries under your full control[4].

2.1 Build Target

The term build target is, at the very least, a tuple of the following orthogonal aspects and produces a set of binary artifacts:

- *Source code*: The entirety of all source files involved in the build, including various configuration files. The state of the aforementioned is usually derived by the checkout of commits from one or more code repositories. For pretty much all non-trivial projects this also encompasses

[3] It should be noted that the primary benefit of Gentoo's approach are optimizations for the specific target hardware (e.g. including modern ISA extensions, like AVX2), but the gained security assurances are a neat benefit on top.

[4] `https://tomspencer.dev/blog/2015/05/20/an-alternative-to-npm-private-modules/`

  - *Libraries/Frameworks*: These are treated as input for the build process and thus any changes in these are likely to result in changes in the binary artifacts.

- *Build Target Architecture*: *Instruction set architectures (ISAs)* as contract between hardware and software vary greatly between platforms and thus require different machine code[5].

  - *Target Operating system*: Binaries that are executed natively (in contrast to interpretation or platform-agnostic intermediate formats) interact with the operating system via *system calls*. These are different for all major OSes and therefore require different machine code.

- *Build Tools*: Tools used for the build process, especially compiler and linker, evolve themselves and thus emit (hopefully better) binaries even if all of the above aspects remain unchanged.

Some projects may extend the above list of aspects with additional considerations. E.g. The Debian project also considers an identical "path to the build directory" as a requirement for a reproducible build [22]. Therefore we opt to keep the term *binary artifact target* purposefully ambiguous, at least in regard to anything beyond the aspects mentioned above.

2.2 Common Problems and their Solutions

However, often, such a bit exact equality is not achievable due to various reasons [9], we will explore some common ones in this section of the bachelor thesis. While some of these are "fixable" by updating the build tooling and/or the application itself, other differences are very much unavoidable (like public keys that are embedded in an artifact).

Even if exact binary equality is not achievable, any difference in a binary artifact that can be accounted for is better than one that cannot be explained. If two builds of the same binary artifact target differ only by fully accountable 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 in the right direction.

2.2.1 Timestamps and Similar Metadata

Many artifacts include timestamps of interest (time of compilation start or finish, birth and/or change of packaged file, etc.) for their core functionality or to augment their usability. The last

[5] While there exists an approach to bundle machine code for multiple platforms, called "fat binaries", these are a niche for specific use cases.

modification timestamp[6] is part of the core ZIP file format specification [30] and allows the preservation of file metadata. Other examples, like the PE file format contain the timestamp[7] solely for debugging purposes [19] and are thus not used in their core use cases.

There are several strategies to deal with timestamps [23], most notably:

- If a timestamp is not essential for the functionality of an artifact, it may simply be stripped in a post process (e.g. `dh_strip_nondeterminism`[8]) or its creation omitted entirely (which may require updating the generating build tool).
- In case an omission is not desirable, one must ensure that a timestamp value is derived in a deterministic fashion, i.e. solely dependent on source code in question. This deterministic timestamp is either written by the builds tool directly (the Reproducible Builds project has established the `SOURCE_DATE_EPOCH` environment variable for this purpose [25]) or patched in a post process step. Different strategies for deriving such a deterministic timestamp exist, e.g.
  - Based on some aspect of the source code. E.g. Debian uses the timestamp of the most recent modification of a packages changelog, thus preserving some of the utility of timestamps while achieving the core goal of determinism.
  - If acceptable for the projects goal, one may use a constant value. E.g. a "0" unix timestamp, indicating January 1st, 1970.

Similar problems exist for various other metadata, these include, but are not limited to:

- Global system properties (hostname, timezone, kernel version, etc.),
- User information (username, locale, etc.),
- Additional metadata on files (owner/group, ACLs, etc.).

2.2.2 Build Path

Some artifacts may contain absolute paths to directories relevant for the build process (location of source code, build output directory, etc.). For example, an ELF binary may contain an optional `RPATH` tag specifying additional lookup directories for the linker [24].

Once again, there are multiple viable solutions:

- If the use case permits it, opt for the specification of a relative path. E.g. Debian recommends this[9] for the previously described `RPATH` tag problem.

[6] Technically these are 2 fields in the ZIP standard, `last mod file time` and `last mod file date` with 2 bytes each.
[7] Specifically the `TimeDateStamp` field in the COFF header embedded in the PE file format.
[8] Part of the `debhelper` package used to bundle software for distribution in Debian.
[9] Specifically of the form `ORIGIN/my/relative/path` to make the path relative to the ELF instead of the current working directory of the invoking process.

- Another approach is ensuring that artifacts contain specific fixed absolute paths while working with (possibly different) paths during the build process. Similar to the timestamp solutions (c.f. section 2.2.1) this may either happen proactively by the build tools (a draft for a `BUILD_PATH_PREFIX_MAP` environment variable aims to solve this [21]; similar to `SOURCE_DATE_EPOCH` for the timestamp issue) or fixed in a post process.

2.2.3 Code Signing

Code signing (i.e. app signing) uses a *public-private key pair* (pk, sk) (for *public key*, *signing key*)[10] to create a signature s of the application to establish authenticity and integrity for binary artifacts. The signature s and pk, which are the two minimal components required to verify a signature, are usually bundled into a digital certificate *cert* in order to integrate with a PKI and provide useful metadata about the key owner (commonly called subject).

Noteworthy examples include Microsofts *Autheticode Code Signing* for Windows [1], used to sign Windows executables. A similar mechanism, called *app signing* is used by Google to sign APKs or app bundles intended to run on Android [26].

Regardless if a software is open source, publicly sharing signing key sk defeats its purpose, thus mandating that each build user has their own (pk, sk) pair. As long as one considers a certificate *cert* as auxiliary metadata, but not part of the binary artifact itself, this is a workable state of conditions for reproducible builds since the different keys/signature are not part of the artifact hash.

[10] Either created by itself or derived from a key chain. This key chain should be rooted in a trusted root authority, managed by one or more certificate authorities (CAs) as part of a public key infrastructure (PKI). Such a PKI vastly strengthens the authenticity claim of the (pk, sk) pair, since a client does not need to trust pk directly, but rather a set of public root keys $\{pk_{root_1}, pk_{root_2}, \ldots, pk_{root_n}\}$ managed by CAs.

3 Implementation of Simple Opinionated AOSP Builds by an External Party (SOAP)

Building AOSP involves several steps, ranging from the preparation of a build environment, check-out of source code and finally the actual build process [3]. A key observation is that the officially documented approach mandates a Linux build system (specifically a Debian based distribution) and all performed steps are commands executed in a bash compatible shell. Additional steps needed for a comparison of binary artifacts, like retrieval of reference images from Google, can easily be done via common utilities (e.g. `wget`) as well. This makes shell scripts the obvious choice for implementing a project to automate the build and analysis process.

3.1 Architecture and Tool Choices

Our project, SOAP consists of several small, composable shell scripts, each responsible for performing a step in the aforementioned process. A master script can be used to execute all steps in proper sequence. Embracing the *Keep is stupid simple (KISS)* philosophy behind Unix compatible shells for the project itself has several advantages:

- Source level modularization makes code more legible by keeping individual files small and focused on specific tasks.
- Splitting of source code facilitates reuse and integration with other tools. Consider the following usages in our project:
  - We use Jenkins to maintain an overview of past and currently running builds. The Jenkinsfile[1] calls the sub-scripts directly, these are grouped in stages (Cloning, Building, Fetch Reference, Analysis) and thus displayed accordingly in the web UI. Allows monitoring of both past and currently active builds, especially their runtime.
  - Sub-scripts are also suitable for execution in a container environment, specifically we validated their successful execution with Docker. In Docker, we first build an image (by running sub-scripts related to the environment setup) and run a container of this image

[1] Configuration file used to specify a pipeline, i.e. a build job description.

(running the checkout/build/analysis related tasks). Again, this simple split is made possible by the division into sub-scripts.

Furthermore the project includes an analysis process that compares our build output to reference builds by Google. This comparison requires the creation of diffs of various file formats, some of which are only containers for other files, e.g. ZIP archives, file system images, APKs. Fortunately there is already a prominent tool for such a (multiple levels deep) recursive and context sensitive diff task, the `diffoscope` tool [10] by the Reproducible Builds initiative. As added benefit, the tool even offers HTML output (via the `--html-dir` flag), helping to fulfill the project goal of providing diffs as a web page (c.f. section 1.3).

3.2 Usage Example

In this section we show an example usage of the SOAP project and guide the user through the required steps. SOAP supports two *build flows*, one tailored for each type of reference build listed in section 1.3. For brevity we only walk the user through the device flow, aiming to reproduce a factory image for a certain Google Pixel device. The following instructions execute the SOAP builds scripts directly, requiring a Debian compatible Linux environment (in contrast to the alternatives via Jenkins or in a Docker container).

1. Ensure you are running a Debian compatible Linux environment. We validated the usage of SOAP with Debian 10 and Ubuntu 18.04.

2. Acquire the SOAP project via Github from `https://github.com/mobilesec/reproducible-builds-aosp`.

3. We need to prepare a build environment by executing all scripts under `scripts/setup`, except `04_config-profile-for-docker.sh`, in proper sequence. The order is indicated by the number in the first two characters of the file name and none of the scripts require parameters.

4. Now we can execute `run-host-device.sh` to perform the device build. Note however, that the following environment variables need to be set[2] before doing so, to properly parameterize your build:

   - `AOSP_REF`: Branch or tag in the AOSP codebase, refer to "Source code tags and builds"[3] for a list. Specifically refer to the "Tag" column, e.g. `android-10.0.0_r40`.

   - `BUILD_ID`: Specific version of AOSP, corresponds to a certain tag. Refer to the "Build" column in the same "Source code tags and builds"[3] table, e.g. `QQ3A.200705.002`.

[2] The preferred way of setting environment variables is via `declare -rx <MY_VAR>="<MY_VAL>"` to make it read-only and export it at the same time, e.g. `declare -rx AOSP_REF="android-10.0.0_r40"`

[3] `https://source.android.com/setup/start/build-numbers#source-code-tags-and-builds`

- `DEVICE_CODENAME`: Internal code name for the device, see the "Fastboot instructions"[4] for a list mapping branding names to the codenames. For example, the "Pixel 3 XL" branding name has the codename `crosshatch`.
- `RB_BUILD_TARGET`: Our build target as chosen in `lunch`[5], consisting of a tuple (`TARGET_PRODUCT` and `TARGET_BUILD_VARIANT`) combined via a dash. The "Choose a target"[6] section provides documentation for these. AOSP offers `TARGET_PRODUCT` values for each device with an "aosp_" prefix. E.g a release build of the previously mentioned "Pixel 3 XL" device would be defined as `aosp_crosshatch-user`.
- `GOOGLE_BUILD_TARGET`: Very similar to `RB_BUILD_TARGET`, except that this represents Google's build target. Note that factory images provided by Google use a non-AOSP `TARGET_PRODUCT` variable. E.g a release build by Google for our running example would be defined as `crosshatch-user`.

Optionally one can set the environment variable `RB_AOSP_BASE` to tell SOAP the base location where all (temporary and non-temporary) files should be placed under. If not specified, it defaults to `${HOME}/aosp`. Refer to "Hardware requirements"[7] for disk space demand.

5. Once the build and subsequent analysis finishes, one can find the uncovered differences in `$RB_AOSP_BASE/diff` in a folder named according to your parameters from the previous step. Details concerning the result format and interpretation can be found in Chapter 4.

Future runs of the SOAP scripts, on the same machine, should start from step 4 since all previous steps are persistent.

3.3 Challenges and their Solutions

During the implementation of the SOAP project we encountered challenges that required unique solutions. While some of these originated from our project goals concerning the analysis of file system images and the general tools we choose to use (c.f section 3.3.5), others were related to the changes that were introduced with Android 10 (c.f. section 3.3.2).

[4] `https://source.android.com/setup/build/running#booting-into-fastboot-mode`
[5] a helper function of AOSP
[6] `https://source.android.com/setup/build/building#choose-a-target`
[7] `https://source.android.com/setup/build/requirements#hardware-requirements`

3.3.1 Sparse Images

Filesystem images can become quite large, especially if certain requirements demand padding with empty space (e.g. due to alignment needs) or contain duplicate blocks of data. To alleviate these problems Android has created their own sparse image format [5] and employs this technique for their factory images [12].

We require raw file system images for our processing and thus need to perform a conversion. Fortunately there is a simple `simg2img` *command line interface (CLI)* utility included in the AOSP project[8] that fits our needs.

3.3.2 ext4 Images with shared_blocks Deduplication

Since Android 10 deduplication has also been integrated into the ext4 file system itself. The `EXT4_FEATURE_RO_COMPAT_SHARED_BLOCKS` feature reduces file size. Noteably the image size savings are also in effect when flashed onto a phones' storage. The previously explained sparse image format also performs deduplication, but is resolved to a regular image during the flash process.

The AOSP repository maintains a "copy" of the `ext4.h` header file (which defines the constants for all ext4 features) used by AOSP to expose symbols to userspace tools. Noteably their version in AOSP[9] does contain the aforementioned `EXT4_FEATURE_RO_COMPAT_SHARED_BLOCKS` feature while the upstream Linux kernel[10] does not define it. This difference stands in contrast to the claim of the header file in AOSP, which reads as follows:

```
***   This header was automatically generated from a Linux kernel header
***   of the same name, to make information necessary for userspace to
***   call into the kernel available to libc.  It contains only constants,
***   structures, and macros generated from the original header, and thus,
***   contains no copyrightable information.
```

The e2fsprogs, the ext2/3/4 file system utilities, do support this feature and refer to it simply as `shared_blocks`[11]. In contrast to the kernel support, these changes to e2fsprogs have been ported upstream[12] and were released with version 1.44.3. Part of this support is the ability to undo the deduplication via `e2fsck -E unshare_blocks <image>` at the cost of additional space. In any sane case

[8] Technicically the source does not include the tool directly, rather it is one of many utilities that is built under `out/host/linux-x86/bin/` during the AOSP build process.

[9] `https://android.googlesource.com/platform/system/extras/+/refs/tags/android-10.0.0_r40/ext4_utils/include/ext4_utils/ext4.h#499`

[10] `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/ext4/ext4.h?h=v5.7.8#n1785`

[11] `https://android.googlesource.com/platform/external/e2fsprogs/+/refs/tags/android-10.0.0_r40/lib/e2p/feature.c#75`

[12] `https://git.kernel.org/pub/scm/fs/ext2/e2fsprogs.git/commit/?id=611d341377607d69b053436fec6de016fe8258fd`

the given image with shared blocks does not contain sufficient space to enable unsharing, after all that is the appeal of this process in the first place, and thus requires increasing the image size via `resize2fs` before unsharing blocks.

Since the `EXT4_FEATURE_RO_COMPAT_SHARED_BLOCKS` feature is not part of the upstream kernel, our Debian based Linux does not understand it. Initially we considered the previously mentioned procedure (file system resizing and unsharing) as a solution to enable working with these images. However, an upstream kernel can still mount the shared images read-only. As noted in the Kernel Wiki[13], the `s_feature_ro_compat` field of the ext4 superblock designates features that permit read-only mounting even if a feature is unrecognized. `EXT4_FEATURE_RO_COMPAT_SHARED_BLOCKS` is such a feature. This is sufficient for our requirements and in fact a benefit, since diffoscope and other tooling can't accidentially perform modifications. We do need to perform the mount operation ourself, instead of delegating it to diffoscope, to ensure it is a read-only mount and thus suceeds.

3.3.3 Part of APEX Files Requires Special Treatment

As part of the effort to make system components easily upgradeable, Google introduced the *Android Pony Express (APEX)* container file format [4] with Android 10. A key part of the APEX file format is the `apex_payload.img`, an ext4 image that also makes use of the `EXT4_FEATURE_RO_COMPAT_SHARED_BLOCKS` feature.

Working with such images requires the consistent usage of a read-only mount and thus we can't delegate this task to the diffoscope tool. The diffoscope tool performs mounts via a C library call (instead of invoking the shell binary) and the author is not aware of an easy way to override the ext4 mount option defaults[14]. To solve this issue we opteded to exclude `apex_payload.img` files during the initial analysis. Subsequently we extract these files in our SOAP scripts from the outer `system.img` and perform the mount operation in these before passing it onto diffoscope for analysis.

3.3.4 Dynamic Partitions

Another recent addition to the file system handling is the introduction of a dynamic `super.img` partition that encapsulates several other partitions, most notably `system.img`, `vendor.img` and `product.img`. As the AOSP documentation explains[15] this only covers partitions that are suitable for read-only mounting and any partition that needs to be accessed by the bootloader is not supported either.

[13] `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#The_Super_Block`
[14] One possible, but somewhat complex, solution would involve monkey patching the C library function via `LD_PRELOAD`.
[15] `https://source.android.com/devices/tech/ota/dynamic_partitions`

The GSI build targets output such a dynamic partition. Therefore the Android CI Dashboard contains only the `super.img` build artifact while ommiting the classical `system.img` and `vendor.img` files. This necessitates the decomposition into the individual partition images, which can be achieved via the `lpunpack` tool from AOSP. While several other utilities (e.g. `simg2img`, see section 3.3.1) are built as part of the standard build targets, this one is not and requires the invocation of its own built target[16].

3.3.5 Image Mounting in Docker Container Requires FUSE

Docker enables the creation of containers, a more lightweight alternative to full fledged virtual machines, that make use of several modern Linux OS level separation mechanisms, like cgroups and namespaces, to isolate one or more programs aimed to provide a specific service. As part of our analysis process of SOAP we need to extract file system images. On the surface this seems like a simple task that should be doable inside a container environment. After all, if a user can read the file system image, like with any other archive format, the extraction is only a matter of properly interpreting the format.

In general, performing a mount operation requires privileged root permissions. Linux also supports a more fine grained permission control via *capabilities* [7], where each capability grants only specific targeted permissions. Docker containers nominally run with a root user, but under the surface only a small subset of capacilities are retained. At first glance it would seem that the `CAP_SYS_ADMIN` capability is sufficient to enable mount operations. While that does work for block devices (as commonly found under `/dev`, e.g. `/dev/sdb`), that trivial solution does not scale to file backed images, which require an allocation via a loopback device. Crucially this does not even work when running the Docker container with `--privileged`[17] [11].

Filesystem in Userspace (FUSE) permits the creation of custom filesystems and also allows mounting in certain limited and well defined circumstances by non-privileged users. Since ext4 is a well established filesystem supported by the main kernel, FUSE implementations don't commonly provide support for it. However, `libguestfs` supports a wide range of file systems by running a QEMU VM ontop of the host kernel via KVM. This works well for our use case and enables mounting of ext4 images even in a container environment.

3.4 Known Deficiencies of Current Approach

Not all of the challenges we encounted could be solved sufficiently. For the sake of completeness we will document the deficiencies here, hoping that future work might provide better solutions

[16] Specifically via `mm -j $(nproc) lpunpack`
[17] Which grants all capabilities and bind mounts the full `/dev` folder from the host.

for these. The one and only noteworthy example of this concerns the interaction between AOSP source code and Docker.

The standard approach for a build process running inside a Docker image is to clone the source code repository as one of the first steps of the container execution. AOSP consists of several git repositories and (as of 2020) consumes roughly 70 GB disk space. Downloading such a huge repository anew for every build in a container environment is a heavy demand and thus we looked for alternative approaches.

One simple approach that comes to mind is embedding the source code into the container image. However, this was quickly discarded since the source code is continually evolving and thus not suitable for this. Furthermore this would expand the image size drastically and the embedding of big files during the image building is generally considered an anti-pattern [6].

We settled on a hybrid approach that employs bind mounts into the container. A considerable advantage is the quick local access to the source code, which we now only need to download as a whole once. Whenever building a newer version of AOSP we still need to fetch code from the remote repository, but the repo synch process only needs to pull small deltas to the current local state. On the other hand this does partially undermine the isolation provided by Docker and crucially prohibits concurrent use of the source code.

4 Interpretation of Uncovered Differences

The analysis of reproducibility and inspection of differences is based on a small subset of builds performed by Google due to resource constraints. After explaining the output format, we first delve into the question if (some) binary artifacts can be built reproducibly (c.f. section 4.2). All artifacts that do not satisfy this condition, are further analysed both qualitatively and quantitively in section 4.3, section 4.4 and section 4.5. The aforementioned analysis is done for both build flows and primarily based of the following versions of the AOSP source code:

- Source tag `android-10.0.0_r35` (security patch level 2020-05-05) for the device build flow and
- Build id `6692013` for the generic build flow.

It should be noted that we did cross check the findings from these specific builds against builds that both preceed and suceed the mentioned versions of the AOSP source code to reduce the possibility of outliers and subsequently wrong analysis. After that we inspect if, and how much the number of differences changes across different version of the AOSP source code (c.f. section 4.6).

4.1 Output Format

An Android installation consists of several partitions that are written to flash storage. Therefore file system images of these partitions, each serving a specific pupose, are the binary artifacts of the AOSP build process. General information about these partitions can be found in the AOSP documentation[1] and a high level understanding is necesarry for the following interpretation.

After the AOSP build proces finishes, SOAP performs an analysis and outputs the result to the `diff` folder. Specifically we create a list of artifacts for both our and Googles' build and only consider files that exist in both for further inspection. For each of these we perform a recursive diff analysis via diffoscope and generate a HTML and CSV report. This is noteworthy because it makes our tooling robust against future partition changes[2] since we have no hard coded assumptions about the partition names and content. As long as the file system image names match, a report will be created. For any file `<artifact>` these reports are:

[1] `https://source.android.com/devices/bootloader/partitions-images` and most of the sibling pages are relevant.

[2] Consider that Android 9 and 10 alone introduced three new partitions, namely `product`, `odm` and `super`.

Table 4.1: Artifacts file lists for source tag `android-10.0.0_r35` and build id `6692013`. Some additional metadata files, notably several `installed-files*.txt/json`, that are themselves file lists for image files, were excluded since they are not part of the actual installation.

(a) Device flow

File Name	Google Build	Our Build
`android-info.txt`	X	X
`boot-debug.img`		X
`boot.img`	X	X
`dtb.img`		X
`dtbo.img`	X	X
`persist.img`		X
`product.img`	X	X
`ramdisk-debug.img`		X
`ramdisk-recovery.img`		X
`ramdisk.img`		X
`super_empty.img`	X	X
`system.img`	X	X
`system_other.img`	X	X
`userdata.img`		X
`vbmeta.img`	X	X
`vendor.img`	X	X

(b) Generic flow

File Name	Google Build	Our Build
`android-info.txt`	X	X
`cache.img`		X
`dtb.img`		X
`encryptionkey.img`		X
`ramdisk-debug.img`	X	X
`ramdisk.img`	X	X
`super.img`	X	
`super_empty.img`	X	X
`system.img`	X	X
`userdata.img`		X
`vbmeta.img`		X
`vbmeta_system.img`		X
`vendor.img`		X

- A detailed HTML report in the folder `<artifact>.diff.html-dir` showing diffs for all artifacts that exhibit variations. View by opening the `index.html` file within.
- Quantitative analysis of the results is provided by a CSV report, located in `<artifact>.diff.json.csv`, summing up the number of added, removed and modified lines for each artifact, essentialy a histogram in text form[3]. This CSV file is visualized in a hierarchical treemap in `<artifact>.change-vis.html`.

Note that APEX files within images are treated specially. Their `apex_payload.img` image files are excluded from the top level image analysis report for technical reasons. Instead they each receive their own reports, specificially for a given `<apex-file>`, these can be found under `<artifact>.apexes/<apex-file>-apex_payload.img.diff` with the respective suffixes.

Finally we aggregate the individual CSV reports (which exist for each artifact) into a single `summary.csv` file. A variation of this, named `summary-major.csv`, exclude certain differences and gives a more concise picture. These big picture reports for our two running samples are reproduced in section 4.5, together with a rational for the "major" variation.

[3]The CSV is the result of a decently complex post process and derived from a JSON output of diffoscope, hence the chained file suffix.

For reference we provide a list of files for the specific builds mentioned at the start of the chapter in table 4.1. Before inspecting any of the image files in detail, we can already disregard several of them (for some build types) due to their nature. These are the following:

- `boot-debug.img`, `ramdisk-debug.img` are needed to run certain compliance test suites since Android 10[4] and, with a minor exception, these are not distributed by Google.
- GSIs rely on the pre-existing boot loader and thus do not supply their own `boot.img`
- Before the introduction of seamless updates, i.e. A/B partitions, the `cache.img` served as temporary download space for system updates, but is no longer part of modern installations.
- Device tree related partitions (both `dtb.img` and `dtbo.img`) are not applicable to GSIs since they are device specific. In case of Pixel phones one can disregard the `dtb.img` (exclusive in our build) since it is only an indermediate output, and will be apended to the kernel image within `boot.img`. However, `dtbo.img`, the Device Tree Blob for Overlay, is relevant for Pixel devices.
- `encryptionkey.img` only exists in our GSI build and thus can't be compared against anything.
- Sensor calibration data stored in `persist.img` is unique to each device and thus neither part of Pixel factory images nor GSIs.
- `product.img` contains customization and apps from the OEM. This is out of scope of the GSIs. Pixel phones ship with a suite of Google proprietary apps, while AOSP contains basic open source variants. Differences are expected and normal.
- `ramdisk-recovery.img` is a specific ramdisk tailored for the recovery mode. It is not part of Pixel factory images/GSIs and thus we have no reference to compare against.
- The regular ramdisk image (`ramdisk.img`) is only an intermediate and packaged into `boot.img`, at least for the regular device builds.
- Section 3.3.4 covers `super.img` in detail, it is only a container for other partitions, namely `system.img`, `vendor.img` and `product.img`. We analyse the relevant embedded ones directly.
- As the name indicates, `userdata.img` contains files and apps from the user and thus is not part of Pixel factory images/GSIs.
- The top level `vbmeta.img` signs, among others, the `boot.img`. With `boot.img` being out of GSI scope, this image can be savely disregarded. On the surface it would appear that `vbmeta_system.img` can be included in a GSI comparision. In practice this is prohibited by the absence of such a partition in Google provided builds.
- Due to their generic nature, GSIs do not have their own `vendor.img`, but rather rely on the pre-existing image.

[4] `https://source.android.com/compatibility/vts/vts-on-gsi`

4.2 Reproducible Artifacts

To determine full reproducibility and thus answering RQ-R we simply performed a cryptographic hash function on the artifacts, as explained in chapter 2. The following artifacts were reproducible:

- For the device flow that was only true for `super_empty.img`.
- For the generic flow these were: `android-info.txt`, `ramdisk-debug.img`, `ramdisk.img`, `super_empty.img`.

After excluding all files that inherently cannot be reproducible, as well as those that already achieve this goal, we are left with the following artifacts that require inspection:

- For the device flow these are: `android-info.txt`, `boot.img`, `dtbo.img`, `system.img`, `system_other.img`, `vbmeta.img`, `vendor.img`.
- For the generic flow these are: `system.img`.

4.3 Accountable Differences

4.3.1 Filesystem Timestamp Metadata in Several Partitions

Google device build targets normalize the filesystem timestamps (ctime/mtime/atime) to a specific date, namely January 1, 2009. This is laudable, since it it serves the goal of reproducibility and is in fact the second solution explained in section 2.2.1. Unfortunately this normalization process is not done for any of the `aosp_*` build targets[5]. Subsequently we encountered filesystem timestamp differences in our original analysis and added the `--exclude-directory-metadata=recursive` flag to diffoscope to suppress them.

4.3.2 App Signing of Embedded APK/APEX Files, OTA Certificates and SELinux

When building AOSP, each APK included and bundled into the images, has its own digital certificate. As established in section 2.2.3 these certificates need to be different for each build user to maintain the security aspects of Android app signing. Therefore we exclude `original/META-INF/CERT.RSA` from our analysis.

Similarily we face this problem for APEX files, thus excluding `META-INF/CERT.RSA` and `apex_pubkey`. Finally there are two additional certificates (`update-payload-key.pub.pem` and `releasekey.x509.pem`; or `testkey.x509.pem` as it is named in our builds) that appear to be related to the *over the air (OTA)* update functionality and were exempt from the report results as well.

[5]This concerns all partitions, except `vendor.img`. That one is normalized even in our build.

Unfotunately the above exclusions are not sufficient to fully address this problem. APEX files contain `META-INF/CERT.SF` and `META-INF/MANIFEST.MF`, both of which contain file listings with checksums. Thus the previously named certificate files show up as changed here. Due to the possibility of other, non-accountable, changes showing up here as well, we did not add these to the exclusion list. A similar argument applies to the `zipinfo` tool invocation for APK files. Ultimately these false positives are a compromise in our report that are mainly rooted in the inability to customize diffoscope for such fine grained exclusions.

Another related subject are the certificates trusted by SELinux. These are 3 ASN.1 certificates, encoded as DER hexdump [13] inside `system/etc/selinux/plat_mac_permissions.xml` in `system.img`. Just like with all previous cases of cryptographic signing, this difference is accountable. However, due to the sensitivity of this file, we opted to tolerate this false postive in our report and did not put it on the exclusion list.

4.3.3 Various Metadata in Property Files

Several partitions feature one or more property files[6] that record build and configuations properties. Several of these properties, e.g. exact timestamp of build, signing keys, build target (in case of device builds), brand name (`google` vs. `Android`), etc., are accountable and self explanatory.

However, in both the device and GSI builds we observed several additional properties, c.f. section 4.4.2 for details.

4.3.4 Google-AOSP Variations in `system.img` (Device Build Only)

Similar to how `product.img` contains apps from the OEM, Google specific applications (as replacement for AOSP parts) can be found in `system.img` as well. We were able to infer the following variations:

- The following apps were found under `/system/app`:
  - `CaptivePortalLogin.apk` - `GoogleCaptivePortalLogin.apk`,
  - `ExtShared.apk` - `GoogleExtShared.apk`,
  - `PrintRecommendationService.apk` - `GooglePrintRecommendationService.apk`.
- Furthermore this applies to the following priviledged apps, located under `/system/priv-app`:
  - `DocumentsUI.apk` - `GoogleDocumentsUIPrebuilt.apk`,

[6] We identified `/prop.default` in the ramdisk within `boot.img`, `/system/build.prop` in `system.img`, `/build.prop` and `/odm/etc/build.prop` in `vendor.img`

- `ExtServices.apk` - `GoogleExtServicesPrebuilt.apk`,
- `NetworkPermissionConfig.apk` - `GoogleNetworkPermissionConfig.apk`,
- `NetworkStack.apk` - `GoogleNetworkStack.apk`,
- `PackageInstaller.apk` - `GooglePackageInstaller.apk`,
- `PermissionController.apk` - `GooglePermissionControllerPrebuilt.apk`,
- `Tag.apk` - `TagGoogle.apk`.

4.3.5 Info Messages utilized by Bootloader (Device Build Only)

The bootloader utilizes simple info messages encoded in images files, found under `/res/images` in the initial ramdisk within `boot.img`. These do not match exactly between our builds, but the purely visual differences are so minor that they are not noticable. Origin of this are likely floating point rounding innacuracies during image generation.

4.3.6 License Attribution for Files

The `NOTICE.xml.gz` file, found under `etc` in various partitions lists installed files on the partition with their associated license. Due to all the other differences covered in section 4.3 and section 4.4 there are many files that exist exclusively in one build or the other. Thus it is logical that these differences show up here as well.

4.4 Unaccountable Differences

4.4.1 Google APEX Files have Different Versions than AOSP Counterparts (Device Builds Only)

The device flow features a subset of APEX files that have Google specific variations from the AOSP base. These start with a `com.google.android` prefix (in contrast to simply `com.android`) and feature different version numbers in `apex_manifest.json`. Beyond that, they appear extremely similar, each having a direct AOSP counterpart and serving the same core functionality. In light of the different version number it makes little sense to look into further differences, which do exist, in these APEX files.

On the one hand, a core design goal of APEX files is that they can be independently upgraded from the system partitions. This gives rise to the argument to classify this difference as accountable. On the other hand, since newer version can be installed anyhow, one would expect that the same version of the AOSP source code would produce APEX files with the same version number. We verified that the APEX version number in our build is sound, i.e that `290000000` found at the `android-10.0.0_r35` tag[7] matches the number in our APEX build artifact. We were not able to find a documentation mapping AOSP git tags to APEX version numbers explicitly, but were able to identify the untagged commit[8] for the `com.google.android.conscrypt.apex` APEX file which bumped the version number to `291601500`, exactly as found in the Google factory image.

4.4.2 Additional Entries in Property Files (Device Builds Only)

All property files in the device builds (except `/odm/etc/build.prop`) exhibit additional entries in our build that have no comparable key in the builds provided by Google.

4.4.3 VIXL Library in Runtime APEX (Device Builds Only)

The VIXL library, located at `/lib/libvixl.so` and `/lib64/libvixl.so` respectively, in the Runtime APEX (named `com.android.runtime.release.apex`) features differences. A notable one is an additional entry to the symbol table, namely for `exp2` from the standard C library. Note that in contrast to section 4.4.1 these both have a `com.android` prefix and their version matches.

4.4.4 SoC vendor (Qualomm) related files in `system.img` (Device Builds Only)

There are several files that originate from Qualcomm and exist exclusively in the `system.img` provided by Google. These are:

- `QAS_DVC_MSP/QAS_DVC_MSP.apk` and `QAS_DVC_MSP_VZW/QAS_DVC_MSP_VZW.apk` in `/system/app`. Affiliation was detemined by the `NOTICE` file found in these APKs.
- `move_time_data.sh` in `/system/bin`, as indicated by the license header in the shell script.
- 3 permission related files found under `/system/etc/permissions`, infered affiliation by filename.
- 17 pairs of `.odex` and `.vdex` files. All their names clearly point to Qualcomm, e.g. `Qti`, `com.qualcomm`, `vendor.qti` filename prefixes.

[7] E.g.for `com.android.conscrypt.apex` that can be seen under `https://android.googlesource.com/platform/external/conscrypt/+/refs/tags/android-10.0.0_r35/apex/apex_manifest.json`

[8] `https://android.googlesource.com/platform/external/conscrypt/+/e33f52804036de692bfe4a4196a86fcea8a25269`

- 40 libraries located at `/system/lib64` (as well as their equally numbered 32-bit counterparts under `/sytem/lib`). All start with a `com.qualcomm.qti.` or `vendor.` prefix.

4.4.5 `vendor.img` uses Inconsistent Build Target Variant (Device Build only)

The `vendor` partition is built with a `userdebug` target variant[9], instead of the `user` variant specified during `lunch` invocation. This might be the origin of the differences enumerated in section 4.4.4 and for this reason we don't expand on the numerous differences found in `vendor.img` in general.

4.4.6 ELF Debug Info Differences (GSI Builds Only)

The majority of GSI differences are ELF binaries (executables and shared libraries; some with `.oat` and `.odex` file endings) located under `/system/framework/oat`, `/system/framework/x86` and `system/framework/x86_64` in `system.img`, as well as `/bin`, `/javalib`, `/lib` and `/lib64` in the `com.android.art.debug` APEX. These files only differ in terms of debug data, that includes the `NT_GNU_BUILD_ID` in all cases. Additionaly `/lib64/libart.so` in the `com.android.art.debug` APEX has more expansive differences. Notably, `com.android.art.debug` is the only APEX (of the 18 in the GSI build) exhibiting differences.

All `.oat` files, as well as the `/system/framework/oat/x86_64/services.odex`, feature minor differences in their accompanying `.art` files at their beginning.

4.4.7 Miscellaneous Differences (Device Builds Only)

Even after considering all of the previously explained accountable and unaccountable differences, we are still left with more. These are unfortunately quite numerous and varied in nature. Instances that don't warrant their own subsection, but are still noteworthy, include:

- An extra ELF binary (`smcinvoked`) that only exists in the Google build, located under `/system/bin` in `system.img`
- `/system/lib64` (and `/system/lib` respectively) in `system.img` feature 6 libraries that have no counterpart. Googles build ships with `libqcbor_system.so`, `libseccam.so` and `libsns_fastRPC_util.so`, while our build contains `libDiagService.so`, `libframesequence.so` and `libgiftranscode.so`

[9] As seen in the `ro.vendor.build.type` property in the `build.prop` property file found in `vendor.img`

Table 4.2: Lines of difference for `android-10.0.0_r35` and build id `6692013`. Note that the APEX entries in this table only cover their `apex_payload.img` content and we ommited the `com.android` prefix for brevity.

(a) Device flow

File Name	ADD	DEL
`android-info.txt`	0	6
`boot.img`	35529	35463
`dtbo.img`	5	5
`system_other.img`	111	218
`system.img`	86854	110073
`vbmeta.img`	206	206
`vendor.img`	937116	756949
`tzdata.apex`	28714	28681
`media.swcodec.apex`	1289	1327
`conscrypt.apex`	147	147
`resolv.apex`	62	64
`apex.cts.shim.apex`	0	0
`runtime.release.apex`	77	77
`media.apex`	481	498
All	1090591	933714

(b) Generic flow

File Name	ADD	DEL
`android-info.txt`	0	0
`system.img`	640943	641031
`ipsec.apex`	0	0
`vndk.v28.apex`	0	0
`art.debug.apex`	74	74
`runtime.apex`	0	0
`cronet.apex`	0	0
`tzdata.apex`	0	0
`media.swcodec.apex`	0	0
`i18n.apex`	0	0
`conscrypt.apex`	0	0
`neuralnetworks.apex`	0	0
`vndk.current.apex`	0	0
`vndk.v29.apex`	0	0
`adbd.apex`	0	0
`resolv.apex`	0	0
`sdkext.apex`	0	0
`apex.cts.shim.apex`	0	0
`tethering.apex`	0	0
`media.apex`	0	0
All	641017	641105

4.5 Quantitative Analysis

A quantitative analysis is done by summing up the number of line differences for each artifact, as well an overall sum. This can be seen for our running examples in table 4.2. Note that our `diffstat` invocation reports a third data point, representing "modified" lines for each change. The diffoscope tool generates *unified diffs*, a format that has no inherent notion of changed lines, only addition and deletion. Since `diffstat` makes no attempt at determinig which pairs of deletion/addition represent a modification[10] and thus reports 0 "modified" lines for all unified diff blocks, we omit this value.

Initially these numbers might seem alarmingly high, but after some review we determined that the vast majority are due to *minor differences*, which are either "effectless" or chain reactions from something trivial. As such we classify:

- `vendor.img` and APEX files from the device flow with a version mismatch are left out entirely, see section 4.4.1 and section 4.4.5 why this is sensible.

[10] As far as the author is aware, such a process is not trivial and inherently heuristic based.

Table 4.3: Lines of major difference for `android-10.0.0_r35` and build id `6692013`. Variation of table 4.2

(a) Device flow

File Name	ADD	DEL
`android-info.txt`	0	6
`boot.img`	153	87
`dtbo.img`	5	5
`system_other.img`	111	218
`system.img`	883	1213
`vbmeta.img`	206	206
`runtime.release.apex`	77	77
All	1435	1812

(b) Generic flow

File Name	ADD	DEL
`android-info.txt`	0	0
`system.img`	190	231
`art.debug.apex`	74	74
Other 17 APEX files	0	0
All	264	305

- Visual differences in the bootloader info messages (section 4.3.5) have no effect on functionality at all.
- `NOTICE.xml.gz` is only a chain reaction from the file list differences[11].

While these differences are valuable in the HTML and individual CSV reports, they obscure the big picture view of this summary. As a consequence we created a "major" variation of this summary, which excludes the aforementioned minor differences, as seen in table 4.3.

4.6 Diff Changes over Time

The quantitative analysis of five different builds acts as basis for answering RQ-R-COT, these are:

- AOSP tags `android-10.0.0_r30` through `android-10.0.0_r40` for device builds. These cover the security patch levels from March through July 2020, one for each month.
- The oldest inspected generic build, `6692013`, was performed by Google on July 19th, 2020. Whereas the newest build `6729552` was done on August 2nd, 2020.

Over the five month span that these tags represent we can hardly see any change (fig. 4.1), save for a minor reduction for the two latest builds. Figure 4.2 shows some variation for the GSI builds. Considering the short timeframe of the examined builds, inference of an overall trend does not seem wise and is likely the reason for the non-monotonic curves.

[11] For `system.img` this is especially drastic, accounting for 98.9% of line differences in the artifact for the device build.

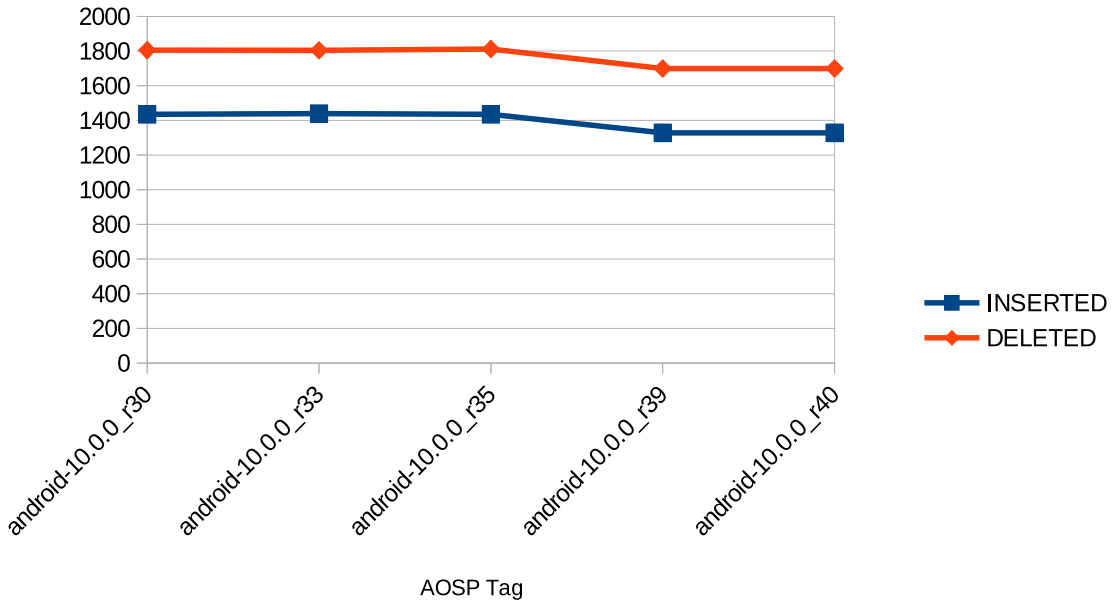

Figure 4.1: Lines of major difference for a range of device builds

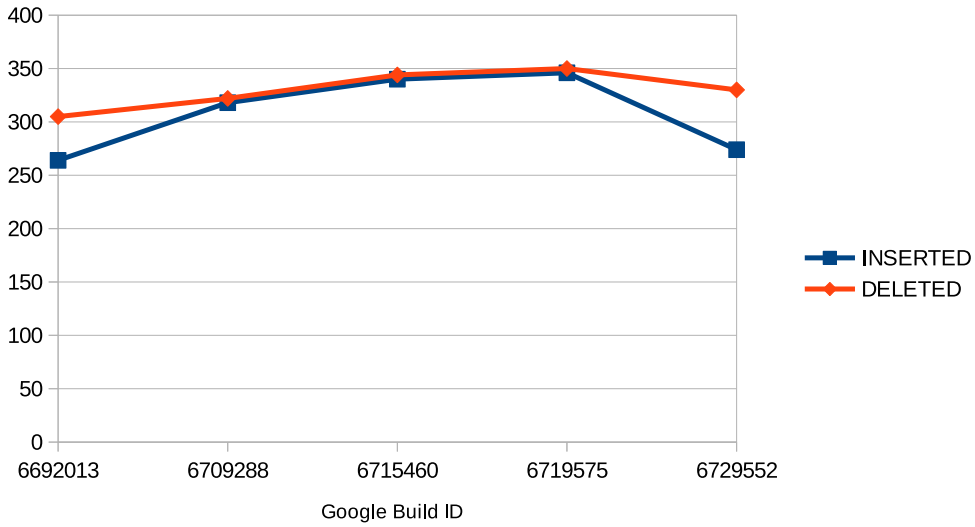

Figure 4.2: Lines of major difference for a range of generic builds

5 Conclusions and Outlook

Overall AOSP can not get built in a reproducible manner (RQ-R). Only some minor artifacts (section 4.2) satisfy the hard requirement for bit identical results. Unfortunately we cannot declare AOSP as an accountable build either (RQ-R-D). While we were able to find explanations for the majority of differences (accountable differences, see section 4.3), some eluded our understanding (unaccountable differences, as listed in section 4.4). It is quite clear that device builds exhibit more differences than generic ones (RQ-R-CBT). This is reflected both in our qualitative and quantitative analysis. Only section 4.4.6 covers GSI exclusive differences and in combination with a few other subsections accounts for nearly all of them, whereas there are seven device specific subsections. This is underlined by the quantitative analysis (1435 vs. 264 major lines added; 1812 vs. 305 major lines removed) for device and generic build flows. Finally the number of major line differences over time is inconclusive overall (RQ-R-COT).

The answer to RQ-BT can be found over the entire Chapter 3 and can be declared a success overall. With SOAP we succeeded in the creation of simple tooling, that given a few basic parameters, generates not just Android build artifacts, but also performs a thorough comparision with official Google build artifacts. However, we also noted that the current implementation has deficiencies. For example, our scripts can not handle cocurrent access to the local AOSP source code copy (even in case of the Docker container). The manual extraction and analysis of `apex_payload.img` is a workaround in search of a better solution. These and others are starting points for future work on SOAP.

Bibliography

[1] *A Brief Introduction to Authenticode*. Microsoft. 2020. URL: https://docs.microsoft.com/en-us/windows/win32/seccrypto/time-stamping-authenticode-signatures#a-brief-introduction-to-authenticode (visited on 07/04/2020).

[2] *Android CI Dashboard by Google*. Google. 2020. URL: https://ci.android.com (visited on 03/13/2020).

[3] *Android Open Source Project - Android Developer Codelab*. Android Open Source Project. 2020. URL: https://source.android.com/setup/start (visited on 04/28/2020).

[4] *Android Pony Express (APEX) file format*. Android Open Source Project. 2020. URL: https://source.android.com/devices/tech/ota/apex (visited on 07/11/2020).

[5] *Android Sparse Image format*. Android Open Source Project. 2020. URL: https://source.android.com/devices/bootloader/partitions-images#sparse-image-format (visited on 07/11/2020).

[6] *Best practices for writing Dockerfiles*. Docker Inc. 2020. URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (visited on 07/14/2020).

[7] *capabilities(7) Linux User's Manual*. https://www.mankier.com/7/capabilities. Feb. 2018.

[8] *Debian Repository Format*. Debian Project. 2020. URL: https://wiki.debian.org/DebianRepository/Format#Size.2C_MD5sum.2C_SHA1.2C_SHA256.2C_SHA512 (visited on 07/05/2020).

[9] *Debian Wiki: Identified problems, and possible solutions*. Debian Project. 2020. URL: https://wiki.debian.org/ReproducibleBuilds/Howto#Identified_problems.2C_and_possible_solutions (visited on 04/24/2020).

[10] *diffoscope - In-depth comparison of files, archives, and directories*. Reproducible Builds project. 2020. URL: https://diffoscope.org/ (visited on 07/04/2020).

[11] *Docker Issue: created loop devices do not appear in container even when run privileged*. Avi Deitcher, Justin Cormack. 2020. URL: https://github.com/moby/moby/issues/27886 (visited on 07/14/2020).

[12] *Factory images for Google phones (Pixel, Nexus)*. Google. 2020. URL: https://developers.google.com/android/images (visited on 03/13/2020).

[13] ITU-T. *ASN. 1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)*. Recommendation X.690. Geneva: International Telecommunication Union, Aug. 2015. URL: https://www.itu.int/rec/T-REC-X.690-201508-I/en.

[14] Saif Al-Kuwari, James H. Davenport, and Russell J. Bradford. *Cryptographic Hash Functions: Recent Design Trends and Security Notions*. Cryptology ePrint Archive, Report 2011/565. https://eprint.iacr.org/2011/565. 2011.

[15] *LineageOS Android Distribution - Usage Statistics*. The LineageOS Project. 2020. URL: https://stats.lineageos.org/ (visited on 04/20/2020).

[16] *Mobile Operating System Market Share Worldwide*. Statcounter. 2020. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 04/25/2020).

[17] *Mobile operating systems' market share worldwide from January 2012 to December 2019*. Statista, Inc. 2020. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (visited on 04/25/2020).

[18] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1271–1287. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin.

[19] *PE Executable Format*. Microsoft. 2020. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 06/01/2020).

[20] *Reproducible Builds*. Reproducible Builds project. 2020. URL: https://reproducible-builds.org/ (visited on 04/24/2020).

[21] *Reproducible Builds - BUILD_PATH_PREFIX_MAP*. Reproducible Builds project. 2020. URL: https://wiki.debian.org/ReproducibleBuilds/BuildPathProposal (visited on 06/01/2020).

[22] *Reproducible Builds - Introduction*. Reproducible Builds project, Debian. 2020. URL: https://wiki.debian.org/ReproducibleBuilds/Howto#Introduction (visited on 07/05/2020).

[23] *Reproducible Builds - Recommendations for timestamps*. Reproducible Builds project, Debian. 2020. URL: https://wiki.debian.org/ReproducibleBuilds/Howto#Files_in_data.tar_contain_timestamps (visited on 06/01/2020).

[24] *Reproducible Builds - RPATH tags in ELF binaries*. Reproducible Builds project, Debian. 2020. URL: https://wiki.debian.org/ReproducibleBuilds/Howto#RPATH_tags_differ (visited on 06/01/2020).

[25] *Reproducible Builds - SOURCE_DATE_EPOCH*. Reproducible Builds project. 2020. URL: https://reproducible-builds.org/docs/source-date-epoch/ (visited on 06/01/2020).

[26] *Sign your app*. Android Open Source Project. 2020. URL: https://developer.android.com/studio/publish/app-signing (visited on 07/04/2020).

[27] *The state of Enterprise Open Source Report*. RedHat. 2020. URL: https://www.redhat.com/en/enterprise-open-source-report/2020 (visited on 04/20/2020).

[28] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. "in-toto: Providing farm-to-table guarantees for bits and bytes". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[29] Elaine J Weyuker. "Evaluating software complexity measures". In: *IEEE Transactions on Software Engineering* 14.9 (1988), pp. 1357–1365.

[30] *Zip Standard, version 6.3.7*. PKWARE Inc. 2020. URL: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT (visited on 06/01/2020).

[31] Horst Zuse. *Software complexity: measures and methods*. First. Vol. 4. Walter de Gruyter GmbH & Co KG, 1991.