Lappeenranta University of Technology

School of Engineering Science

Intelligent Computing Major

Maria Glukhova

TOOLS FOR ENSURING REPRODUCIBLE BUILDS FOR OPEN-SOURCE SOFTWARE

Examiners: Associate Professor Arto Kaarna

Senior lecturer Vitaly Bragilevsky

Supervisors: Associate Professor Arto Kaarna

Senior lecturer Vitaly Bragilevsky

Aidin Hassanzadeh

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Intelligent Computing Major

Maria Glukhova

Tools for Ensuring Reproducible Builds for Open-Source Software

Master's Thesis

2017
36 pages, 7 figures.

Examiners: Associate Professor Arto Kaarna
Senior lecturer Vitaly Bragilevsky

Keywords: reproducible builds, open source software, free software, software development, Debian and Linux

The Reproducible Builds project is a collective effort of multiple open-source software projects, aiming to provide a verifiable path from human readable source code to the binary code used by computers. To achieve this goal, several tools were created, allowing for identifying common sources of unreproducibility in build process. In this work, an overview of the Reproducible Builds project and the tools designed is made. One of the tools, named diffoscope, is discussed in details; several improvements to this tool are made as part of this work.

Acknowledgements

The work presented in this thesis is done as part of my Outreachy internship with Debian. I would like to express my deepest gratitude to Software Freedom Conservancy, who run that program, helping women and members of underrepresented groups to start their journey in the world of Free and Open Source Software, and Debian for sponsoring my participation in it.

I would also like to thank Reproducible Builds team for being such a friendly and helping community. I hope my work will let more people know about this project, so together we can achieve our shared goal. I believe the work this project is doing is of great importance to all the software world, and I shall assist it in any way I can.

Special thanks to my mentors, Mattia Rizzolo and Chris Lamb, for all the guidance and help!

Maria Glukhova

May 2017

Lappeenranta, Finland

Contents

LIST OF SYMBOLS AND ABBREVIATIONS

APK	Android Package
ASCII	American Standard Code for Information Interchange – a character encoding standard
BTS	Bug tracking system
BZIP2	free and open-source file compression program that uses the Burrows–Wheeler algorithm
DEX	Dalvik Executable File
EXIF	Exchangeable Image File Format
FOSS	Free and Open Source Software
FUSE	Filesystem in Userspace
HTML	Hypertext Markup Language
ICC	International Color Consortium
ICO	Image file format for computer icons in Microsoft Windows
JPEG	Joint Photographic Experts Group - method of lossy compression for digital images
OS	Operating System
PNG	Portable Network Graphics - raster graphics file format that supports lossless data compression
TAR	(t)ape (ar)chive - computer software utility for collecting many files into one archive file
TLSH	Trend Micro Locality Sensitive Hash
Tor	Free software for enabling anonymous communication
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XZ	lossless data compression program and file format
ZIP	ZIP archive file format

1 Introduction

1.1 Background

One of the main ideas behind Free and Open Source Software (FOSS) is that anyone who uses the software can access the source code to make sure it does not perform any suspicious actions. However, in practice software products are often used in form of build artifacts, which are results of software build process. These artifacts are usually not human-readable, rather consisting of binaries intended to run on the user machine or compiled libraries. Thus, they cannot be inspected to ensure quality and security of the software.

The described approach introduces a potential flaw in quality assurance, where the result of the build may be different from expected due to some changes on the build stage, either intended or accidental. That flaw can be exploited in order to break software security, which is something that most open source software projects take very seriously. Possibility of changes being introduced by build process also have a negative impact on development process, since it becomes impossible to tell for sure what changes have been introduced by the changes in the source code, and which were caused by build process. To solve this problem, people who develop and distribute software must ensure that their build process is deterministic with respect to different build environments.

The Reproducible Builds project is a collective effort of multiple free software projects concerned about quality and security of their products. The goal of the project is to ensure that the software built under different environments produces exactly the same results. If that goal is achieved, people who build the software would be able to compare results between each other, and if someone's results are different, that would be a warning message meaning their build system could be compromised.

To help software developers achieve reproducibility of their products, several tools were created by the Reproducible Builds team. These tools are designed for testing reproducibility of software, comparing build results and identifying common issues.

This study describes how the reproducibility issues are identified, categorized and dealt with by Reproducible Builds project, and the tools used in this process. In particular, this thesis describes diffoscope, a tool that performs in-depth comparison for many file formats used in software development. This tool is widely used to identify reproducibility issues affecting software. As part of this work, several features were added to diffoscope, enhancing its performance on specific types of files.

1.2 Objectives and delimitations

The practical goal of this thesis is improvement of the diffoscope tool, making it more useful for the Reproducible Builds project. This work focuses on improvements made for specific types of files, aiming to provide a more human-readable comparison results for them. Diffoscope would also benefit from speed improvements, but in this field the main task is making diffoscope run in parallel. The difficulty of this task renders it too big for the scope of the presented work.

Apart from the practical goal described above, this work aims to raise awareness of the build reproducibility problem and ways to solve it. For this reason, a substantial part of the report is dedicated to the details of the Reproducible Builds project, its motivation, goals, proposed methods and achieved results.

1.3 Structure of the thesis

Section 2 explains in detail what the Reproducible Builds project is. Short background of the problem and motivation for this collective effort are given. This section also includes definition of reproducible builds, and gives a short review of tools and solutions that the project has achieved so far. Section 3 gives more details about diffoscope, the tool developed as part of the Reproducible Builds effort. It provides general information about the tool, its development model and ways of distribution. Sections 4 and 5 describe the process of adding new features to diffoscope. Section 6 gives the overview of the current state of the Reproducible Builds project with focus on how the described tools are used in various situations. Section 7 summarises the work and gives a short overview on what could be the possible next steps to address the discussed problem.

2 Reproducible Builds for Open-Source Software

2.1 Motivation and history

With the increasing popularity of open-source software [3] and its adoption in large scale systems, the security and quality assurance of that software becomes a high-priority question. Commercial closed-source software projects rely on internal quality assurance methods, and users of their products are expected to trust results of such assurance. While it is a working model for many companies, it may lead to cases when known or deliberately introduced flaws in software are hidden from end-user, the most known example being 2015 Volkswagen emissions scandal [21].

FOSS, on the other hand, uses completely different approach. While there is often no person directly responsible of quality assurance, publicly available source code means that every interested user or company using that software, may examine the code for potential flaws, therefore assuring quality for themselves [14]. That model assumes that the source code ultimately defines the software product. However, in reality people usually use software products in forms of binaries, compiled libraries or other form of build artifacts, which are the results of software build process. In fact, it is more common for users to obtain software products in form of build artifacts distributed by someone who built it, rather than acquiring the source code and building it for themselves.

That consideration brings forth certain questions. Is the source code really the only thing that defines the output of a software building process? Do other factors, such as the configuration of the build system, affect it? How can we be sure these factors do not cause modification of the resulting binary? If the binary gets modified by varying build environment, then just ensuring the source code does not have any flaws does not guarantee the resulting binary does not have them, since these flaws could be introduced, whether intentionally or not, by a build system.

Open-source projects that place an extra emphasis on security and privacy are especially concerned with this possibility, as it may leave the products they are building vulnerable to cyberattacks. The Tor project was one of the first projects to ring the alarm in the 2013, alerting others of the problem and taking first steps in ensuring what soon would be named reproducible builds [17, 22]. Since then, a lot of free software projects acknowledged the problem and tried to come up with a solution. Individual projects concerned with quality assurance of software they are producing started researching possible sources of non-determinism in build process as well as ways to remove them, as can be seen on example of TrueCrypt [5].

An important role in the following actions taken to raise awareness of the problem was played by Debian, one of the most popular Linux distributions. Over the last two years, members of Debian community gathered these efforts under the flag of the project called Reproducible Builds [19], gave numerous talks on technical conferences on the matter [4, 24], and organized two summits dedicated to problem of reproducible builds. They also set up a continuous testing infrastructure [6], which checks reproducibility of Debian packages, and designed several tools helping to find differences between build artifacts and identify common issues leading to unreproducible builds.

2.2 Definitions

Representatives of projects taking part in the Reproducible Builds effort agreed on following definition of reproducible build [7]:

Definition 2.1. Reproducible build.
A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.

The relevant attributes of the build environment, the build instructions and the source code as well as the expected reproducible artifacts are defined by the authors or distributors. The artifacts of a build are the parts of the build results that are the desired primary output.

The following terms are used in this definition [7]:

- *Source code* is usually a checkout from version control at a specific revision or a source code archive.
- *Relevant attributes of the build environment* would usually include dependencies and their versions, build configuration flags and environment variables as far as they are used by the build system (e.g. the locale). It is preferable to reduce this set of attributes.
- *Artifacts* would include executables, distribution packages or filesystem images. They would not usually include build logs or similar ancillary outputs.
- The reproducibility of artifacts is verified by *bit-by-bit* comparison. This is usually performed using cryptographically secure hash functions.
- Authors or distributors means parties that claim *reproducibility* of a set of artifacts. These may be upstream authors, distribution maintainers or any other distributor.

It can be seen from the definition that reproducibility is an attribute of a software product as a whole, as it is affect the build process, that is the link between the source code and the distributed binaries. If the software builds into different binaries when built twice, one cannot assume one of the versions is “right” and another one is “wrong” – instead, it can be concluded that the whole software has reproducibility issue that needs to be resolved by altering either the source code or the predefined way this software is built.

As described in the documentation section of [19], it is not realistically possible, or desirable, to mandate that the same source code is turned into the same sequence of bytes in all situations. The output of a compiler is likely to be different from one version to another as better optimizations are integrated all the time. Instead, reproducible builds happen in the context of a build environment. It usually comprises the set of tools, required versions, and other assumptions about the operating system and its configuration. A description of this environment should typically be provided alongside any distributed binary package.

2.3 Testing reproducibility

In practice, reproducibility is tested using the following procedure:

- The software is built from the provided source the first time, and the result is saved.
- Some variations are introduced into the build environment.
- Build the software the second time from the same sources.
- Compare the results. If they are bit-by-bit identical, it can be assumed the software is reproducible with respect to the introduced variations.
- If the results are not identical, specific tools can be used to find out what exactly has changed between them, therefore helping to identify the underlying reproducibility issue.

The list of variations used by the Reproducible Builds project in the testing process includes date and time, build path, hostname, domain name, filesystem, environment variables, timezone, language, locale, user name, user id, group name, group id, kernel version, umask, CPU type and number of CPU cores.

2.4 Tools

A number of tools were designed by the Reproducible Builds community to help software projects check their products for reproducibility, find out what gets changed between two builds, identify and fix common unreproducibility issues.

- *diffoscope* is a tool for in-depth comparison of files, archives, and directories. In content of reproducibility testing, it is usually used on build artifacts to find what exactly is different between them, allowing for easier identification of reproducibility issues.

- *trydiffoscope* is an online version of diffoscope. It can be used to compare files without installing diffoscope with all its dependencies.

- *disorderfs* is an overlay FUSE (Filesystem in Userspace) filesystem that deliberately introduces non-determinism into filesystem metadata. Since quite a few reproducibility issues are related to the filesystem specifics – e.g. file order – disorderfs can be used to identify these issues.

- *strip-nondeterminism* is a library for stripping non-deterministic information, such as timestamps and file system order, from files. Usage of strip-determinism in the build process can be used to achieve reproducibility. However, it is designed as a quick fix tool, not as a permanent solution. In general, it is preferred that the software handles reproducibility issues internally, without resorting to a third-party tool every time it is built.

- *reprotest* is a tool for building the same source code under different environments to check if these changes led to changes in the resulting binaries. *reprotest* builds the same source code twice in different environments, and then checks the binaries produced by each build for differences. If any are found, then diffoscope is used to display them in detail for later analysis.

The Reproducible Builds project members have also come up with a `SOURCE_DATE_EPOCH` specification [15] to deal with reproducibility issues arising from the usage of timestamps in the build process. These issues arise when software packages embed compile-time timestamps into generated files. As the current time changes between builds, this approach leads to the differences in the binaries' contents.

The idea behind the `SOURCE_DATE_EPOCH` proposal is to modify build process of the software to use the environment variable `SOURCE_DATE_EPOCH` instead of the current time if that environment variable is set. That proposal was adopted by several build tools and individual packages[18], significantly decreasing the number of software packages affected by timestamp-related reproducibility issues.

3 diffoscope

3.1 General information

This work focuses on one of the tools being developed by the Reproducible Build project, diffoscope. diffoscope is a tool for in-depth comparison of files, archives and directories [8]. The motivation behind development of diffoscope was to have a tool telling what exactly have changed between two builds by comparing their results. It is widely used by developers to identify reproducibility issues.

diffoscope is free software licensed under the GNU General Public License version 3 or later. It is written in Python3 programming language. It is available in the form of Debian package (unstable distribution), Python package (PYPI), Homebrew package, Arch Linux package. Source code of diffoscope is also available from Git repository at [10]. There is also an online version of diffoscope [23], so users can try this tool without installing it on their system. Bugs and feature requests can be submitted and reviewed at Debian bug tracking system [9].

Community of diffoscope developers can be described as having onion model [1]. The project is being developed constantly by number of dedicated developers and welcomes contributions as well as bug reports and feature requests from everyone. diffoscope development follows the Open Source Development Model [13], with frequent small releases and constant quality improvements. These improvements are often not planned in advance and often done as result of resolving a bug or fulfilling a feature request.

In this work, several features were implemented within diffoscope, with focus on better support of various file formats and usability. These improvements, too, were mostly dictated by bug reports and feature requests received for diffoscope.

3.2 Comparing arbitrary files

diffoscope is expected to be run on two files, which the user wants to compare. Here is the simplified version of how the process of comparing two arbitrary files looks like. First, the quick check is made to determine if the files are identical, that is, their content is the same bit-for-bit. If they are, no further comparison is made and diffoscope reports no difference for that file. Otherwise, the files content is compared depending on their type. For that to work, the files are specialized – that is, their type is determined and the object of the corresponding class is created from the file. There are different tests used to determine if the file match certain class; usually, the decision is made based on the extension or a magic number of the file.

Depending on the type of the file, some external tool is called to extract the content of the file in a human-readable form. For example, for ICC files, containing color profiles, the `cd-iccdump` tool is used to get the description of the color profile in a text form. These tools are not part of diffoscope itself. Instead, they are listed as "recommended" for it in package managers so users are expected to install them along with diffoscope. That helps to maintain a modular system, allowing users to install only the needed tools. If the tool is missing, the detailed comparison will be skipped and a binary comparison will be used instead.

After the file is translated to a human-readable form using external tools, the result of the translation is then used to compare files using a tool called `diff`, which operates on the text inputs and find the lines and words that are different. Since the text here is the processed content of the files, `diff` essentially finds a difference in the files' content in the form that makes it easier to read for humans.

If the files can be seen as "containers" – i.e. they include other files in their contents – diffoscope additionally runs a recursive comparison over their content. To do this, the content files are matched against each other. If their names are the same, the match can be found easily. If not, diffoscope uses TLSH library[16] to find hashes of files and match the files with “closest” hashes. Trivial examples of containers include directories, archives and tarballs, but files like Debian `.changes` files or `.dex` files are also compared as containers.

There are a number of cases where detailed comparison does not work. These include the cases where file types are different, the required external tool is missing, there are no support for that type of file in diffoscope, the translation resulted in error or diffoscope could not find any difference in translated forms even though the files themselves are different. In any of these cases, diffoscope falls back to “binary comparison” – the content of the file is treated as binary data, hex dump of that data is made using `xxd` or similar tool and the results are compared using `diff`. This option is used as the last resort as the hex dumps of binary data are not really human-readable.

After the results of the comparison are acquired, one of the presenter modules is used to construct the easy-to-view result and return it to the user. By default, a text presenter is used, and the result is returned to the standard output. By providing additional flags, the user can choose another form of output, e.g. HTML or a restructured text. The user can also specify the name of the output file. trydiffoscope, Debian reproducibility testing server and other applications that show diffoscope output as web pages use HTML as it is easier for users to read and navigate.

4 diffoscope improvement requests

As the diffoscope is used by different projects, some of them may have the unique needs, e.g. for some specific file type comparison. Some other features, on the other hand, would improve overall functionality and usability of diffoscope and would therefore benefit all its users.

Object-oriented structure of the code makes it relatively easy to add support for new file formats. However, even for the file formats that are already supported, there is still work to be done in order to make output more informative and to detect differences that otherwise would be hidden. Bugs and feature requests are usually submitted and reviewed at the Debian bug tracking system [9]. In this work, several feature requests were fulfilled, mostly with focus on improved support for specific types of files.

4.1 Order-like difference

Sometimes it can be useful to know if the two texts differ only in line numbering. While these differences should still be reported, as they mean the build is not reproducible, there should be a comment telling the inputs vary only in line ordering.

4.2 APK files

APK (Android Package) is the package file format used by the Android operating system for distribution and installation of mobile apps. It is essentially an ZIP-archive containing compiled source code, resources used by the application, and special control files.
To retrieve human-readable information from APK files, external tool called `apktool` is used.
Feature requests for APK files comparison in diffoscope include:

- Handle APK metadata generated by apktool correctly.

apktool, aside from other files, generates `apktool.yml`. That file contains meta information about package, such as Android SDK version and compression type. It also contains name of the input APK file.

A request was made (bug #850501 on Debian bug tracking system (BTS) [9]) to use more appropriate name, identifying that information as APK metadata and not a part of package itself. It is also desirable that input filename be removed from this information, as the goal of comparison is compare file contents, not names.

- Add ZIP archive information.

  As APK files is just a specific type of ZIP archives, it is possible to extract ZIP archive specific information, such as access permissions, encryption status and type of compression, from them. A request was made (bug #850502 on Debian BTS [9]) to include comparison of that kind of information in diffoscope output for APK files.

- Handle `AndroidManifest.xml` properly.

  `AndroidManifest.xml` file is an obligatory part of every Android package. The manifest file provides information about the application to the Android system, which the system must have before it can run any code of the application [2].

  Currently, `AndroidManifest.xml` is included in diffoscope output twice: its undecoded version, recovered by apktool, in XML format and its original (decoded) version in binary format. It may be better to show only undecoded version in most cases, as binary version is not informative when differences were already reported in more human-readable format. This idea comes from the feature request #850758 on Debian BTS [9]

4.3 Image files

Diffoscope has support for four types of image files: JPEG, MS Windows Icon (*.ico), GIF and PNG. Among these, JPEG and ICO images are compared using img2text tool, converting image into ASCII-graphics, allowing for easy comparison. PNG files are handled by `svg` tool, which extracts both metadata and image content (in text format), and

GIF images are handled by `gifbuild`.

There are two main feature requests for image comparison:

- It would be useful to add metadata comparison for JPEG and ICO images. That would allow to detect differences in e.g. compression type, channel mode, color profile used, transparency setting.
- When HTML output option is used, it could be possible to display difference between image content in image format and not in pseudo-graphics.

4.4 Cross-container comparison

Unpacking archives to compare their contents and get to the bottom of what makes two files different is the main feature of diffoscope, since the software is usually distributed in the container format. diffoscope can process archives and containers of many kinds. However, it still expect both compared containers to have the same type.

Some users would prefer being able to compare container content even when their type differs. For example, if one version of package is packed into ZIP archive and the other into TAR container with BZIP2 compression applied, it would be preferable to still compare their content. Right now diffoscope does not have this feature – it would treat these two packages as completely different files and compare them as binary files (byte-for-byte comparison).

5 Implementation of the new features

5.1 Order-like difference

Of all the categorized issues, there are several that are identified by a different ordering in the two files. To easily identify these issues, ability to mark the files that are different only in the ordering of lines was needed.

The solution was to modify comparison of text files, adding the following algorithm of detecting order-like difference:

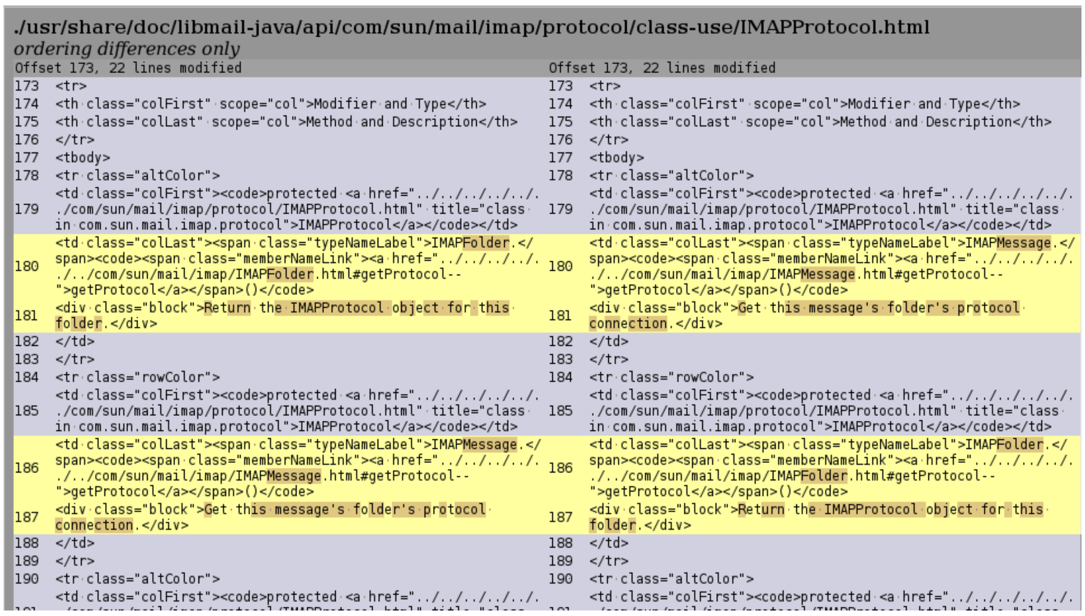
- Get the list of lines in the first file that are different from the second file and list of lines from the second file that are different from the first file.
- Compare the number of lines in both lists. If they are different, it can be concluded that the difference is not just in ordering of lines.
- If the number of lines is the same, sort them and compare the results element-wise. If that comparison revealed no difference, then the original difference was only in ordering of lines.

The new feature is limited to the plain text and some other file types that are treated as plain text (e.g. HTML, XML). It was decided there are no sufficient use cases to justify the wider range of supported formats.

Figure 5.1 shows how the HTML output looks like when run over two HTML files that are different only in the ordering of lines. Note the "ordering differences only" line on the top of the section.

5.2 Improved support for APK files

Diffoscope compares APK files using `apktool`. This tool tries to disassemble the package back to the project and resource files. The disassembled files are then compared using the suitable tool.


./usr/share/doc/libmail-java/api/com/sun/mail/imap/protocol/class-use/IMAPProtocol.html
ordering differences only

```
Offset 173, 22 lines modified
<tr>
<th class="colFirst" scope="col">Modifier and Type</th>
<th class="colLast" scope="col">Method and Description</th>
</tr>
<tbody>
<tr class="altColor">
<td class="colFirst"><code>protected <a href="../../../../.
./com/sun/mail/imap/protocol/IMAPProtocol.html" title="class
in com.sun.mail.imap.protocol">IMAPProtocol</a></code></td>
<td class="colLast"><span class="typeNameLabel">IMAPFolder.</
span><code><span class="memberNameLink"><a href="../../../../.
./../com/sun/mail/imap/IMAPFolder.html#getProtocol--
">getProtocol</a></span>()</code>
<div class="block">Return the IMAPProtocol object for this
folder.</div>
</td>
</tr>
<tr class="rowColor">
<td class="colFirst"><code>protected <a href="../../../../.
./com/sun/mail/imap/protocol/IMAPProtocol.html" title="class
in com.sun.mail.imap.protocol">IMAPProtocol</a></code></td>
<td class="colLast"><span class="typeNameLabel">IMAPMessage.</
span><code><span class="memberNameLink"><a href="../../../../.
./../com/sun/mail/imap/IMAPMessage.html#getProtocol--
">getProtocol</a></span>()</code>
<div class="block">Get this message's folder's protocol
connection.</div>
</td>
</tr>
<tr class="altColor">
<td class="colFirst"><code>protected <a href="../../../../.
```

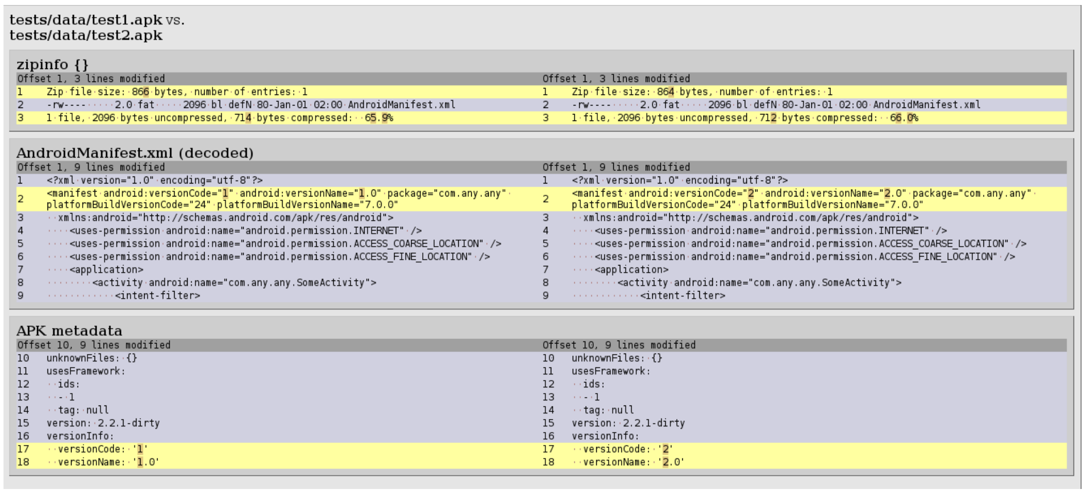
```
Offset 173, 22 lines modified
<tr>
<th class="colFirst" scope="col">Modifier and Type</th>
<th class="colLast" scope="col">Method and Description</th>
</tr>
<tbody>
<tr class="altColor">
<td class="colFirst"><code>protected <a href="../../../../.
./com/sun/mail/imap/protocol/IMAPProtocol.html" title="class
in com.sun.mail.imap.protocol">IMAPProtocol</a></code></td>
<td class="colLast"><span class="typeNameLabel">IMAPMessage.</
span><code><span class="memberNameLink"><a href="../../../../.
./../com/sun/mail/imap/IMAPMessage.html#getProtocol--
">getProtocol</a></span>()</code>
<div class="block">Get this message's folder's protocol
connection.</div>
</td>
</tr>
<tr class="rowColor">
<td class="colFirst"><code>protected <a href="../../../../.
./com/sun/mail/imap/protocol/IMAPProtocol.html" title="class
in com.sun.mail.imap.protocol">IMAPProtocol</a></code></td>
<td class="colLast"><span class="typeNameLabel">IMAPFolder.</
span><code><span class="memberNameLink"><a href="../../../../.
./../com/sun/mail/imap/IMAPFolder.html#getProtocol--
">getProtocol</a></span>()</code>
<div class="block">Return the IMAPProtocol object for this
folder.</div>
</td>
</tr>
<tr class="altColor">
<td class="colFirst"><code>protected <a href="../../../../.
```

Figure 5.1: Comparison of two HTML files with ordering-only differences.

There was a number of improvements made for APK files comparison.

- APK metadata, generated by `apktool` and written into the special file `apktool.yml`, is now handled separately from the rest of the files. It is renamed into “APK metadata” to reflect its contents; in addition, the APK file name itself is removed from the metadata, since the goal of diffoscope is to compare the contents of the file, regardless of their name.
- APK files difference now includes information about APK file as archive: level of compression, access rights etc. This is done using `zipinfo`, since APK files are essentially a kind of ZIP archive files.
- `AndroidManifest.xml`, a mandatory file for every Android package including the general information about it, has a better support now. Previously, these files were compared twice: once in their encoded form and once in the decoded plain text form generated by `apktool`. Now diffoscope tries to find the difference in the decoded `AndroidManifest.xml`, and falls back to the comparison of encoded ones only if the former approach does not reveal any differences.

Figure 5.2 shows how the HTML output looks like when run over two APK files. The



tests/data/test1.apk vs.
tests/data/test2.apk

zipinfo {}

```
Offset 1, 3 lines modified                                                      Offset 1, 3 lines modified
1    Zip file size: 866 bytes, number of entries: 1                             1    Zip file size: 864 bytes, number of entries: 1
2    -rw----     2.0 fat     2096 bl defN 80-Jan-01 02:00 AndroidManifest.xml   2    -rw----     2.0 fat     2096 bl defN 80-Jan-01 02:00 AndroidManifest.xml
3    1 file, 2096 bytes uncompressed, 714 bytes compressed:  65.9%              3    1 file, 2096 bytes uncompressed, 712 bytes compressed:  66.0%
```

AndroidManifest.xml (decoded)

```
Offset 1, 9 lines modified
1    <?xml version="1.0" encoding="utf-8"?>
2    <manifest android:versionCode="1" android:versionName="1.0" package="com.any.any"
     platformBuildVersionCode="24" platformBuildVersionName="7.0.0"
3      xmlns:android="http://schemas.android.com/apk/res/android">
4        <uses-permission android:name="android.permission.INTERNET" />
5        <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
6        <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
7        <application>
8            <activity android:name="com.any.any.SomeActivity">
9                <intent-filter>

Offset 1, 9 lines modified
1    <?xml version="1.0" encoding="utf-8"?>
2    <manifest android:versionCode="2" android:versionName="2.0" package="com.any.any"
     platformBuildVersionCode="24" platformBuildVersionName="7.0.0"
3      xmlns:android="http://schemas.android.com/apk/res/android">
4        <uses-permission android:name="android.permission.INTERNET" />
5        <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
6        <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
7        <application>
8            <activity android:name="com.any.any.SomeActivity">
9                <intent-filter>
```

APK metadata

```
Offset 10, 9 lines modified              Offset 10, 9 lines modified
10   unknownFiles: {}                    10   unknownFiles: {}
11   usesFramework:                      11   usesFramework:
12     ids:                              12     ids:
13     - 1                               13     - 1
14     tag: null                         14     tag: null
15   version: 2.2.1-dirty                15   version: 2.2.1-dirty
16   versionInfo:                        16   versionInfo:
17     versionCode: '1'                  17     versionCode: '2'
18     versionName: '1.0'                18     versionName: '2.0'
```

Figure 5.2: Comparison of two APK files.

diffoscope version in use has the described features included.

These changes are mostly useful for the F-Droid project [11], aiming to provide a collection of free Android applications. As part of the F-Droid project, the F-Droid Verification Server [12] was set up, with all the applications being tested for reproducibility automatically. Better support for APK files allows for a more informative diffoscope output provided for the application that fail these tests.

5.3 Improved support for image files

Since diffoscope is mainly used to compare packages, support for the images is needed mainly for resource files used in some software. Currently, supported formats include JPEG, ICO, GIF and PNG.

The main approach to comparison of images at the moment involves converting them into ASCII-pseudographics using `libcaca` and comparing the resulting text. That approach allows for identifying issues related to content of the image, but some information – such as palette, EXIF data, number of colors – might be lost. Considering this, the new feature, comparison of image metadata, was added to diffoscope.

Metadata comparison is enabled for JPEG and ICO images. Information about the image is retrieved using `ImageMagick` tool. The list of image properties extracted include e.g.

image format, file size, height, width, orientation, compression type, compression quality, colorspace, number of channels. Information is extracted in text form and then compared using text comparison.

In addition, in cases where diffoscope is producing HTML output, it is possible to output images without converting them to text form. In these cases, generating "visual difference", image, showing the difference in the visual form, can result in a more informative output then pseudographics allows. This behaviour was implemented in form of providing an additional field for the `Difference` objects, containing the visual difference images. These images are converted to `base64` form and embedded into the resulting HTML page using data URI protocol. Two types of visual difference are being generated for each pair of images at the moment:

- Pixel difference – generated using `ImageMagick compare` tool. This kind of images underlines all the pixels that have different values for the compared images. This often includes differences which are not normally noticeable by human eye, such as compression artifacts.
- Flicker difference – animation composed by alternating between the two images. Flicker difference is designed to provide a more human-perceptible way to view the difference, as it allows to notice the more significant changes between the two frames.

Currently, the visual difference is computed only for the images of the same size and with only one frame per image, meaning this feature does not support comparing animated GIFs.

5.4 Cross-container comparison

The problem of comparing different types of archives was considered to be best solved by alerting fuzzy-matching module. Diffoscope uses TLSH [16] library that implements fuzzy-matching using special hash values for files. By comparing hash values for different files, diffoscope is able to correctly recognize renamed files even when there is difference in the contents.

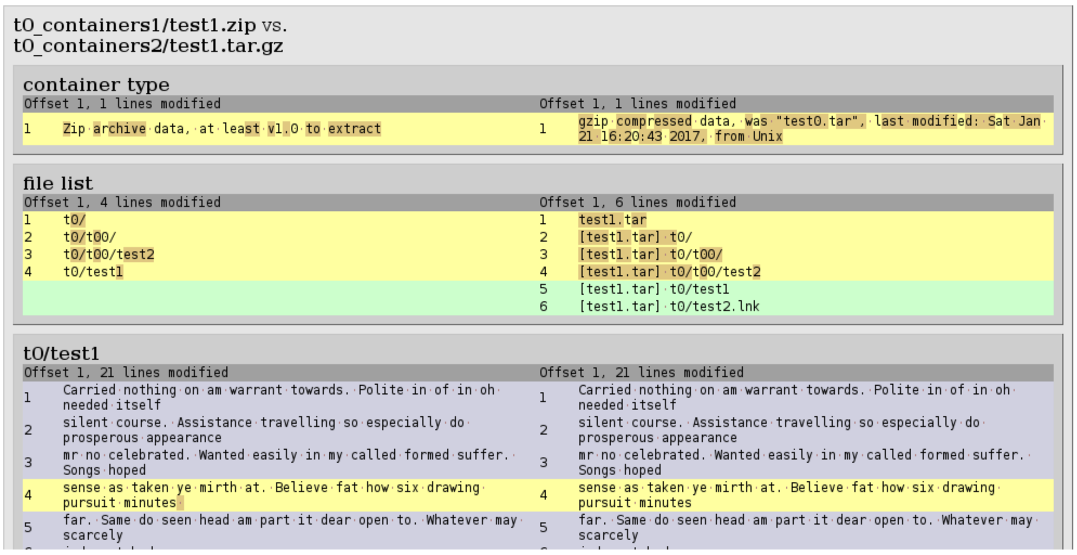
However, implementing the similar behaviour for containers was not a trivial task. If the container is considered a regular file and the hash is computed as for any other type of file, the result will be dependent on the type of the container, compression method, etc. Another option would be to compute the hash as combination of individual files' hashes. Unfortunately, that approach leads to unnecessary increase in computation time, since every container has to be extracted before finding the correspondence.

Even more important is the problem of comparing the nested containers. The most common use case is comparing compressed tarballs, e.g. `TAR.GZ` with `ZIP` archives. Both approaches described above would fail at this example, since not only the archive type, but the nesting level is different. However, both methods are quite common for software distribution, and it is desirable to have diffoscope compare their content, instead of stopping on the difference in the container type.

The proposed changes implement the following behaviour:

- If both compared objects are containers, but have different types, they are compared by finding difference in:
  - Type of container
  - List of contents
  - Content
- If one of the compared containers has one of following types: `gzip`, `bzip2`, `xz`, `dex`, it gets unpacked and the contents are compared. This can be safely done because all of these container types are guaranteed to have only one member. This way, the most common use case of nested containers is addressed.

Figure 5.3 shows how the HTML output would look like for different types of containers with the proposed changes accepted. In this example, the types of containers, their list



```
t0_containers1/test1.zip vs.
t0_containers2/test1.tar.gz

container type
Offset 1, 1 lines modified                                  Offset 1, 1 lines modified
1    Zip·archive·data,·at·least·v1.0·to·extract             1    gzip·compressed·data,·was·"test0.tar",·last·modified:·Sat·Jan·
                                                                 21·16:20:43·2017,·from·Unix

file list
Offset 1, 4 lines modified                                  Offset 1, 6 lines modified
1    t0/                                                    1    test1.tar
2    t0/t00/                                                2    [test1.tar]·t0/
3    t0/t00/test2                                           3    [test1.tar]·t0/t00/
4    t0/test1                                               4    [test1.tar]·t0/t00/test2
                                                            5    [test1.tar]·t0/test1
                                                            6    [test1.tar]·t0/test2.lnk

t0/test1
Offset 1, 21 lines modified                                 Offset 1, 21 lines modified
1    Carried·nothing·on·am·warrant·towards.·Polite·in·of·in·oh·    1    Carried·nothing·on·am·warrant·towards.·Polite·in·of·in·oh·
     needed·itself                                                      needed·itself
2    silent·course.·Assistance·travelling·so·especially·do·         2    silent·course.·Assistance·travelling·so·especially·do·
     prosperous·appearance                                              prosperous·appearance
3    mr·no·celebrated.·Wanted·easily·in·my·called·formed·suffer.·   3    mr·no·celebrated.·Wanted·easily·in·my·called·formed·suffer.·
     Songs·hoped                                                        Songs·hoped
4    sense·as·taken·ye·mirth·at.·Believe·fat·how·six·drawing·       4    sense·as·taken·ye·mirth·at.·Believe·fat·how·six·drawing·
     pursuit·minutes·                                                   pursuit·minutes
5    far.·Same·do·seen·head·am·part·it·dear·open·to.·Whatever·may·  5    far.·Same·do·seen·head·am·part·it·dear·open·to.·Whatever·may·
     scarcely                                                           scarcely
```

Figure 5.3: Comparison of two containers.

of content and the actual content of the files inside the containers differ between the two compared files.

The described changes for cross-container comparison are currently under review and therefore are not included in diffoscope yet.

6 Current state of the Reproducible Builds project

6.1 Debian

Debian continues its work towards ensuring that all of the packages in the distribution are bit-by-bit reproducible. To continuously check the status of reproducibility of individual packages, testing infrastructure is set up, reporting results on [6]. For the packages that are not reproducible for the tested variations, the diffoscope output is provided. That allows for an easier identification of underlying issues.

Currently, the work of Reproducible Builds project members within Debian is focused on three main areas:

- Continuous testing of Debian packages
- Categorizing the issues
- Fixing reproducibility bugs

6.1.1 Continuous testing of Debian packages

Debian reproducibility testing server [6] shows the status of reproducibility of Debian packages across several releases (stable, unstable, experimental) and architectures (amd64, i386, arm64, armhf). Reproducibility testing here means that the package is built two times, with some variations introduced between builds. For the packages that are not reproducible within this setup, diffoscope is run and its output is published, so it is easily accessible for package maintainers, Reproducible Builds project members and other interested parties. Reproducibility tests are re-run periodically, and the statistics about results of these tests is gathered.

Figure 6.1 shows the current status of reproducibility of Debian packages for unstable version of the distribution built for the amd64 target architecture. Note that the drop in number of reproducible packages in August, 2016 is connected to the new variation –

build path – being introduced for unstable and experimental distribution. It turned out quite a lot of packages are affected by build path storing issue, and since then, there was quite a lot of work within a project targeting this issue.

At the moment of writing, 23017 of Debian packages build reproducibly in the unstable distribution on the amd64 architecture under the described testing environment, equivalent to 87.6% of all the packages tested.

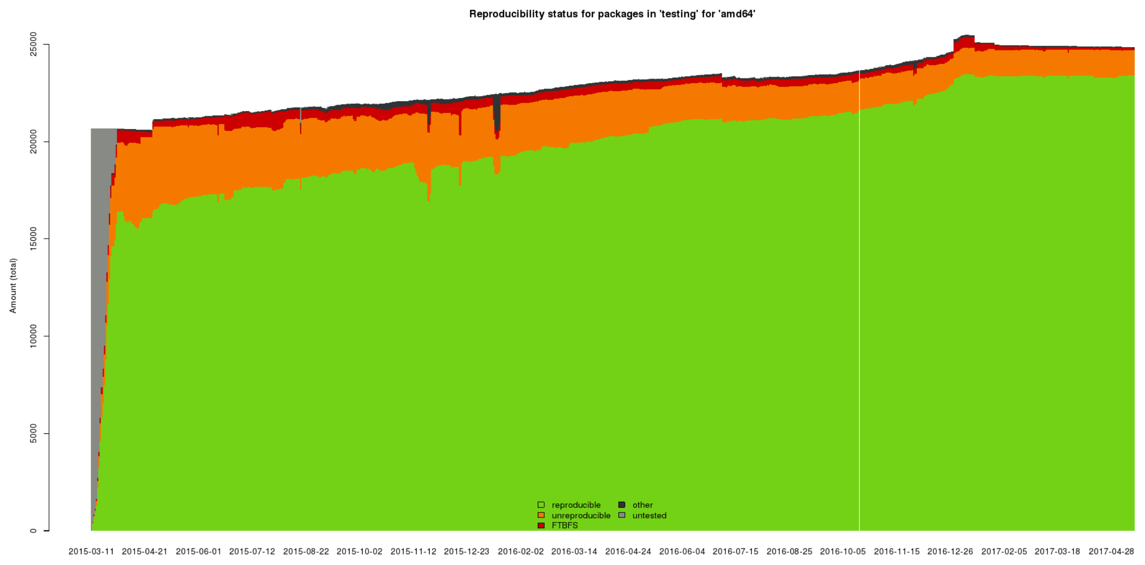


Figure 6.1: Overview of reproducible builds for packages in Debian unstable for amd64 architecture. [6]

6.1.2 Categorizing the issues

With diffoscope result for the packages available, Reproducible Builds project members try to identify the root issue causing reproducibility. It is often the case that the issue is not within the package itself, but rather in the tools used in the build process. Therefore, it is important to identify common issues and focus on resolving them in a unified way.

Two examples of the most common issues are:

- *Capturing build path* – a lot of packages store the path where the package was built, during the build phase. This is inconvenient since this information is rarely helpful to end users; usually, the package is built on some dedicated build server and not on a target computer. This issue is one of the main focus of Reproducible Build project members now, and they coordinate with the maintainers of packages as well as the build tools developers in order to address it.

- *Storing Timestamps* – while this issue was addressed by `SOURCE_DATE_EPOCH`, some packages still record the build date or time.

Figure 6.2 illustrates the changes in the number of identified issues. This number generally rises over time, as new issues are being found and the old ones are often categorized into smaller and more specific issues. Even if the issue is completely dealt with, it is often left in records if it is believed information about it could be useful to other issues or other projects.

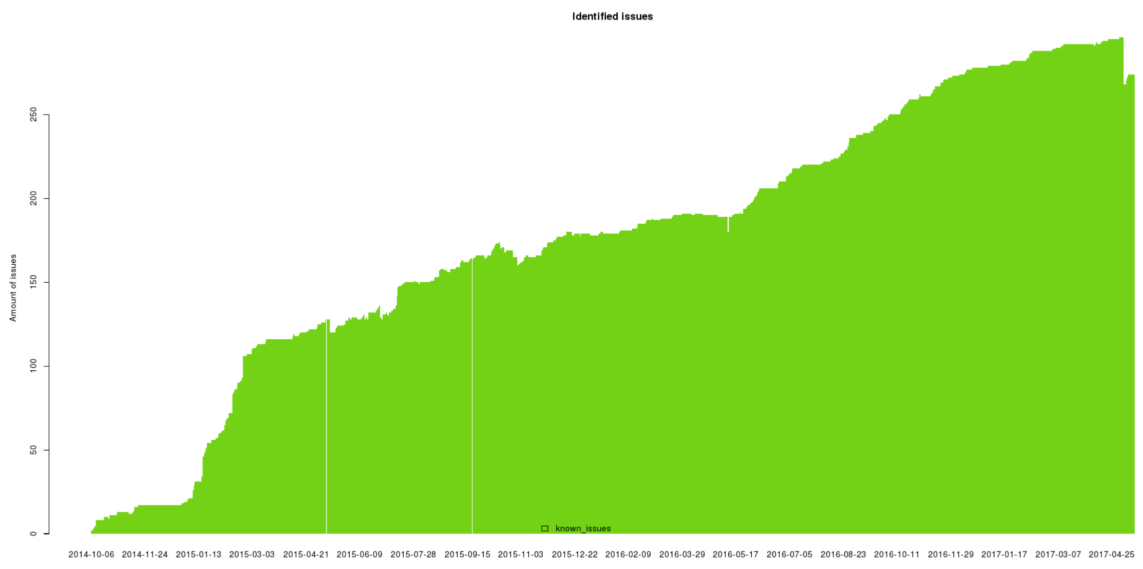
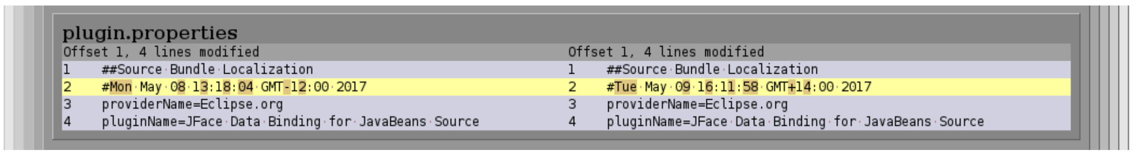


Figure 6.2: Number of categorized reproducibility issues. [6]

Figure 6.3 shows an example of how the diffoscope output looks like for the package that is storing the time of build and the timezone.



```
plugin.properties
Offset 1, 4 lines modified                                       Offset 1, 4 lines modified
1    ##Source Bundle Localization                                1    ##Source Bundle Localization
2    #Mon May 08 13:18:04 GMT-12:00 2017                         2    #Tue May 09 16:11:58 GMT+14:00 2017
3    providerName=Eclipse.org                                    3    providerName=Eclipse.org
4    pluginName=JFace Data Binding for JavaBeans Source          4    pluginName=JFace Data Binding for JavaBeans Source
```

Figure 6.3: Example of timestamp-related reproducibility issue.

Figure 6.4 shows statistics about number of packages with notes attached to them, meaning that there is an identified issue or some sort of comment for them attached to them. Once again, a massive change is observed at around August, 2016, as the build path variation was introduced, resulting in a lot of packages being labeled with build path recording issue.

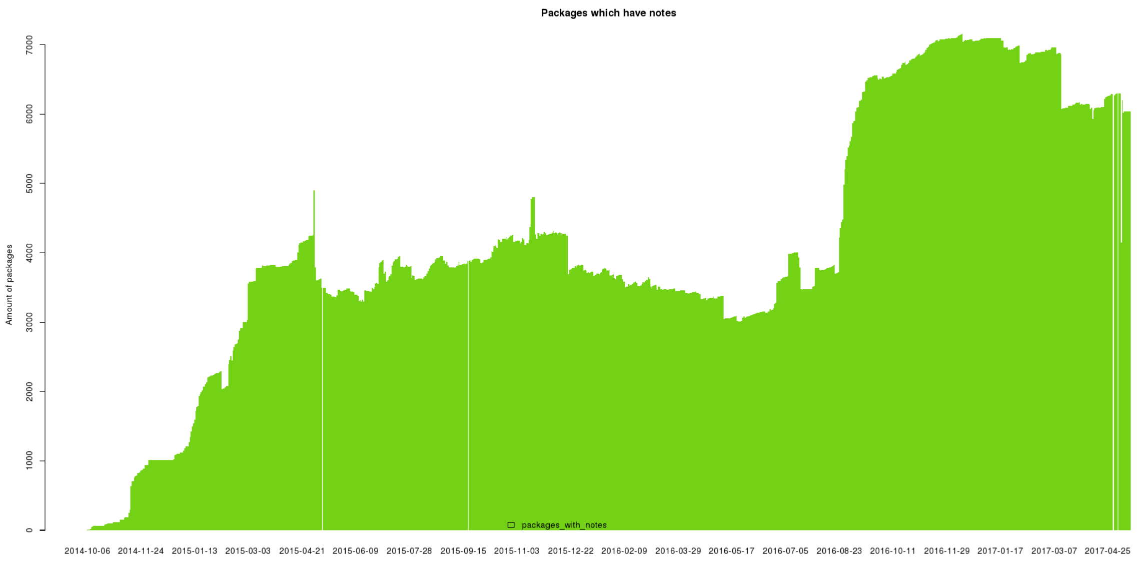


Figure 6.4: Number of packages that have notes. [6]

6.1.3 Fixing reproducibility bugs

While only package maintainers or developers actually have the means to fix their package, members of Reproducible Builds project try to help them by reporting the reproducibility issue. As a rule, they also provide a patch that is expected to fix the issue. Sometimes additional coordination with package maintainer or developer is required to ensure the patch fits well within the package and to clarify its meaning.

A lot of issues are related to the usage of the specific tool in the build process, and therefore it is usually preferred to focus on fixing these issues in the build tool in question, therefore helping all the packages using that tool to achieve reproducibility.

6.2 F-droid

The F-Droid project [11], that aims to provide an environment of free software for Android smartphones, has set up its own Verification Server [12]. It functions in the similar fashion as Debian test infrastructure, rebuilding packages and providing the diffoscope output when results do not match.

6.3 Other projects

FreeBSD and NetBSD distributions also continue their work on reproducible builds. Currently, they use less variations between builds than Debian does, but within these constraint, they have achieved significant progress: NetBSD reported 100% reproducibility within their build system in [25] and FreeBSD is at 99.6% at the moment [20].
Arch Linux, Fedora, Coreboot and OpenWrt are also tested along with Debian on reproducibility testing server [6], although at the lesser scope.

A lot of other projects has taken reproducible builds problem seriously and try to provide the users of their products with a way to verify their software is built reproducibly. The list of currently involved projects along with their recent progress can be found on the Reproducible Project website under “Who is involved” section [19].

7 Conclusions

Reproducibility of software builds is important problem, critical for ensuring quality and security of open-source software. With the software building reproducibly, users can easily ensure no flaws were added to the product during build process and that the software indeed matches its source code. This problem attracted attention in various open-source projects, but outside of open-source world it is still not discussed enough.

In this report, the history and motivation behind the Reproducible Builds project was presented. The definition of reproducibility in the software building process was given. An overview on the current reproducibility status of Debian operating system, as well as the steps taken to improve it, was discussed. Specifically discussed were the tools that the Reproducible Build project uses for testing reproducibility of software and identifying sources of unreproducibility. The particular tool for comparing two build outputs, named diffoscope, was discussed in detail, with emphasis on what can be done to improve it. The improvements for handling APK files, image files, containers and files with ordering-only difference were implemented and discussed as part of the work.

While the significant progress has been made in spreading the word about the reproducibility problem, testing various software and fixing key reproducibility issues, the work is far from being done. With the new variations introduced in the testing process and new projects joining the initiative, it is mandatory that the work on categorizing the issues and fixing them continues as well. To make this process easier for all the interested parties, the tools used in the process should be constantly evolving to match the arising needs.

The constant effort of people from different FOSS projects working on reproducible builds shows how highly the security and quality are valued in the world of open-source and how much is being done every day to keep the software we use secure.

References

[1] Mark Aberdour. “Achieving quality in open-source software”. In: *IEEE Software* 24.1 (2007), pp. 58–64.

[2] *Android API Guides - App Manifest*. https://developer.android.com/guide/topics/manifest/manifest-intro.html. Accessed: 2017-01-24.

[3] Gary Anthes. “Open source software no longer optional.” In: *Communications of the ACM* 59.8 (2016), pp. 15–17. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm59.html#Anthes16a;%20http://doi.acm.org/10.1145/2949684.

[4] Jérémy Bobbio (Lunar). *Reproducible builds for Debian*. https://archive.fosdem.org/2014/schedule/event/reproducibledebian/. Accessed: 2017-01-23. Feb. 2014.

[5] Xavier de Carné de Carnavalet and Mohammad Mannan. “Challenges and implications of verifiable builds for security-critical open-source software”. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM. 2014, pp. 16–25.

[6] *Continuous tests of Debian packages reproducibility*. https://tests.reproducible-builds.org/debian/reproducible.html. Accessed: 2017-01-23.

[7] *Definition of Reproducible Builds*. https://reproducible-builds.org/docs/definition/. Accessed: 2017-01-23.

[8] *Diffoscope*. https://diffoscope.org/. Accessed: 2017-01-23.

[9] *Diffoscope bug and feature requests*. https://bugs.debian.org/src:diffoscope. Accessed: 2017-01-23.

[10] *Diffoscope Git repository*. https://anonscm.debian.org/git/reproducible/diffoscope.git. Accessed: 2017-01-23.

[11] *F-Droid Project*. URL: https://f-droid.org.

[12] *F-Droid Verification Server*. URL: https://verification.f-droid.org.

[13] Ibrahim Haddad. *The Open Source Development Model: Overview, Benefits and Recommendations*. http://aaaea.org/Al-muhandes/2008/February/open_src_dev_model.htm. Accessed: 2017-01-24.

[14] Jaap-Henk Hoepman and Bart Jacobs. “Increased security through open source”. In: *Communications of the ACM* 50.1 (2007), pp. 79–83.

[15] Chris Lamb. *SOURCE_DATE_EPOCH specification*. https://reproducible-builds.org/specs/source-date-epoch/. Accessed: 2017-01-23.

[16] Jonathan Oliver, Chun Cheng, and Yanggui Chen. “TLSH–A Locality Sensitive Hash”. In: *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*. IEEE. 2013, pp. 7–13.

[17] Mike Perry. *Deterministic Builds Part One: Cyberwar and Global Compromise*. https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise. Accessed: 2017-01-23. 2013.

[18] Reproducible Builds Project. URL: https://wiki.debian.org/ReproducibleBuilds/TimestampsProposal.

[19] *Reproducible Builds*. https://reproducible-builds.org/. Accessed: 2017-01-23.

[20] *Reproducible FreeBSD?* https://tests.reproducible-builds.org/freebsd/freebsd.html. Accessed: 2017-05-03.

[21] Quirin Schiermeier. “The science behind the Volkswagen emissions scandal”. In: *Nature News* 25 (2015).

[22] Seth Schoen, Mike Perry, and Hans Steiner. *Reproducible Builds: Moving Beyond Single Points of Failure for Software Distribution*. `https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner`. Accessed: 2017-01-23. Dec. 2014.

[23] *Trydiffoscope: diffoscope online version*. `https://try.diffoscope.org/`. Accessed: 2017-01-23.

[24] Valerie Young. *Reproducible Builds for a Better Future*. Accessed: 2017-01-23. Jan. 2017.

[25] Christos Zoulas. *NetBSD fully reproducible builds*. `http://blog.netbsd.org/tnf/entry/netbsd_fully_reproducible_builds`. Accessed: 2017-05-03. Feb. 2017.