École Polytechnique Fédérale de Lausanne

Flatpak attestation using Reproducible Builds

by Zacharie Timothée Tevaearai

Bachelor Project Report

Flavio Toffalini
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

Abstract

Software can be built from source code or distributed as pre-compiled packages. These packages are the result of a software supply chain which can be subject to attacks or bugs and cannot be trusted. A way to attest that a product from such a supply chain corresponds to its original source code is called reproducible builds and consists in performing the exact build steps, for a certain package, independently of the main supply chain and comparing the rebuilt artifact with the original one. If they are bit by bit identical, then we prove that the path between the source code and the final artifact is legitimate. Here we look at the possibility of using reproducible builds with flatpak, a packaging technology used to ship the same binaries on multiple Linux distributions. In particular, we focus on flathub, its main repository. We present a tool, called flatpak-rebuilder, which rebuilds a package available on flathub, by recreating a minimal environment. We then measure the amount of reproducible packages, and find out that it represents 42% of packages on flathub. We then do further analysis to understand why they are not reproducible and conclude that we can reach 60% of reproducible programs by providing extra information to recreate the environment, the key one being the time at which the build take place.

Contents

Chapter 1

Introduction

In theory, with free and open source software (FOSS) we can audit the code and compile only software we trust. In reality most software are shipped as pre-built binaries to end-users, through a software supply chain. In this context how can we be sure that the resulting artifact is produced from the original source code ? Indeed, the machine on which the software was compiled might be compromised, a bug may occur during the process, or some maintainer may manually add a malicious patch to the final product. There are all sorts of reasons to not trust what is shipped to the end-users. Supply chain attacks can be conducted on both open source and closed source software, and typically consist in injecting malicious code into an existing product. Examples include the SolarWinds exploit from 2020, where attackers had compromised the update publishing infrastructure of Orion, one of SolarWinds software, and were shipping a modified version with a malware injected [13]. In the FOSS world, attacks targeting package manager such as npm or RubyGem occurred and are already studied in the literature [6]. Techniques exist to counter such attacks, and the one we use in this project is called *reproducible builds* (R-B). It is a set of practices to make sure we can, independently of the supply chain, obtain the same package, bit by bit, starting from the source code, by reproducing the exact same build steps. The exact definition for a build process to be reproducible, as stated by Chris Lamb and Stefano Zacchiroli, is as follows:

Definition 1.0.1 (reproducible build)**.** *"The build process of a software product is reproducible if, after designating a specific version of its source code and all of its build dependencies, every build produces bit-for- bit identical artifacts, no matter the environment in which the build is performed."* [5]

If a piece of software is reproducible we can therefore attest its integrity, by reproducing it on our side and comparing it with the one resulting from an untrusted build process. For example, we can use it with a Linux distribution. Here users download, through a package manager, programs which are built by the distribution vendor. A hypothetical reproducible builds setup, like in Figure 1.1, would go as follows: The source code is produced upstream and is published on platform such as GitHub or GitLab. Maintainers then build it on some infrastructure, with certain dependencies and a specific tool chain. They sign and send the resulting artifact to the end-user. This process, as shown before, cannot be trusted. On the other hand upstream code can be audited and verified manually. Now, the maintainers, alongside the build, can record and publish the necessary information to redo the build with the same conditions. This information will include, among others, the exact dependencies and which version of the tool chain were used. From these, a set of trusted rebuilder will redo the build and publish the checksum of the artifact they obtain. When the end-user download a program from the distribution's repositories, they can compare the checksum they have with the one given by the rebuilders and can then decide to trust or not what they just downloaded. The exact process of deciding if the artifact is legitimate by comparing the checksums can be automated, and computed differently depending on the number of independent rebuilder and how much we trust them. Using reproducible builds we can prove that the path between upstream and downstream is legitimate. Since we can audit and trust the source code, this trust can propagate to the build artifact, without the cost of auditing binaries. An attacker would have to take over not only the supply chain, but also a certain amount of these independent rebuilder (depending on how we compare the checksums), reducing the feasibility of such attacks. Integrity also proves us that no bugs or errors occurred during the build. The practicality of using reproducible builds to provide integrity for a software supply chain has already been shown, for instance on Debian, where 82.5% of packages on the unstable branch are reported as reproducible as of June 10, 2022 [4].

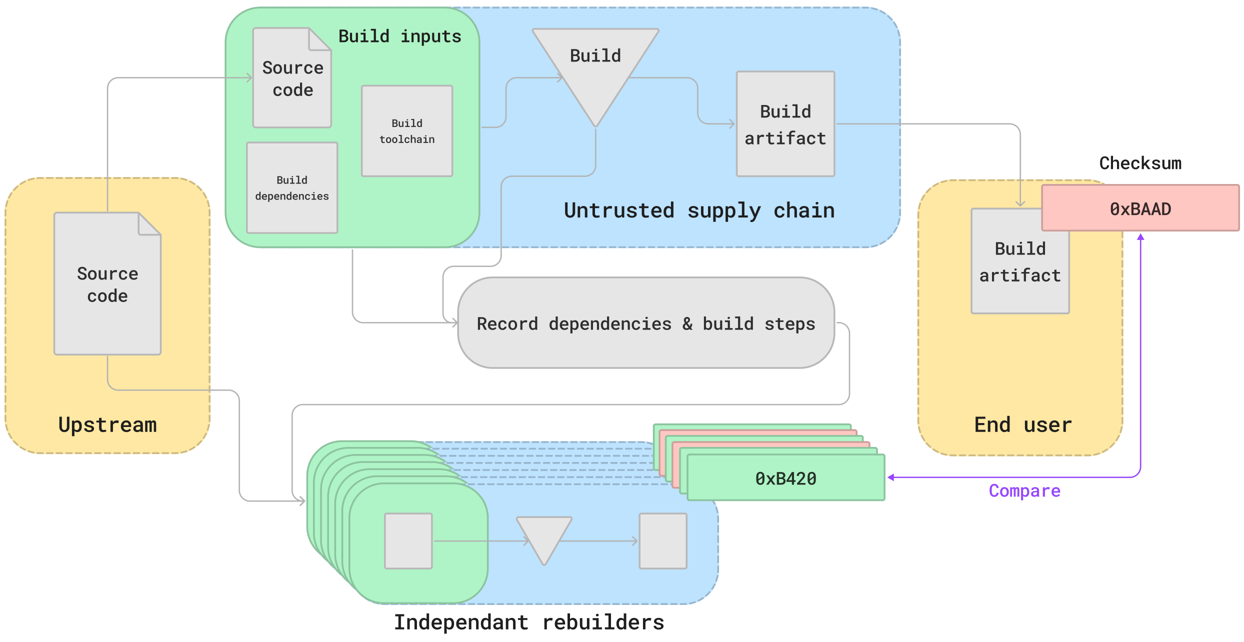


Figure 1.1: A reproducible builds setup. Here the end user can compare the checksum of the package they download, with a set of independent rebuilders. Since in majority the checksums do not match, they reject the package. Adapted from [5].

One software distribution system that can benefit from reproducible builds is *flatpak*, and more precisely its main repository called *flathub*. Flatpak is way to build, distribute and deploy applications compatible on most Linux distributions. It works by running them inside a container (which is referred to as a sandbox) where they have the minimum permissions and are as much isolated as possible from the host system. Since the build can also be performed inside the sandbox, it is an ideal environment to use reproducible builds. However, there are no tools to redo the build of a flatpak from flathub, and therefore there are no statistics on how many flatpaks are reproducible. In this project, we create a tool called *flatpak-rebuilder*, to allow individual to redo a build of a flatpak from flathub, and attest if it produces the same artifact or not. Using flatpak-rebuilder we then measure the amount of reproducible programs on flathub, and provide improvement to the flathub build infrastructure, in order to increase this number. Since flatpak works across Linux distributions, flatpak-rebuilder needs to be as portable. This limits the use of strong container technologies, such as Docker, or tools such as `faketime` to better simulate the build environment. After rebuilding 729 programs, which is about half of flathub, we have 42% reproducible programs, which is much lower than what is found on Debian. This is mainly due to the lack of certain information, such as the time at which the build originally took place. Consequently, we propose several improvements to *flatpak-builder*, the main tool used to build flatpak on flathub, which is also the one capable of recording some missing build information. We also introduce new notions to characterize how reproducible a program is, such that we can still build trust in programs which are only partially reproducible.

The structure of the report is the following, chapter 2 explains all the concept used in the next chapters. In chapter 3 and chapter 4 we describe the overall functioning of flatpak-rebuilder. In chapter 5 we measure the reproducibility of flathub and provide a manual analysis of the main causes of non-reproducibility. Finally, in chapter 6 we discuss and compare our approach to other similar tools, and also compare reproducible builds as a whole to other techniques to secure the supply chain.

1.1 Motivation

No tools exist to easily redo a build of a flatpak from flathub, consequently there are no measurement of how many flatpaks are reproducible. This implies that there are no knowledge about issues, specific to flatpak, concerning the feasibility of reproducible builds there. The main goal, is to have a first idea of the problems of applying reproducible builds to flatpak and then to use that knowledge to further improve the situation. The challenge is that since no one really tried to see if flatpaks are easily reproducible or not, we will not have access to all the information to properly recreate a build environment which resemble the one used to build programs on flathub. Indeed, on Debian for instance, it is only after they started to measure the amount of reproducible program on their repositories, that they could improve the situation. And as a matter of fact, they started with fairly low ratio of reproducible programs, at the beginning (roughly 60%) [4]. We therefore expect in this project to reach a fairly low amount of reproducible program, but also to find ways to greatly improve it.

Chapter 2

Background

To better understand this project, we should introduce a bit more reproducible builds, especially the most common reasons a program may not be reproducible. From 1.0.1, we conclude that a build is not reproducible when after performing the exact same build steps, with the same build dependencies, we reach another result. One reason it could happen is when a program depends on more than just its build dependencies. Typical examples are build process affected by the time or the path in which the build took place. This is what C. Lamb refer to as *uncontrolled build input*, because the build is affected by additional inputs which are not under our control. In practice some of these issues (such as time) can be solved if we include these as extra dependencies to the original definition of reproducible builds. It can be tempting to put the whole environment as a dependency (kernel version, CPU model, etc.) but doing so will make the rebuilding step much harder and will slow down build time, which would not make reproducible builds a practical solution. To mitigate the effects of *uncontrolled build input*, we rely on another definition of reproducible builds:

Definition 2.0.1 (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."* [12]

This definition come from the *Reproducible Builds project* (R-B project), an organization which promotes reproducible builds. A good example of "relevant attributes of the build environment", are the `.buildinfo` files on Debian [3]. Those are files associated to a package, describing the environment in which the build took place. Not everything in a `.buildinfo` is necessary to perform a rebuild (for instance the kernel versions), but certain information, such as the time or the active environment variables are easy to manipulate and can be considered as valid additional input for the build process. In particular, time can be controlled by overriding the SOURCE_DATE_EPOCH environment variable. It is defined as part of the R-B project, and can be consumed by build systems to make sure the resulting build looks like it was done entirely at the time specified by the variable [9]. Other than uncontrolled build inputs, there is another class of problems affecting the reproducibility of a program, which is *build non-determinism*. It occurs when some parts of the build are random. One example is the serialization of frozenset items in Python, which does not happen in a deterministic order [11]. Other non-determinism issue can be for instance related to the parallelism of the build process, or to filesystem ordering, since both do not follow a strict order. Even though work still need to be done in order to have deterministic builds for certain build systems, such as with Java [15], the most common one can behave deterministically.

The other main concept to introduce is flatpak and flathub, more specifically how a flatpak is built and shipped on flathub. As stated before, flatpaks are deployed and built inside a sandbox, a container mostly isolated from the host system. Extra permissions, such as access to certain directories, access to the network and so on, are managed in the metadata of a flatpak. There are no general build process for a flatpak. As long as the build directory respect certain conventions, and the necessary metadata are accessible at the root, it is a valid flatpak. The way flatpak are shipped and deployed is through OSTree. It is a library, working similarly as git, but optimized to handle binary objects. Furthermore, it is specialized to handle entire file system trees. Flatpak uses it in the following way, the build directory of a program is committed through OSTree and can be push to a remote and pulled by end-users. OSTree is used to deploy a read-only, using hard links, version of the flatpak, which is mounted inside the sandbox (at /app). Since flatpaks are isolated from the host, they cannot access its libraries, such as libc. Instead, those are bundled with the flatpak and therefore also mounted in the sandbox. While this works and ensure a program built against one library will not suddenly run against another version, this has multiple issues. First it is not disk efficient, for instance a lot of programs will ship with libc. Secondly, security critical libraries such as libssl need regular updates, and having each flatpak shipping their own copy increases the risk of having some which are out of date. To address these issues, flatpak comes with a simple shared dependencies systems called *runtimes*. A flatpak will always depend on a specific runtime (specified in its metadata), which is another flatpak containing the most common and security critical dependencies. The runtime is also mounted in the sandbox, at /usr. Another runtime, called the *SDK*, exists, it just includes additional dependencies only useful at build time, or for debugging purposes, such as gdb and gcc. A flatpak can also depend on a *base-app*, which is a flatpak containing certain big build dependencies, for certain type of applications, such as those based on Electron. Finally, a last type of dependencies are extensions, for SDK or base-app, which are just additional libraries or programs that can be optionally mounted. An example is the Rust tool chain, which is available as an SDK-extension, such that only rust programs needs to include it at build time. The build process is not standardized, a flatpak can be built using any build system available. However, a helper tool called flatpak-builder exists and is used on flathub. It parses a description of the build, called a manifest file, and produces the resulting flatpak. The manifest is separated in modules, where each describes how to build one dependency or the final program, and each specifies how to fetch the necessary build dependencies. The manifest also includes which runtime and SDK need to be used and which permissions need to be applied to the resulting flatpak. Flathub, the main repository for flatpak, is a community based project, where each application has a GitHub repository under the flathub organization. These repositories contain at least the manifest file to build the app using flatpak-builder. The flathub buildbot is a bot which after any commit to the master branch of an application will fetch and build the manifest file. The resulting artifact is committed using OSTree and pushed to the flathub remote.

Chapter 3

Design

This section describes the high-level functioning of flatpak-rebuilder, it is summarized in Figure 3.1.

3.1 Recreating the environment

To rebuild a program, flatpak-rebuilder must first make sure to gather all build dependencies to the exact version used during the original build. There are two types of dependencies defined in a flatpak-builder manifest file, the one which are embedded in the final package, those are defined in the different modules of the manifest and their version is always perfectly specified. For instance a dependency on a git repository will come with the hash of the commit. Those dependencies are therefore easy to gather, and it is already handled by flatpak-builder. The other type of dependency is what we refer to as *flatpak dependencies*. They are dependencies of the form of a flatpak, and are mounted in the sandbox during the build and sometimes also when running the final app. Those dependencies include runtimes, SDK, SDK-extensions, base-app and base-app extensions. In the case of flathub, they will come from flathub directly, but the original manifest does not include the exact OSTree commit associated to each of them. Fortunately, flatpak-builder will ship a modified version of the manifest in the final product, which include the commit of the runtime, the SDK, and (if there is any) the base-app. Unfortunately it will not include the ones for SDK-extensions and base-app extensions, but we know the flathub buildbot will use the latest version available at the time of the build. However, the exact time of the build is also not specified, it is bounded by the commit to GitHub, which is before and the OSTree commit, which happens after the build. We therefore guess which commit was used by taking the latest available at the OSTree commit time. This will introduce some error, but we will show that it is relatively small. Another dependency we need to pin to its exact version is flatpak-builder itself. In the case of the flathub buildbot, it uses the last one available on flathub. We therefore consider it as any other flatpak dependencies, and apply the same guessing technique. To manipulate the time of the build, we override the SOURCE_DATE_EPOCH variable. Under the hood, flatpak-builder overrides it with the last modification time of the manifest file, but this information is lost when a file is committed to git or OSTree. We therefore do as before, and use the OSTree commit time to override SOURCE_DATE_EPOCH, which is another source of error.

3.2 Rebuilding

Once the environment matches roughly the one from the original build, we need to execute the exact same steps done by the flathub buildbot. Since its code is open-source, we merely follow the same code, we just ignore the verification and exporting steps done at the end.

3.3 Measuring reproducibility

At the end of a rebuild we compare the checksum of the original package, with the rebuilt one, and conclude that the build is reproducible if the two are the same. This follows a very strict definition of reproducibility, and we therefore introduce three more reproducibility tests. The first is the notion of binary-reproducibility, where we compute the hash of only non human-readable files (ELF, images, archives, etc.). The second is ELF-reproducibility, where we only compare ELF files, and the last one is the repro-score, which is defined as the number of non-reproducible files over the total number of files. The reasoning behind the binary and ELF reproducibility is the following: we can still audit and understand non-binary files, and therefore if the only sources of non-reproducibility are in files we can easily verify manually, we can still attest that the build is legitimate, even if it is only partially reproducible. The idea behind the repro-score is to find a way to automatically differentiate a program which is highly non-reproducible, from those which are mostly reproducible, by assuming that a program with a high repro-score will be easy to fix, and make reproducible.

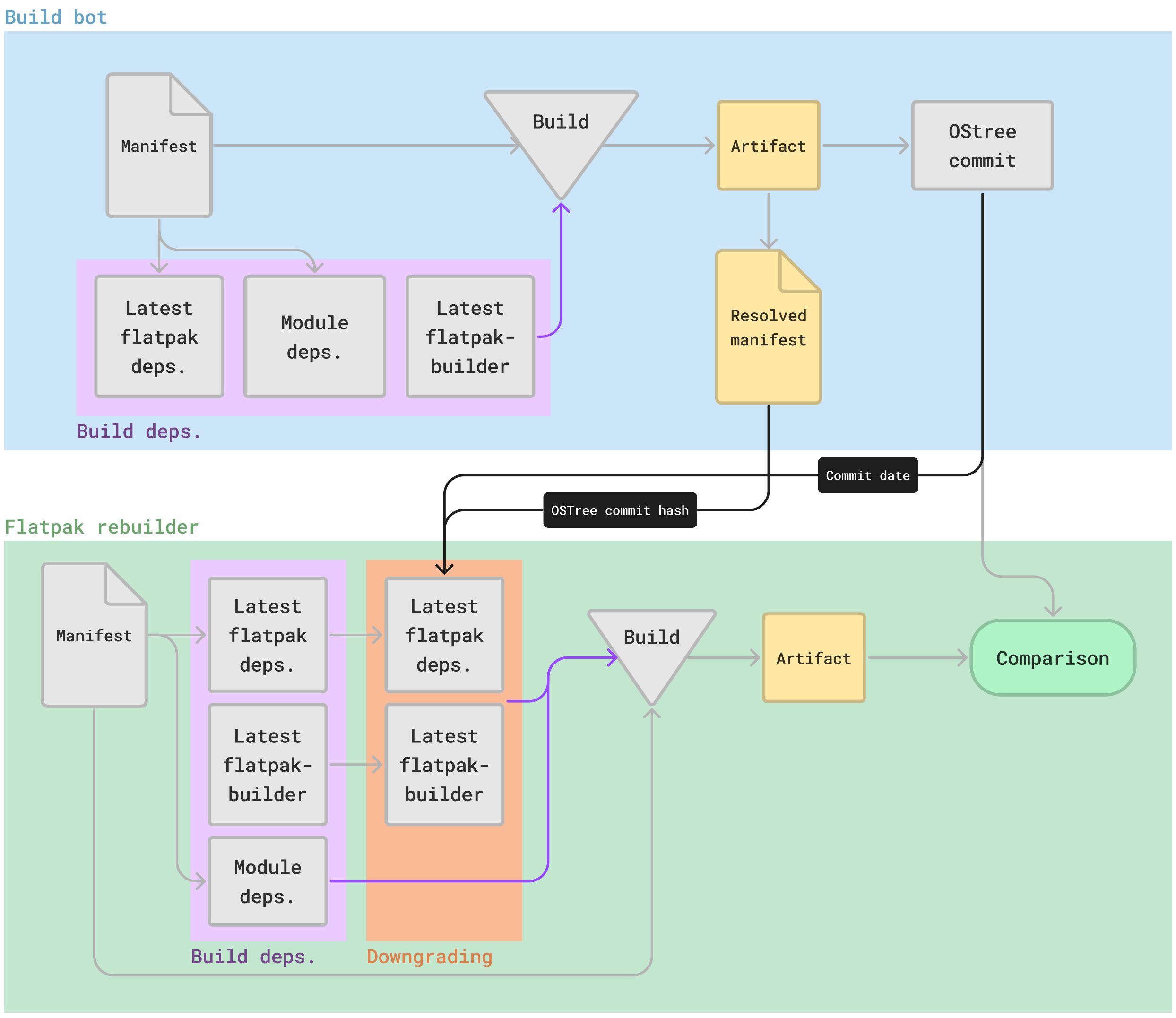


Figure 3.1: High level description of flatpak-rebuilder.

Chapter 4

Implementation

For simplicity, and portability, flatpak-rebuilder is written in python, using the poetry package manager, both available on any platform susceptible to run flatpak. Other than python, it relies on GNU coreutils, which should also be supported on most Linux distribution, and optionally uses two external tools, diffoscope and strip-nondeterminism. Both are developed as part of the R-B project. Diffoscope is used to compute the difference between two artifacts, by going deeply inside the file tree, even doing things such as diffing recursively archives. Strip-nondeterminism is a tool to strip certain non-necessary, but not deterministic, bits of information, such as archives timestamps. Also, flatpak-rebuilder relies on flatpak, and interact with it through the CLI tool, rather than using libflatpak. This way the code looks closer to a shell script, and matches more the process of building manually using flatpak-builder. Also, the flathub buildbot does the same and uses the CLI directly, making it easier to copy the build steps done by the flathub buildbot. The overall behavior of flatpak-rebuilder is the following. Before rebuilding, we first make sure the flathub remote is correctly configured, and if not, configure it. Then we gather all flathub dependencies. To do so, we should parse the package metadata, but instead we rely on flatpak-builder to do the work. Using the `--install-deps-from=flathub` flag, flatpak-builder figures out which dependencies are used, and it will print them to stdout. To have the list of flathub dependencies, we simply parse this output. As said before, flatpak-builder will take the latest dependencies available, meaning we need to downgrade them just after. To correctly downgrade flathub dependencies, we download the original flatpak, since it includes a modified version of the manifest file, with the exact commit of some of these dependencies. We also need it to compare the checksum after the rebuild. For the others, we apply the guessing technique described in chapter 3. Then, we pull the GitHub repository containing the manifest file. We have to be careful to fetch the right branch, generally master, and the right commit. Based on the OSTree commit of the package, we take the git commit which is the closest before. We also use the OSTree commit date to change the last modification time of the manifest, since flatpak-builder uses it to internally override the SOURCE_DATE_EPOCH environment variable. From there, flatpak-rebuilder is ready to rebuild. The rebuilding part occurs in three steps. First we check that the flathub dependencies were indeed downgraded to the right version. Then we only download the dependencies used by each module. This lets us compute a few statistics on the amount of dependency used, and to more accurately measure the time to build. Both are used in the analysis. Finally, we rebuild, by using the same command run by the flathub buildbot. Once done, flatpak-rebuilder compares the checksums of the original package against the freshly rebuilt one. On Arch Linux or Debian, packages are distributed in an archived format, making it straightforward to compute the checksums. Here the format is an OSTree commit. To do the comparison, we first deploy each commit, meaning we copy its content in a concrete folder and then perform the checksums on both. During all these steps we record different statistics, and serialize them at the end. These statistics include, among other things, if the build succeeded or not, what dependencies were used, what checksums were obtained. This provides an easy way to analyze the result of a rebuild and is also a nice way to communicate it. It can be signed and used as a replacement to the simple checksum used in Figure 1.1. The code is small (a bit more than a thousand line) making it easy to audit.

Chapter 5

Evaluation

In this part, we answer these two main project questions:

- PQ1: how many applications are reproducible on flathub ?
- PQ2: what are the main reasons some applications are not reproducible ?

A third minor question is to know if the repro-score introduced in section 3.3 is a good metric. We answer it in section 5.3. We also discuss some improvements that can be done to the flathub supply chain in section 5.4

We run several experiments, on three different machines, with the following specifications:

Arch desktop

- OS: Arch Linux
- Kernel: 5.18.1-arch1-1
- CPU: AMD Ryzen 7 2700X (16)
- GPU: NVIDIA GeForce GTX 760
- Memory: 16007MiB

Running version 1.12.7 of flatpak, from the Arch repository

Arch laptop

- OS: Arch Linux
- Kernel: 5.18.0-arch1-1
- CPU: Intel i5-5300U (4)
- GPU: Intel HD Graphics 5500
- Memory: 7828MiB

Running version 1.12.7 of flatpak, from the Arch repository.

Fedora desktop

- OS: Fedora Linux 35 (Workstation Edition)
- Kernel: 5.17.5-200.fc35.x86_64
- CPU: Intel i7-3770 (8)
- GPU: NVIDIA GeForce GTX 660 Ti
- Memory: 7886MiB

Running version 1.12.7 of flatpak, from the Fedora repository (rpm).

For the first experiment, we run flatpak-rebuilder on as many applications as possible, on the three machines, all using the commit `6668247ca253c0d45e387e109ce0c8d8a29f893d` of flatpak-rebuilder. As of June 10, 2022, flathub contains 1728 programs, categorized as an application (by opposition with runtimes). We only focus on applications because those are built using the flathub buildbot. In order to save time, we only rebuild 729 of them, which we selected at random. We also design a second experiment. We rebuild some non-reproducible programs, twice, on two different machines, and see if we obtain the same checksum on both. If that is the case, the program can be considered as theoretically reproducible. Because it should be reproducible but the way it is rebuilt by flatpak-rebuilder is not matching enough what is done by the flathub buildbot. The two machines used are the Arch desktop and the Fedora desktop, both using commit `18b9f49cf39d91a11fc5dbcac67098ff8c528840` of flatpak-rebuilder. Instead of rebuilding everything we reduce the number of program we consider, to save time. During the previous experiment we measured the build time of each program, and the amount of dependencies fetched from the internet. Using this information we characterize the total time to rebuild a program as such: $\frac{deps}{internetspeed} + rebuildtime$ plus some constant overhead. We can reduce the total amount of time by 2 by removing 7% of them, as shown in Figure 5.1. To further reduce it, we select 200 programs to rebuild, at random.

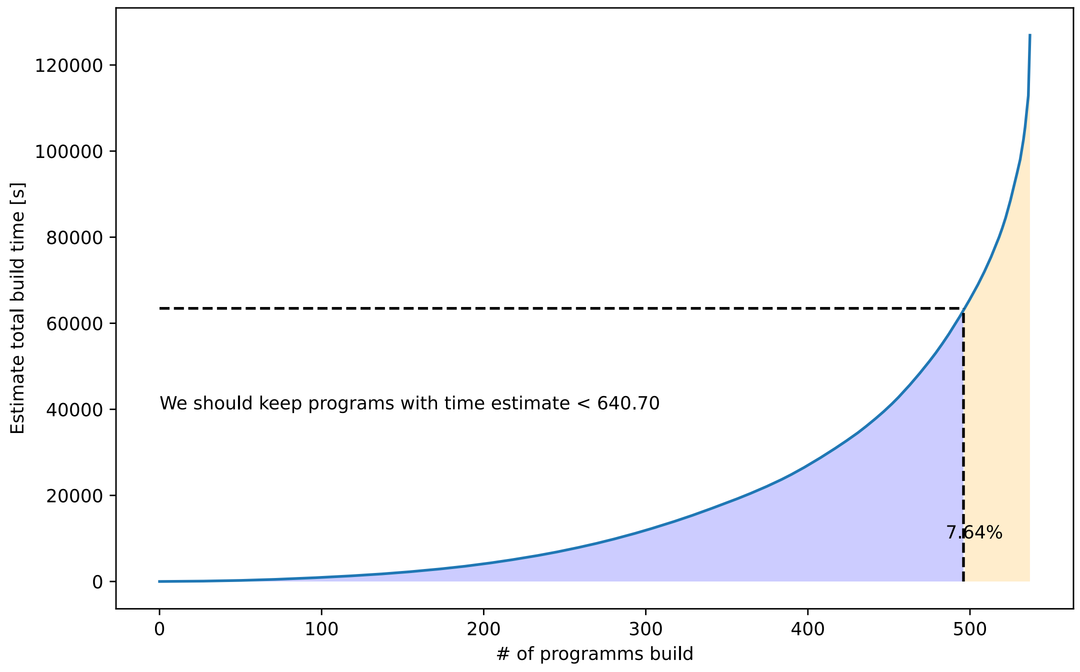


Figure 5.1: Cumulative build time, 7.64% of programs account for half of the total build time, we therefore remove them.

5.1 PQ1: Current reproducibility of flathub

From the first experiment, we obtain the following results:

	Reproducible	Non reproducible	Failed
All	306	368	55
All%	42%	50.5%	7.5%
ELF	423	251	55
ELF%	58%	34.5%	7.5%

Table 5.1: Reproducibility over 729 programs [1]

42% applications on flathub are reproducible. As a comparison, at the same time, Arch Linux has 86.4% reproducibility on their repository [1], which is much better than our results. On the other hand, we reach 58% ELF-reproducibility, meaning that in practice we can still use reproducible builds on these to provide integrity, since non ELF files can be audited manually.

[1]Two minor things need to be clarified. Here, ELF which stands for ELF-reproducibility, only compares files inside /app/bin and /app/lib, which captures the majority of ELF files but not all. This was fixed for the other experiments. Secondly, the binary-reproducibility test is not yet present at that moment.

After running the second experiment, we obtain the following results. By building twice 185 programs, 70 of them (37.6%) are in theory reproducible, and 73 are binary-reproducible. Note that 15 programs did not build on both machines, hence they were removed. The code that evaluate ELF-reproducibility was buggy at that moment. If we extrapolate from that sample we conclude that we could reach 61% reproducibility by better mimicking the original build environment.

5.2 PQ2: Causes of non reproducibility

To understand why some applications are not reproducible on flatpak we first look at one source of error which we already knew about: dependencies' version guessing. We can approximate the probability of the error introduced by guessing some build dependencies, in the following way: For a particular application, the time of the GitHub commit and the OSTree commit are known, and the build happens in between. We make a commit guessing error when an update of one dependency is published between the start of the build and the OSTree commit. If we assume that the start of the build is uniformly distributed between the GitHub commit GH_{commit} and the OSTree commit OT_{commit}, then we make an error when the start is behind the update of the dependency closest to OT_{commit}:

$$
\begin{aligned}
P_{error} &= \max_{d \in Dependencies} (P_{error\_guessing\_d}) \\
&= \max_{d} \begin{cases} \frac{OT_{commit} - d_{update}}{OT_{commit} - GH_{commit}} & \text{if } d_{update} \in [GH_{commit}; OT_{commit}] \\ 0 & \text{if } d_{update} \notin [GH_{commit}; OT_{commit}] \end{cases}
\end{aligned}
$$

The expected value of the error on all apps on flathub is 19.6 ($< 2\%$). Our guessing technique is therefore accurate and cannot explain the amount of non-reproducible programs.

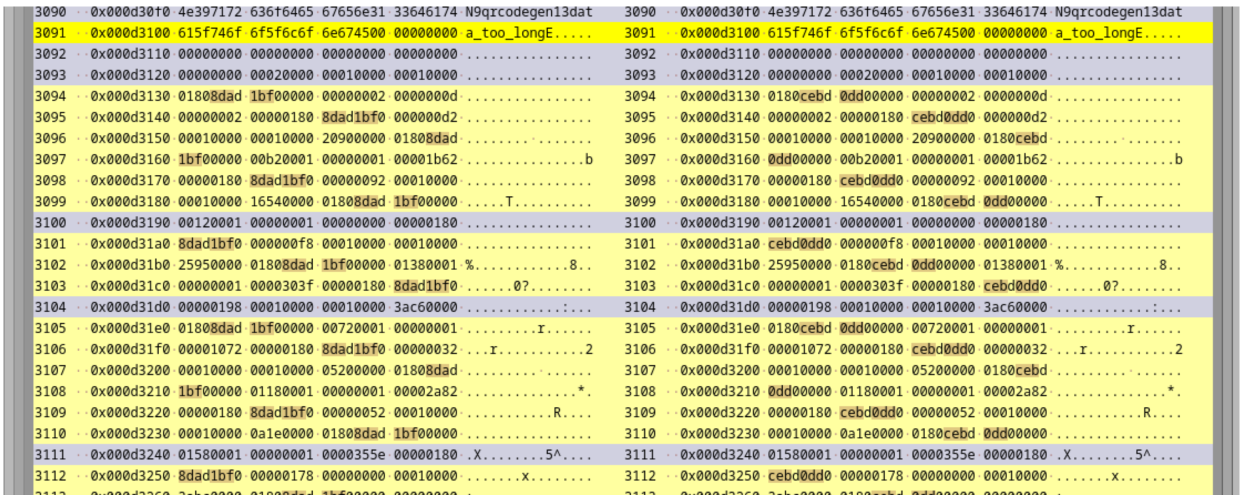
To further answer PQ2, we also manually analyze the output of diffoscope on 200 rebuilds, using commit `2685df5edcc00a8e4585fa694859223cb32db6d2` of flatpak-rebuilder. For simplicity we select the same 200 applications which we build twice for the second experiment. 9 of them did not build and we remove them from the analysis. Here are the most common issues we spot.

Timestamps 19 applications have timestamps embedded in their static strings, or in some files' metadata. They are equal to the value of SOURCE_DATE_EPOCH, meaning that it is easy to manipulate them. Furthermore, 26 programs have what we call *uncontrolled timestamps* embedded in the final package. Uncontrolled timestamps are not affected by SOURCE_DATE_EPOCH, and therefore flatpak-rebuilder cannot enforce their value.

Debug link When applicable, debug symbols of a program on flathub are available, as an extension. In order to do so, debug symbols are stripped and put in a separate *gnu_debuglink* file, and a section called .gnu-debug-link is added to the source ELF binary. This section contains the path to the debug symbol file, and a checksum of this file, this checksum is a typical source of non-reproducibility, 75 of the analyzed programs are affected.

Resolved manifest serialization The second most important source of non-reproducibility is the resolved manifest file. Indeed, while this should be serialized in exactly the same way (it is a json file), regardless of the environment, 50 programs have an extra newline in the version from the flathub buildbot. This newline is inside every empty arrays, but otherwise both contents are logically the same. To understand this we need to look at the publishing time (OSTree commit time) of the packages. We observe that every package affected by this issue has been published before January 2022. This indicates that the flathub buildbot might behave differently before, and since flatpak-rebuilder copies what is done on the last commit of the build bot, we might have a new source of error. And indeed, commit `bfcaaf2` switch to the flatpak-builder from flathub, while before it would use the one from the host, which is completely unknown [2]. This has two consequences, first this issue will be solved automatically with time, since any package updated after January 4, 2022, will use the right flatpak-builder version. Secondly, it means on the programs rebuilt in section 5.1, 182 were built using the wrong version of flatpak-builder. Furthermore, 134 were reported as not reproducible.

.ro_data The .ro_data section is a common location of non-reproducibility, first because it includes static strings that may have timestamps or path information in them, but there is another common pattern shown in Figure 5.2. Some section only differ at some bytes, with always the same pattern. This affects 37 applications.



```
3090  0x000d30f0 4e397172 636f6465 67656e31 33646174 N9qrcodegen13dat    3090  0x000d30f0 4e397172 636f6465 67656e31 33646174 N9qrcodegen13dat
3091  0x000d3100 615f746f 6f5f6c6f 6e674500 00000000 a_too_longE.....    3091  0x000d3100 615f746f 6f5f6c6f 6e674500 00000000 a_too_longE.....
3092  0x000d3110 00000000 00000000 00000000 00000000 ................    3092  0x000d3110 00000000 00000000 00000000 00000000 ................
3093  0x000d3120 00000000 00020000 00010000 00010000 ................    3093  0x000d3120 00000000 00020000 00010000 00010000 ................
3094  0x000d3130 01808dad 1bf00000 00000002 0000000d ................    3094  0x000d3130 0180cebd 0dd00000 00000002 0000000d ................
3095  0x000d3140 00000002 00000180 8dad1bf0 000000d2 ................    3095  0x000d3140 00000002 00000180 cebd0dd0 000000d2 ................
3096  0x000d3150 00010000 00010000 20900000 01808dad ................    3096  0x000d3150 00010000 00010000 20900000 0180cebd ................
3097  0x000d3160 1bf00000 00b20001 00000001 00001b62 ...............b    3097  0x000d3160 0dd00000 00b20001 00000001 00001b62 ...............b
3098  0x000d3170 00000180 8dad1bf0 00000092 00010000 ................    3098  0x000d3170 00000180 cebd0dd0 00000092 00010000 ................
3099  0x000d3180 00010000 16540000 01808dad 1bf00000 .....T..........    3099  0x000d3180 00010000 16540000 0180cebd 0dd00000 .....T..........
3100  0x000d3190 00120001 00000001 00000000 00000180 ................    3100  0x000d3190 00120001 00000001 00000000 00000180 ................
3101  0x000d31a0 8dad1bf0 000000f8 00010000 00010000 ................    3101  0x000d31a0 cebd0dd0 000000f8 00010000 00010000 ................
3102  0x000d31b0 25950000 01808dad 1bf00000 01380001 %...........8..    3102  0x000d31b0 25950000 0180cebd 0dd00000 01380001 %...........8..
3103  0x000d31c0 00000001 0000303f 00000180 8dad1bf0 ......0?........    3103  0x000d31c0 00000001 0000303f 00000180 cebd0dd0 ......0?........
3104  0x000d31d0 00000198 00010000 00010000 3ac60000 ............:...    3104  0x000d31d0 00000198 00010000 00010000 3ac60000 ............:...
3105  0x000d31e0 01808dad 1bf00000 00720001 00000001 .........r......    3105  0x000d31e0 0180cebd 0dd00000 00720001 00000001 .........r......
3106  0x000d31f0 00001072 00000180 8dad1bf0 00000032 ...r...........2    3106  0x000d31f0 00001072 00000180 cebd0dd0 00000032 ...r...........2
3107  0x000d3200 00010000 00010000 05200000 01808dad ................    3107  0x000d3200 00010000 00010000 05200000 0180cebd ................
3108  0x000d3210 1bf00000 01180001 00000001 00002a82 ..............*.    3108  0x000d3210 0dd00000 01180001 00000001 00002a82 ..............*.
3109  0x000d3220 00000180 8dad1bf0 00000052 00010000 ...........R....    3109  0x000d3220 00000180 cebd0dd0 00000052 00010000 ...........R....
3110  0x000d3230 00010000 0a1e0000 01808dad 1bf00000 ................    3110  0x000d3230 00010000 0a1e0000 0180cebd 0dd00000 ................
3111  0x000d3240 01580001 00000001 0000355e 00000180 .X........5^....    3111  0x000d3240 01580001 00000001 0000355e 00000180 .X........5^....
3112  0x000d3250 8dad1bf0 00000178 00000000 00010000 .......x........    3112  0x000d3250 cebd0dd0 00000178 00000000 00010000 .......x........
```

Figure 5.2: Example of diffoscope output on a program which differ in the .ro_data section, with the same pattern, the same 28bits separate by 148bits each time

Python files Python's byte code files (`.pyc`) are not reproducible, on 29 programs. Three common patterns appear. First, certain serialization are not deterministic, like with frozenset, as mentioned already in chapter 2. Some of these bit of non-determinism are due to the way python compute hashes, based on a random seed. This seed can be fixed by overriding the PYTHONHASHSEED environment variable, and can be done on a per-program basis, by overriding it in the manifest file. Another source of problems is how certain temporary path used by the pip package manager, are embedded in the final `.pyc` file. Those paths have random names, and points to files which are not even present in the final flatpak, since they are at /tmp. Those are therefore useless but not deterministic bit of information. Lastly, a commonly used python library, called sysconfig, embed a description of the active environment variable at the time of the build. This should not be a problem, they should always be the same since those are controlled inside the sandbox. However, when running a flatpak, every extension related is automatically mounted inside the sandbox. In particular, every SDK-extensions or runtime-extensions available on the host system (not only those specified in the manifest) are mounted and are added to the LD_LIBRARY_PATH environment variable. This creates yet another source of uncontrolled build inputs.

Path A general source of non-reproducibility is the path in which the build take place. This is a minor issue in the case of flatpak, since the build take place in the sandbox, where most paths are the same, regardless of the environment. However, for consistency reasons, the hostname is replicated inside the sandbox, and a /home/<username> directory is created. This can affect certain applications, in our case only 8.

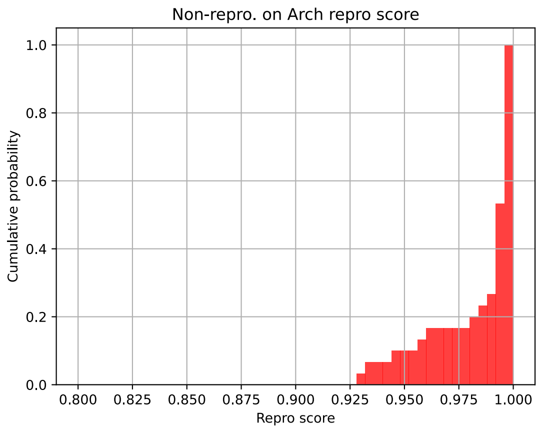
Hardware dependant compilation While, for portability, flatpak cannot use extended instruction set, they can still be optimized to better run on specific processors, with the usage of the `-mtune` compilation flag for instance. This breaks reproducible builds if the program is built with another processor model, and it does appear on certain flatpak. The identification process is a bit more complicated, without looking at the build script directly. Only 3 programs are for sure compiled using mtune, because it appears in some of their metadata. In particular, the issue come from a common library, called ImageMagick. A second issue is that the flathub buildbot runs on an Intel broadwell CPU, like the Arch Laptop used in some experiments. This means that a program compiled with mtune and reproduce on the laptop, will not be detected as not reproducible, even though it should.

Appdata The last common issue resides in the appdata file. This file provides metadata to correctly present the program in an app store or through a package manager. The whole specification is not important, what is relevant for reproducible builds are the way screenshots' links are handled. Since they point to servers which are not maintained by flathub, the flathub buildbot add the `--mirror-screenshots-from=flathub` flag to flatpak-builder, to mirror the screenshots on flathub directly. This rewrite the appdata file, to make links point to the flathub servers.

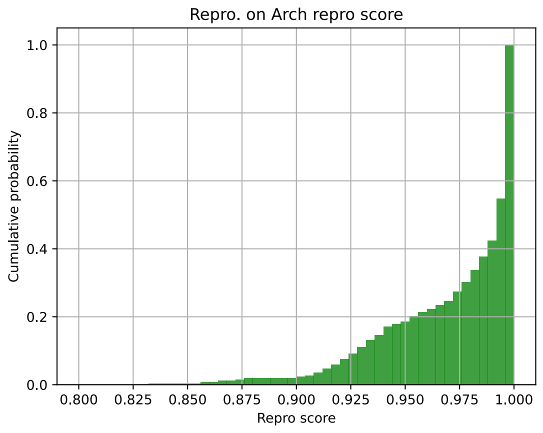
The new links are the concatenation of the package's name and the md5sum of the screenshot. Additionally, flatpak-builder also includes different sizes for each screenshot. This mirroring is useful in case one of the original link gets down, but it also causes troubles to have proper reproducible builds. That is because, the original link might point to nothing anymore when we rebuild, in which case flatpak-builder will just remove the links from the appdata file. Or the link might point to a newer version of the screenshot, in which case flatpak-builder mirror another screenshots, and the links will be different, since they are based on the md5sum of the file. This overall result in an additional form of build dependency which cannot be controlled. It affects 14 applications.

Finally To better understand how time influences a build, we add an option to manipulate the time of the build used by flatpak-rebuilder, and we observe the following things. Obviously timestamps which match the SOURCE_DATE_EPOCH variable change but so do the `.ro_data` sections and the `.gnu_debuglink`.

5.3 Repro score analysis



(a) Non-reproducible packages on Arch Linux.



(b) Reproducible packages on Arch Linux.

Figure 5.3: Comparison of the cumulative distribution of the repro-score, between packages reproducible on Arch Linux, and those which are not.

To understand if the repro-score is a good metric or not, we do the following experiment. We find all packages available on both flathub and Arch Linux. From there we rebuild them and compute their repro-score. We then compare the repro-score distribution, for packages which are reproducible on Arch, and for those which are not, shown in Figure 5.3. If the difference between the two is significant, we conclude that the repro-score is a good metric to estimate if a non-reproducible program can easily be made reproducible. The reasoning is as follows: reproducible builds on Arch Linux are developed since a few years back, and programs which are not reproducible there can be considered as hard to make reproducible. Unfortunately in Figure 5.3, both probability distributions are similar. Even worse, the one for reproducible packages has a bigger tail towards low repro-scores, which is the opposite of what is expected. This makes sense in reality. We already showed that a simple modification to the SOURCE_DATE_EPOCH variable can lead to a lot of modification in the final files, even though the original modification is small.

5.4 Improvements

Considering the different main reasons flatpaks are not reproducible, we come up with several fixes. The first one is to include the value of SOURCE_DATE_EPOCH inside the resolved manifest. On top, we include an easier way to override its value when calling flatpak-builder [2]. The second main issue, manifest serialization, will get solved by itself, since any program that gets an update now, will not be affected anymore. To improve a bit more, we also optionally post process the artifact through strip-nondeterminism. This should counter errors such as timestamps in archives or images. Overall this would affect 129 programs, and make 67 of them completely reproducible, on the 191 manually analyzed program from section 5.2. The theoretical results are shown in Table 5.2.

Fix	Affected	Completely fixed
Time	87	31
Manifest serialization	50	29
Striping	7	3
Overall	129%	67
Overall%	67.54%	35%

Table 5.2: Theoretical benefit of various fixes.

[2]The PR is available here: `https://github.com/flatpak/flatpak-builder/pull/470`

Chapter 6

Related Work

Other work is closely related to this project. First tools similar to flatpak-rebuilder but for other package distributions exist, and they achieve much better performance. Also, projects exist to simplify the analysis step and automatically fix non-reproducible packages.

6.1 Other rebuilding tools

Tools exist to replicate builds for other package distributions, two important one being Debian with their tool debrebuild, and Arch Linux with arch-repro. Both achieve much better performances than flatpak-rebuilder [1, 4], but are also around since much longer, 2014 for deb-rebuild and 2017 for arch-repro. Nevertheless, flatpak-rebuilder provides two advantages, the first one is that it works across distribution, by mostly relying on flatpak (which underneath uses bubblewrap) and not other container technologies. This make it easier to have independent rebuilders. The second advantage is the introduction of the two more relaxed notion of reproducibility, which allows us to have trust in packages which are not entirely reproducible, by manually inspecting the package difference inside human-readable files. Flatpak-rebuilder can also be run rootless, which again, can increase the number of potential independent rebuilders.

6.2 Automatic analysis

In this project, the analysis of the causes of non-reproducibility of flatpak was done manually, by reading both the diffoscope output and the manifest files. This task is tedious and work was done to automate it. For instance, Zhilei et al. [8] propose to use system call tracing to figure out which files and which commands are responsible for unreproducible builds. More recently, they also present a new approach, RepFix, to automatically patch these problematic files [7].

6.3 Trusting trust

While it improves the security of the supply chain, reproducible builds cannot address all issues, like the "Trusting Trust" problem. This is the colloquial name given to a potential attack first described by Ken Thompson 1984 [10]. Here the compiler is the compromised part of the supply chain, and it injects malicious code in the final product. This is a sophisticate attack, but the XcodeGhost attack from 2015 is a concrete example, where a modified version of Apple's Xcode was shipped and used by many developers [14]. Our reproducible builds setup is susceptible to such attack, because we always compile using pre-built compilers, provided by the SDK in the case of flatpak. One way to counter this would be to use reproducible builds on compilers themselves, but since they are compiled using another compiler, for instance a previous version, the process of rebuilding needs to be applied recursively. Therefore, other projects, like *Bootstrappable Builds*, attempts to provide a clear and simple path from source code to binary-blobs like compiler, that can be itself verified using reproducible builds.

Chapter 7

Conclusion

In this project, we investigate the feasibility of using reproducible builds with flatpak. We develop flatpak-rebuilder to automate the process and measure only 42% reproducibility, which is lower than what found on other package distributions. After a manual analysis, we provide several changes to flatpak-builder to improve this number, in theory to 60%. For future work, we should develop a way to share the result of flatpak-rebuilder, and provide the ability to flatpak end-users to easily query independent rebuilders, before trusting a package. Also, we are interested in using RepFix to automatically analyze and patch non-reproducible flatpaks.

Bibliography

[1] *Archlinux rebuilderd*. 2022. URL: `https://reproducible.archlinux.org/` (visited on 05/29/2022).

[2] barthalion. *Use org.flatpak.Builder instead of host packages.* [Online; accessed 04-June-2022]. URL: `https://reproducible-builds.org/docs/source-date-epoch/`.

[3] Debian contributors. *deb-buildinfo - Debian build information file format.* [Online; accessed 31-May-2022]. 2022. URL: `https://manpages.debian.org/bullseye/dpkg-dev/deb-buildinfo.5.en.html`.

[4] Debian contributors. *Overview of various statistics about reproducible builds.* [Online; accessed 31-May-2022]. 2022.

[5] Chris Lamb and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains". In: *CoRR* abs/2104.06020 (2021). arXiv: `2104.06020`. URL: `https://arxiv.org/abs/2104.06020`.

[6] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: (2020). Ed. by Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, pp. 23–43.

[7] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. "Automated Localization for Unreproducible Builds". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 71–81. DOI: `10.1145/3180155.3180224`.

[8] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. "Automated Localization for Unreproducible Builds". In: *CoRR* abs/1803.06766 (2018). arXiv: `1803.06766`. URL: `http://arxiv.org/abs/1803.06766`.

[9] Reproducible Builds contributors. *SOURCE_DATE_EPOCH.* [Online; accessed 04-June-2022]. URL: `https://reproducible-builds.org/docs/source-date-epoch/`.

[10] Ken Thompson. "Reflections on trusting trust". In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763. DOI: `10.1145/358198.358210`.

[11] vstinner. *Reproducible pyc: frozenset is not serialized in a deterministic order.* [Online; accessed 01-June-2022]. URL: `https://github.com/python/cpython/issues/81777`.

[12] *When is a build reproducible?* [Online; accessed 05-June-2022]. URL: `https://reproducible-builds.org/docs/definition/`.

[13] Wikipedia contributors. *2020 United States federal government data breach — Wikipedia, The Free Encyclopedia.* [Online; accessed 30-May-2022]. 2022.

[14] Wikipedia contributors. *XcodeGhost — Wikipedia, The Free Encyclopedia.* `https://en.wikipedia.org/w/index.php?title=XcodeGhost&oldid=1054394297`. [Online; accessed 7-June-2022]. 2021. URL: `https://en.wikipedia.org/w/index.php?title=XcodeGhost&oldid=1054394297`.

[15] Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, Zhen Ming, et al. "Towards Build Verifiability for Java-based Systems". In: *arXiv preprint arXiv:2202.05906* (2022).