



**HAL**  
open science

# Does Functional Package Management Enable Reproducible Builds at Scale? Yes

Julien Malka, Stefano Zacchiroli, Théo Zimmermann

► **To cite this version:**

Julien Malka, Stefano Zacchiroli, Théo Zimmermann. Does Functional Package Management Enable Reproducible Builds at Scale? Yes. 22nd International Conference on Mining Software Repositories, Apr 2025, Ottawa, France. hal-04913007v2

**HAL Id: hal-04913007**

**<https://hal.science/hal-04913007v2>**

Submitted on 4 Feb 2025

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Does Functional Package Management Enable Reproducible Builds at Scale? Yes.

Julien Malka  
LTCI, Télécom Paris  
Institut Polytechnique de Paris  
Palaiseau, France  
julien.malka@telecom-paris.fr

Stefano Zacchiroli  
LTCI, Télécom Paris  
Institut Polytechnique de Paris  
Palaiseau, France  
stefano.zacchiroli@telecom-paris.fr

Théo Zimmermann  
LTCI, Télécom Paris  
Institut Polytechnique de Paris  
Palaiseau, France  
theo.zimmermann@telecom-paris.fr

**Abstract**—Reproducible Builds (R-B) guarantee that rebuilding a software package from source leads to bitwise identical artifacts. R-B is a promising approach to increase the integrity of the software supply chain, when installing open source software built by third parties. Unfortunately, despite success stories like high build reproducibility levels in Debian packages, uncertainty remains among field experts on the scalability of R-B to very large package repositories.

In this work, we perform the first large-scale study of bitwise reproducibility, in the context of the Nix functional package manager, rebuilding 709 816 packages from historical snapshots of the nixpkgs repository, the largest cross-ecosystem open source software distribution, sampled in the period 2017–2023.

We obtain very high bitwise reproducibility rates, between 69 and 91% with an upward trend, and even higher rebuildability rates, over 99%. We investigate unreproducibility causes, showing that about 15% of failures are due to embedded build dates. We release a novel dataset with all build statuses, logs, as well as full “diffscopes”: recursive diffs of where unreproducible build artifacts differ.

**Index Terms**—reproducible builds, functional package management, software supply chain, reproducibility, security

## I. INTRODUCTION

Free and open source software (FOSS) is a great asset to build trust in a computing system, because one can audit the source code of installed components to determine if their security is up to one’s standards. However trusting the source code of the components making up a system is not enough to trust the system itself: before a program can be run on a user machine, it is typically *built*<sup>1</sup> to obtain an executable artifact, and then *distributed* onto the target system, involving a set of processes and actors generally referred to as the *software supply chain*. In recent years, large scale attacks like *Solarwinds* [1] or the *xz* backdoor [25] have specifically targeted the software supply chain, underlying the importance of measures to increase its security and also triggering policy response in the European Union and the United States of America [9, 15]. The particular effectiveness of these attacks is due to the difficulty to analyze binary artifacts in order to understand how they might act on the system, hence increasing the need for tooling that provide traceability from executable binaries to their source code.

<sup>1</sup>A term that we will also use in this work for interpreted programs, where it is the runtime environment that has to be built.

*Reproducible builds (R-B)*—the property of being able to obtain the same, bitwise identical, artifacts from two independent builds of a software component—is recognized as a promising way to increase trust in the distribution phase of binary artifacts [16]. Indeed, if a software is bitwise reproducible, a user may require several independent parties to reach a consensus on the result of a compilation before downloading the built artifacts from one of them, effectively distributing the trust in these artifacts between those parties. For an attacker wanting to compromise the supply chain of that component, it is no longer sufficient to compromise only one of the involved parties. Build reproducibility is however not easy to obtain in general, due to non-determinism in the build processes, documented both by practitioners and researchers [16, 2]. The *Reproducible Builds* [32] project has since 2015 worked to increase bitwise reproducibility throughout the FOSS ecosystems, by coming up with fixes for compilers and other toolchain components, working closely with upstream projects to integrate them.

Unfortunately, a recent study [14] which interviewed 24 R-B experts still concluded that there is “*a perceived impracticality of fully reproducible builds due to workload, missing organizational buy-in, unhelpful communication with upstream projects, or the goal being perceived as only theoretically achievable*” and that “*much of the industry believes [R-B] is out of reach*”. While there exist some successful examples of package sets with high reproducibility levels like Debian, which consistently achieves a reproducibility rate of more than 95% [26], those good performances should be put in perspective with the strict quality policies applied in Debian and the relatively limited size of the package set. Uncertainty remains among field experts about the scalability of this approach to larger software distributions.

*Nixpkgs* is the largest cross-ecosystem FOSS distribution, totaling as of October 2024 about 100 000 packages.<sup>2</sup> It includes components from a large variety of software ecosystems, making it an interesting target to study bitwise reproducibility at scale. *Nixpkgs* is built upon *Nix*, the seminal implementation of the *functional package management (FPM) model* [12]. It is generally believed that the FPM model is ef-

<sup>2</sup>Based on the Repology rankings <https://repology.org>, accessed Oct. 2024.

fective to obtain R-B: FPM packages are *pure functions* (in the mathematical sense) from build- and run-time dependencies to build artifacts, described as “build recipe”s that can be executed locally by the package manager. Components are built in a sandboxed environment, disallowing access to unspecified dependencies, even if they are present on the system. Previous work has highlighted that this model allows to reproduce build environments both in space and time [21], a necessary property for build reproducibility. Additionally, nixpkgs’ pre-defined build processes implement best practices to ensure build reproducibility, like setting the `SOURCE_DATE_EPOCH` environment variable [33] or automatically verifying that the build path does not appear in the built artifacts [3]. Despite the potential for insightful distribution-wide build reproducibility metrics, nixpkgs limits its monitoring to the narrow set of packages included in the minimal and gnome-based ISO images [24], where a reproducibility rate higher than 95% is consistently reported.

*Contributions:* In this work, we perform the first ever large scale empirical study of bitwise reproducibility of FOSS going back in time, rebuilding historical packages from evenly spaced snapshots of the nixpkgs package repository taken every 4.1 months from 2017 to 2023. With this experiment, we answer the following research questions:

- **RQ1: What is the evolution of bitwise reproducible packages in nixpkgs between 2017 and 2023?** How does the reproducibility rate evolve over time? Are unreproducible packages eventually fixed? Do reproducible packages remain reproducible?
- **RQ2: What are the unreproducible packages?** Are they concentrated in specific ecosystems? Are critical packages more likely to be reproducible?
- **RQ3: Why are packages unreproducible?** Is large-scale identification of common causes possible?
- **RQ4: How are unreproducibilities fixed?** Are they fixed by specific patches or as part of larger package updates? Are the fixes intentional or accidental?

Besides, we use our experiment to replicate and extend previous results [21], leading to an additional research question:

- **RQ0: Does Nix allow rebuilding past packages reliably (even if not bitwise reproducibly)?**

*Results:* Thanks to this large-scale experiment, we are able to establish for the first time that **bitwise reproducibility is achievable at scale**, with reproducibility rates ranging from 69% to 91% over the period 2017–2023, despite a continuous increase in the number of packages in nixpkgs. We highlight the wide variability in reproducibility rates across ecosystems packaged in nixpkgs, and show the significant impact that some core packages can have on the overall reproducibility rate of an ecosystem.

We estimate the prevalence of some common causes of non-reproducibility at a large scale for the first time, showing that about **15% of failures are due to embedded build dates**.

As part of this work, we introduce a **novel dataset containing build logs and metadata** of over 709 000 package builds,

and more than 114 000 occurrences of non-reproducibility with full artifacts including “diffoscopes”, i.e., recursive diffs of where unreproducible build artifacts differ. Ample room for further research is left open by the dataset, including exploiting the build logs, or applying more complex heuristics or qualitative research to the diffoscopes.

*Paper structure:* Section II presents the related work. Section III gives some background that is required to understand the experiment, whose methodology is then presented in Section IV. Some descriptive statistics about the dataset are presented in Section V, and the results to our RQs in Section VI. We discuss them in Section VII, and the threats to validity in Section VIII, concluding in Section IX.

## II. RELATED WORK

### A. Reproducible builds (R-B)

R-B are a relatively recent concept, which has been picked up and developed mostly by practitioners from Linux distributions and upstream maintainers. The *Reproducible Builds* project [32] has been the main actor in the area. The project has produced a definition of R-B, best practices to achieve them, and tools to monitor the reproducibility of software distributions, and debug unreproducibilities (the *diffoscope*).

Besides, R-B have picked the interest of the academic community, with a growing number of papers on the topic.

a) *R-B for the security of the software supply chain:* R-B are often seen as a way to increase the security of the software supply chain [16]. Torres-Arias *et al.* provide a framework to enforce the integrity of the software supply chain for which they demonstrate an application to enforce R-B [34]. Our paper does not contribute directly to this line of research but, by demonstrating the feasibility of R-B at scale, it strengthens the case of the approach.

b) *Techniques for build reproducibility:* In a series of articles [29, 30, 31], Ren *et al.* devised a methodology to automate the localization of sources of non-reproducibility in build processes and to automatically fix them, using a database of common patches that are then automatically adapted and applied.

An alternative technique to achieve build reproducibility is proposed by Navarro *et al.* [23]. They propose “reproducible containers” that are built in a way that makes the build process fully deterministic, at the expense of performance.

FPMs such as Nix [12] and Guix [7] are also presented as a way to achieve R-B. Malka *et al.* [21] showed that Nix allows reproducing past build environment reliably, as well as rebuilding old packages with high confidence, but they do not address the question of bitwise reproducibility, which we do with this work.

c) *Relaxing the bitwise reproducibility criterion:* Because of the difficulty (real or perceived) to achieve bitwise reproducibility, some authors have proposed to relax the criterion to a more practical one. For instance, *accountable builds* [27] aim to distinguish between differences that can be explained (accountable differences) or not (unaccountable differences). Our work highlights that bitwise reproducibility

is achievable at scale in practice, and thus that relaxing the reproducibility criterion may not be necessary after all.

d) *Empirical studies of R-B*: Some other recent academic works have empirically studied R-B in the wild. Two papers from 2023 [5, 14] looked into business adoption of R-B and perceived effectiveness through interviews.

Bajaj *et al.* [2] mined historical results from R-B tracking in Debian to investigate causes, fix time, and other properties of unreproducibility in the distribution. Our work is similar, but instead of relying on historical R-B tracking, we actually rebuild packages and compare them bitwise to historical build results. When we report on packages being reproducible, it means they have stood the test of time. It also allows us to provide more detailed information on the causes of unreproducibility, in particular by generating diffoscopes and saving them for future research as part of our dataset; whereas diffoscopes from the Debian R-B tracking are not preserved in the long-term. Finally, Bajaj *et al.* used issue tracker data from the R-B project to identify the most common causes of non-reproducibility, possibly introducing a sampling bias since only root causes that were identified by Debian developers are counted in their statistics. In our work, we try to avoid this bias by performing a large-scale automatic analysis of diffoscopes to automatically identify the prevalence of a selection of causes of non-reproducibility. While we present heuristics comparable to some of the causes identified in Bajaj *et al.*'s taxonomy, we derive them from empirical data rather than relying on pre-labeled data from the Debian issue tracker.

The only other work that performed an experimental study of R-B investigated the impact of configuration options [28]. Contrary to them, we rebuild historical versions of packages in their default configuration. Combining the historical snapshot approach of our work with their approach of varying the configuration options could be an interesting future work.

### B. Linux distributions and package ecosystems

Besides R-B, our work also relates to the literature on Linux distributions and package ecosystems. The nixpkgs repository being the largest cross-ecosystem software distribution, we are able to compare properties of packages across ecosystems. Several previous works have compared package ecosystems (e.g., [10]). For an overview of recent research on package ecosystems, see Mens and Decan [22].

More specifically, nixpkgs is the basis of the NixOS Linux distribution. Linux distributions have a long history of being studied by the research community. Recently, Legay *et al.* [17] measured the package freshness in Linux distributions. While this is not the topic of this work, our dataset could be used, e.g., to study how frequently packages are updated in nixpkgs.

## III. BACKGROUND

We provide in this section some background knowledge about Nix and R-B, which is required to understand the details of our experiments.

```
{stdenv, fetchFromGitHub, ncurses, autoreconfHook}:

stdenv.mkDerivation rec {
  pname = "htop";
  version = "3.2.1";

  src = fetchFromGitHub {
    owner = "htop-dev";
    repo = "htop";
    rev = version;
    sha256 = "sha256-MwtsvdPHcUdegsYj9NGyded5XJQxXrilIM1j4gef1Xk=";
  };

  nativeBuildInputs = [ autoreconfHook ];
  buildInputs = [ ncurses ];
};
```

Fig. 1. Example Nix expression for the htop package.

### A. The FPM model and the Nix store

The FPM model applies the idea of functional programming to package management. Nix packages are viewed as pure functions from their inputs (source code, dependencies, build scripts) to their outputs (binaries, documentation, etc.). Any change to the inputs should produce a different package version. Nix allows multiple versions of the same package to be built and coexist on the same system. To that end, Nix stores build outputs in input-addressed directories (using a hashing function of the inputs) in the *Nix store*, usually located in the `/nix/store` directory on disk. Figure 1 shows an example of a Nix packaging expression (a *Nix recipe*) for the htop package.

### B. Nix evaluation-build pipeline

Building binary outputs from a Nix package recipe is a two-step process. First, Nix *evaluates* the expression and transforms it into a *derivation*, an intermediary representation in the Nix store containing all the necessary information to run the build process. In particular, the derivation contains ahead of time the (input-addressed) *output path*, that is the exact location in the Nix store where the build artifacts will be stored if that derivation were to be built.

Then, given the derivation file as input, the `nix-build` command performs the build, creating the pre-computed output path in the Nix store upon completion.

In the same fashion as other Linux distributions, Nix packages may produce multiple outputs (a main output with binaries, one with documentation, etc.). Each output has its own directory in the Nix store, and building the derivation from source systematically produces all its outputs.

### C. Path substitution and binary caches

Alternatively to building from source, Nix offers the option to download prebuilt artifacts from third party *binary caches*, which are databases populated with build outputs generated by Nix. Binary caches are indexed by output paths, making it possible for Nix to check for the presence of a precompiled package in a configured cache after the evaluation phase. <https://cache.nixos.org> is the official cache for the Nix community and most Nix installations come configured to use it as a trusted cache.

#### D. The nixpkgs continuous integration

Hydra [13] is the continuous integration (CI) platform for the nixpkgs project. At regular intervals in time, it fetches the latest version of nixpkgs’ git master branch and evaluates the `pkgs/top-level/release.nix` file embedded in the repository. This evaluation yields a list of derivations (or *jobs*) that are then built by Hydra: one derivation for each of the  $\approx 100\,000$  packages contained in nixpkgs nowadays. Upon success of a predefined subset of these jobs, the revision is deemed valid and all the built artifacts are uploaded to the official binary cache to be available to end users.

#### E. Testing bitwise reproducibility with Nix

Nix embarks some minimal tooling to test the reproducibility of a given derivation in the form of a `--check` flag passed to the `nix-build` command. To check for bitwise reproducibility, Nix needs a reference that it will try to acquire from one of the configured caches, or fail if not possible. Nix then acquires the build environment of the derivation under consideration, builds the derivation, and compares each of the outputs of the derivation against the local version. The `--keep-failed` flag can be used to instruct Nix to keep the unreproducible outputs locally for further processing.<sup>3</sup>

#### F. Diffoscope

Diffoscope [11] is a tool developed and maintained by the Reproducible Builds project that aims to simplify the analysis of differences between software artifacts. It is able to recursively unpack binary archives and automatically use ecosystem specific diffing tools to allow for better understanding of what makes two software artifacts different. It generates HTML or JSON artifacts—also called *diffscopes*—that can be either interpreted by humans or automatically processed.

### IV. METHODOLOGY

Our build and analysis pipeline is summarized in Figure 2.

#### A. Reproducibility experiments

1) *Revision sampling*: We start from the 200 nixpkgs revisions selected by Malka *et al.* [21] in the period July 2017–April 2023. Since *building* revisions, as opposed to just evaluating them, is very computationally intensive, it was not feasible to build all 200 revisions. Also, it was difficult to correctly estimate how many revisions we could build, due to the ever-growing number of packages in each revision. We hence applied dichotomic sampling: we first build the most recent revision, then the oldest one, then the one in the middle of them, and so on always picking the one in the middle of the largest time interval when choosing. After 17 revisions built, we obtain a regularly spaced sample set of nixpkgs revisions, with one sampled revision every 4.1 months. On average, each revision corresponds to building more than 41 thousand packages (see Figure 3 for details, discussed later).

<sup>3</sup>For our experiment, we alter the behavior of `nix-build --check` to prevent it from failing early as soon as one unreproducible output is detected.

To perform our builds, we used a distributed infrastructure based on Buildbot [4], a Python CI pipeline framework. Our infrastructure has two types of machines: a *coordinator* and multiple *builders*. The coordinator is in charge of distributing the workload and storing data that must be persisted, while the builders are stateless and perform workloads sent by the coordinator. During the course of the experiment (from June to October 2024) the set of builders we used was composed of shared bare-metal machines running various versions of Ubuntu and Fedora and our coordinator was a virtual machine running Ubuntu. Note that to perform our bitwise reproducibility checks, we compare to the historical results coming from Hydra, that uses builders that, at the time, ran older versions of Nix than the ones we used on our builders.<sup>4</sup>

2) *Evaluation and preprocessing*: For each revision considered, the coordinator first evaluates `pkgs/top-level/release.nix` (containing the list of jobs built by Hydra for this revision) using `nix-eval-jobs`, a standalone and parallel Nix evaluator similar to the one used on Hydra. The outcome of this operation is a list of derivations. The `release.nix` file is human crafted and does not contain all the dependencies of the listed packages, even though they are built by Hydra along the way. Since we are interested in testing the reproducibility of the entire package graph built by Hydra, we post-process the list of jobs obtained after the evaluation phase to include all intermediary derivations by walking through the dependency graph of each derivation. During this post-processing, we also check that the derivation outputs are present in the official Nix binary cache. This is required to compare our build outputs with historical results for bitwise reproducibility. Derivations missing from the cache can indicate that they historically failed to build, although there can be other reasons for their absence.

3) *Building*: To build a job *A* and test the reproducibility of its build outputs, the builder uses the `nix-build --check` command, as described in Section III-E. This means that we always assume that *A*’s build environment is buildable and always fetch it from the cache. This allows all the derivations in the sample set to be built and checked independently and in parallel, irrespective of where they are located in the package dependency graph. Note that the source code of packages to build is part of the build environment. Relying on the NixOS cache hence avoids incurring into issues such as source code disappearing from the original upstream distribution place. Investigating how much of NixOS can be rebuilt without relying on the cache is an interesting research question, recently explored for Guix [8], but out of scope for this paper.

After each build, we classify the derivation as either *building reproducibly*, *building but not reproducibly* or *not building*. We save the build metadata and the logs, and when available we download and store the historical build logs from the nixpkgs binary cache. Finally, for every unreproducible output path, we

<sup>4</sup>Further details on the operating systems, Nix versions and kernel versions that we used on builders can be found in the replication package.

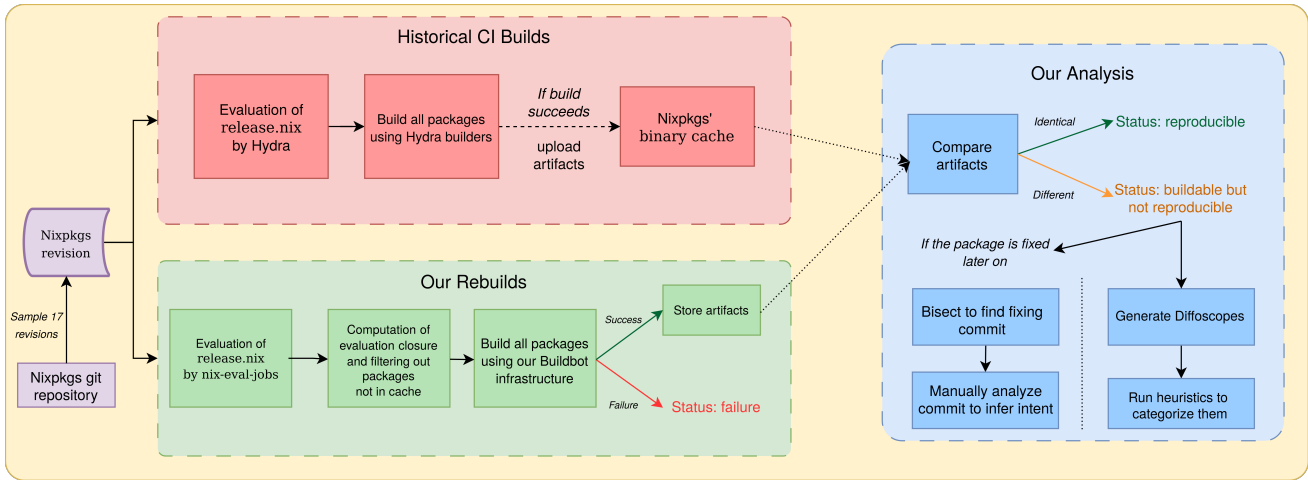


Fig. 2. Description of our build and analysis pipeline.

store both the historical artifacts and our locally built ones, for comparison purposes.

### B. Ecosystem identification and package tracking

To answer RQ1, RQ2 and RQ4, we need to be able to discriminate packages by provenance ecosystem, and track them over time to follow their evolution. To categorize packages by ecosystem, we rely on the first component of the package name when it has several components (for example a package named `haskellPackages.network` is sorted into the Haskell ecosystem). Sometimes, there are several co-existing versions of an ecosystem in a given nixpkgs revision (for example `python37Packages` and `python38Packages` being present in the same revision), and sometimes the name of the ecosystem is modified between successive nixpkgs revisions. Therefore, some deduplication step is necessary. The first and last authors performed this step manually by inspecting the 144 ecosystems from the 17 nixpkgs revisions considered, ordered alphabetically, and deciding which ones to merge independently, then checking the consistency of their results, discussing the few differences (missed merges, or false positives) and reaching a consensus. For instance, the following ecosystems were merged into a single one: `php56Packages`, `php70Packages`, ..., `php82Packages`, `phpExtensions`, `phpPackages` and `phpPackages-unit`.

To deduplicate packages appearing in several versions of the same ecosystem, we order by version (favoring the most recent one) and consider any package set without a version number as having a higher priority (since it is the default one in the considered revision, as chosen by the nixpkgs maintainers).

### C. Comparison with the minimal ISO image

As part of RQ2, we investigate the difference of reproducibility rate between critical packages whose reproducibility is monitored and the rest of the package set. We are also interested in knowing whether observing the reproducibility health of this subset of packages gives a good enough information on

the state of the rest of the project. The minimal and gnome-based ISO images are considered critical subsets of packages and benefit from a community-maintained reproducibility monitoring. We study the minimal ISO image because it contains a limited amount of core packages. We evaluate the Nix expression associated with the image, compute its runtime closure (the set of packages included in the image) and match it with the packages of our dataset to infer their reproducibility statuses.

### D. Analyzing causes of unreproducibility using diffoscopes

Analyzing causes of unreproducibility is a tricky debugging activity, usually carried out by practitioners (in particular, by Linux distribution maintainers and members of the Reproducible Builds project). Some automatic fault localization methods have been proposed [30], but they rely on instrumenting the build, while we have to run the Nix builds unchanged to avoid introducing biases.

For each unreproducible output, we run `diffoscope` with a 5-minute timeout, yielding a dataset of 86476 diffoscopes. We then investigate whether we can use our large dataset of diffoscopes for automatic detection of causes of non-reproducibility. The `diffoscope` tool was mainly designed to help human debugging, but it also supports producing a JSON output, which can then be machine processed.

We wish to explore heuristics that can be applied at the line level, so we recurse through `diffoscope` structures until leaf nodes, which are diffs in unified diff format. We randomly draw one added line from 10000 diffoscopes, sort them by similarity to ease visual inspection, and manually inspect them to derive relevant heuristics. We then run these heuristics on the full `diffoscope` dataset to determine the proportion of packages impacted by each cause (multiple causes can apply to the same package). The first and last author then evaluate the precision of each these heuristics by manually counting false positives in samples of matched lines for each heuristic.

### E. Automatic identification of reproducibility fixes

To investigate fixes to unreproducibilities, for each unreproducible package that becomes reproducible, we run an automatic bisection process to find the first commit that fixes the reproducibility. By looking into the corresponding pull request on GitHub, we check if the maintainers provide information on why this fixes a reproducibility issue, or link to a corresponding bug report. In particular, we are interested to check how often the maintainers are aware that their commit is a reproducibility fix (as opposed to a routine package update, which embeds a reproducibility fix that would have been crafted by the upstream maintainers).

We start from the set of packages from all revisions, and we look specifically at packages that change status from unreproducible to reproducible in two successive revisions. These are our candidate packages (and “old” and “new” commits) for the bisection process.

Since we perform the bisection process on a different runner and at a different time compared to the dataset creation, it can happen that we cannot reproduce the status (reproducible or unreproducible) of some builds. Therefore, before starting the bisection, we verify that we obtain consistent results on the “old” and the “new” revision. Then, for those which behave as expected, we start an automatic `git bisect` process.

The script used for the automatic `git bisect` checks for the reproducibility status of the build to mark the selected commit as “old” or “new”. Commits that fail to build or are not available in cache are marked as “skipped”. We use `git bisect` with the `--first-parent` flag because intermediate pull request commits are typically not in cache.

For the qualitative analysis, we first group packages by fixing commit, as seeing all packages fixed by a given commit gives valuable information that might help to understand the reproducibility failure being fixed. We then randomly sample fixes and open their commit page on GitHub, locating the corresponding pull request. We manually inspect the pull request (description, commit log, code changes) to first confirm that the bisect phase successfully identified the commit that fixed the reproducibility failure. It may be the case that after careful inspection, the change looks unrelated to the package being fixed (the bisection process can give incoherent results in case of a flaky reproducibility issue or because the package changed status several times between two data points) in which case we discard it. Once we have confirmed that the identified commit is correct, we check whether the commit authors indicate that they are fixing a reproducibility issue and if the commit is a package update or another change. We analyze 100 randomly sampled reproducibility fixes and report our findings. Additionally, we perform the same analysis on the 15 commits that fix the most packages (from 3052 down to 27 packages fixed) to find potential differences of behavior of the contributors for those larger-scale fixes.

### V. DATASET

The main result of running the pipeline of Figure 2 is a large-scale dataset of historical package rebuilds, including

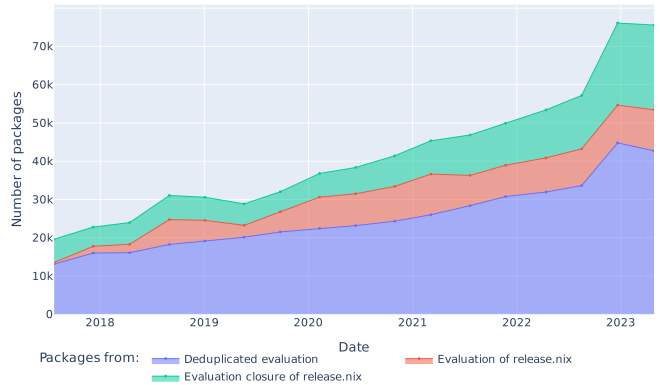


Fig. 3. Evolution of the number of packages in each nixpkgs revision (as defined by `release.nix`), in their evaluation closure and after deduplicating ecosystem copies.

(re)build information, bitwise reproducibility status and, in case of non-reproducibility, generated diffoscopes. In this paper, we use the dataset to answer our stated research questions, but many other research questions could be addressed using the dataset, including more in-depth analysis of non-reproducibility causes. We make the dataset available to the research and technical community to foster further exploration on the topic. In the remainder of this section, we provide some descriptive statistics of the dataset.

The dataset spans 709 816 package builds coming from 17 nixpkgs revisions, built over a total of 14 296 hours. From those builds, 548 390 are coming directly from the `release.nix` file and can be tracked by name. They correspond to 58 103 unique packages that appear over the span of sampled revisions. As can be seen on Figure 3, the number of packages listed to be built by Hydra increased from 13 527 in 2017 to 53 432 in 2023. Ecosystems can be present in multiple versions. On average, deduplicating packages in multiple ecosystem copies decreases their number by 21% while adding the evaluation closure of the `release.nix` file increases the number of jobs by 29%.

Figure 4 shows the evolution of the top 9 ecosystem sizes in nixpkgs, plus the base namespace. 61.7% of packages belong to an ecosystem, while the rest live in nixpkgs base namespace. The three largest ecosystems are Haskell, Python and Perl, which together account for 42.4% of the packages.

Figure 5 outlines the number of packages introduced in the dataset by each revision, and their survival over time. In particular, as of July 2017 the package set contained 13 114 elements, 8929 of which were still present in April 2023.

### VI. RESULTS

We present our experimental results below, organized by research question. Their discussion is provided later, in Section VII.

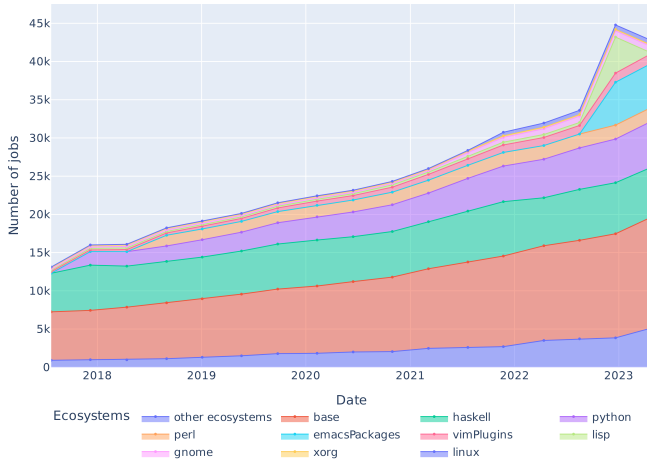


Fig. 4. Evolution of the size of the nine most popular software ecosystems in nixpkgs, the packages whose ecosystem is undetermined (base), and the packages from other ecosystems (only packages listed in `release.nix`).

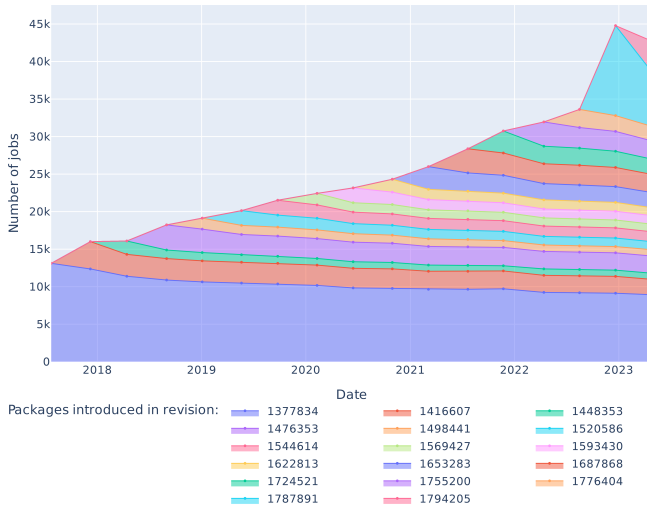


Fig. 5. Number of packages introduced by every revision of the dataset and their survival in the package set over time.

**A. RQ0: Does Nix allow rebuilding past packages reliably (even if not bitwise reproducibly)?**

This research question aims to reproduce the results from Malka *et al.* [21], as a starting baseline. That earlier work only built *one* nixpkgs revision, the most ancient in their dataset; in our case, we rebuilt that revision alongside with 16 others, evenly spaced over time to study *trends*. Figure 6 shows the proportion of packages between 2017 and 2023 that we successfully rebuilt (not necessarily in a bitwise reproducible manner, merely “successfully built” for this RQ).

This proportion varies between 99.68% and 99.95%, confirming previously reported findings: Nix reproducibility of build environments allows for very high rebuildability rate over time. Note that this is not an exact replication of the revision in [21], because we also included packages not explicitly listed in `release.nix`, but present in the dependency graph,

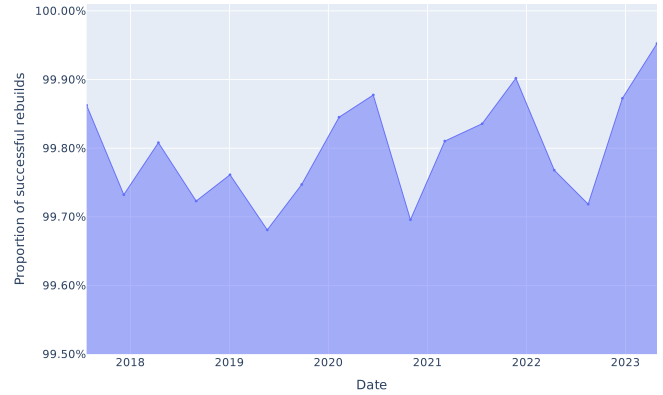


Fig. 6. Proportion of rebuildable packages over time.

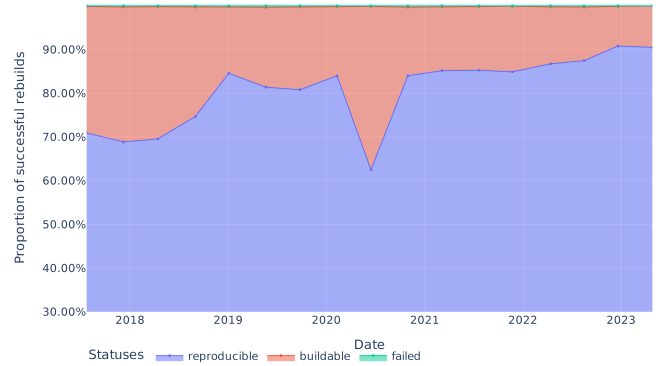


Fig. 7. Proportion of reproducible, rebuildable (but unreproducible) and non-rebuildable packages over time.

whereas they did not.

**B. RQ1: What is the evolution of bitwise reproducible packages in nixpkgs between 2017 and 2023?**

Apart from a significant regression in 2020, we obtain bitwise reproducibility levels between 69% and 91% (see Figure 7). The trends in Figure 8 show that the absolute number of bitwise reproducible packages has consistently gone up and followed the fast growth of the package set. The only exception is the data point for June 2020, where the number of reproducible packages dropped even though the total number of packages grew. We study and explain this reproducibility regression in Section VI-C below.

Figure 9 shows for each revision the cumulative amount of unreproducibilities introduced by that revision getting fixed over time. The large slope between the two first points of each plot indicates that most of the unreproducibilities introduced in a revision are fixed in the next revision, on average 62% of them (even raising to 85% if we account for packages fixed after the 2020 reproducibility regression).

Excluding the June 2020 revision (corresponding to the observed bitwise reproducibility regression), Figure 10 depicts the average evolution of the package set between two consecutive revisions. In particular, only 0.60% of the packages transition from *reproducible* to *buildable but not reproducible*



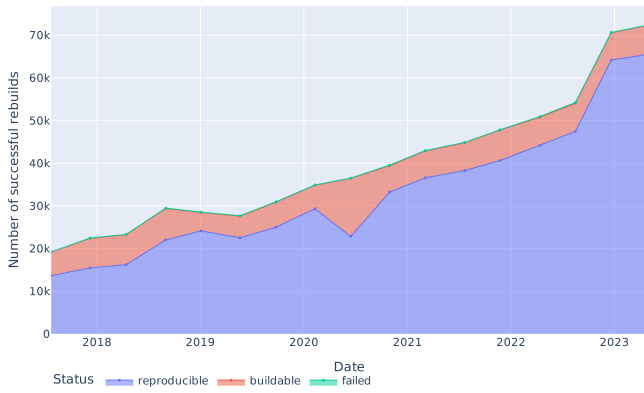


Fig. 8. Absolute numbers of reproducible, rebuildable (but unreproducible) and non-rebuildable packages over time.

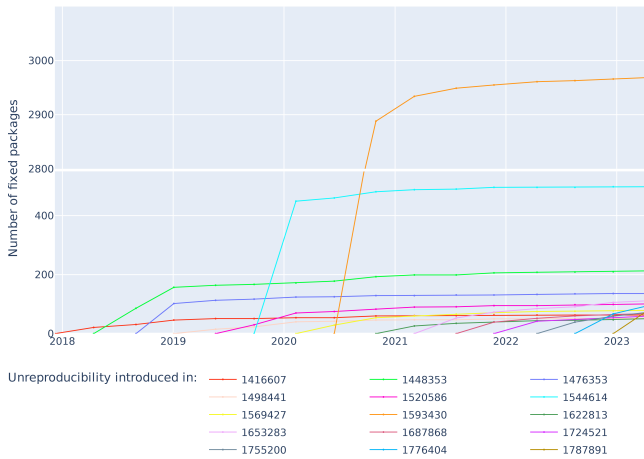


Fig. 9. Evolution of the cumulative number of fixes to non-reproducibilities, separated by the revision in which the non-reproducibilities were introduced.

status between two revisions, while 2.07% of the *buildable but not reproducible* packages become reproducible. The average growth rate of the package set is 1.07 and, on average, 80.02% of the new packages are reproducible.

### C. RQ2: What are the unreproducible packages?

We observe large disparities both in trends and in reproducibility by package ecosystem (see Figure 11). In the top

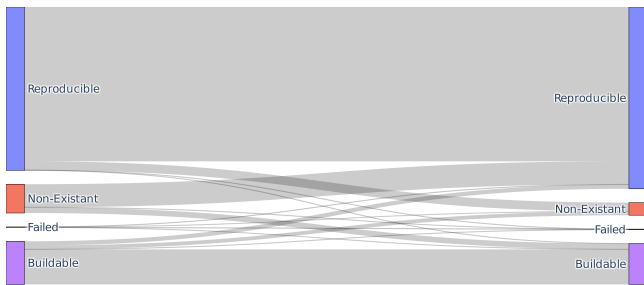


Fig. 10. Sankey graph of the average flow of packages between two revisions, excluding the revision from June 2020, considered as an outlier.

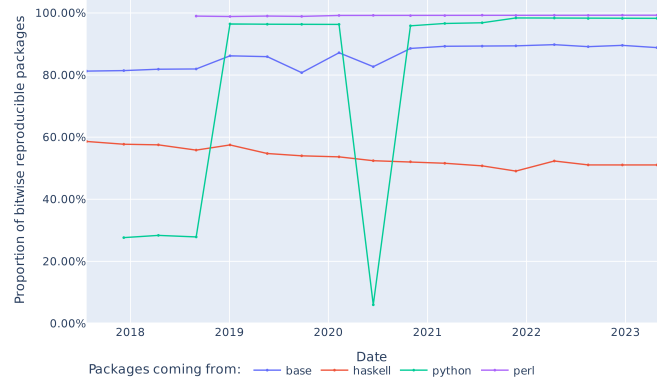


Fig. 11. Proportion of reproducible packages belonging to the three most popular ecosystems and the base namespace of nixpkgs.

three most popular ecosystems, Perl has consistently maintained a proportion of reproducible packages above 98%, while Haskell reproducibility rate stagnated around the 60% mark, even decreasing by more than 7 percentage points during the time of our study.<sup>5</sup> The Python ecosystem on the contrary sees a positive evolution over time, with reproducibility rates as low as 27.64% in December 2017, reaching 98.28% in April 2023. As can be seen on Figure 11, the Python ecosystem has however known a major dip in reproducibility in June 2020, with a drop to 6.01% (for almost  $-90$  percentage points!).

As can be seen in Figure 12, this dip in the proportion of reproducible Python packages can be explained by a large number of packages transitioning from the reproducible to the buildable but not reproducible state, indicating a regression in bitwise reproducibility at that time. To identify the root cause of that regression, we used a git bisection on the nixpkgs repository between June and October 2020 in order to identify the commit fixing the unreproducibility issue, and derived that the root cause of the regression was an update in the `pip` executable changing the behavior of byte-compilation.<sup>6</sup>

Figure 13 shows the difference in reproducibility rates between the NixOS minimal ISO image—a package set for which there exists community-based reproducibility monitoring, and made of packages that are considered critical—and the whole package set. The minimal ISO image has a very high proportion of reproducible packages in the considered period, with rates consistently higher than 95% starting from May 2019. Its reproducibility rates however do not follow the evolution of the overall package set, such that observing the reproducibility of the minimal ISO image does not give any clue about the reproducibility of the package set as a whole.

### D. RQ3: Why are packages unreproducible?

We identified four heuristics that are both *effective* (they give a large number of matches in our diffoscope dataset) and *relevant* (they correspond to a software engineering practice

<sup>5</sup>A fix to a long-standing reproducibility issue has been introduced in GHC 9.12.1 (new `-fobject-determinism` flag). As of January 2025, nixpkgs does not make use of this flag to improve Haskell build reproducibility.

<sup>6</sup>Details can be found in <https://github.com/pypa/pip/issues/7808>.

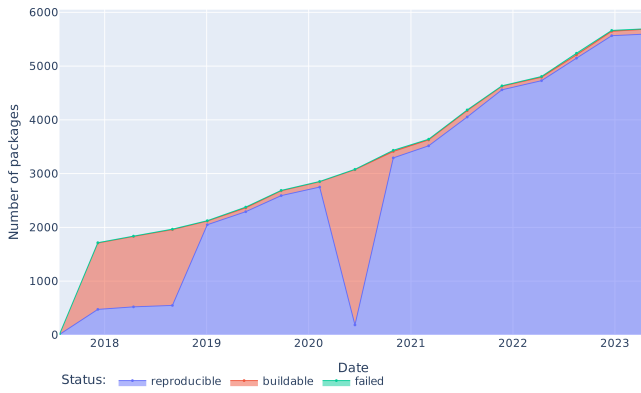


Fig. 12. Evolution of the absolute number of reproducible, rebuildable (but unreproducible) and non-rebuildable packages from the Python ecosystem.

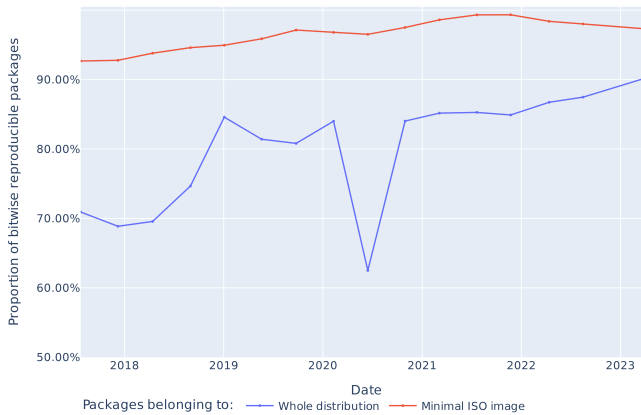


Fig. 13. Evolution of the proportion of reproducible packages belonging to the minimal ISO image and in the entire package set.

that can be changed) to automatically determine why packages are not bitwise reproducible:

- **Embedded dates:** presence of build dates in various places in the built artifacts (version file, log file, file names, etc.). To detect dates embedded in build outputs, we look specifically for the year 2024 in added lines, given that we ran all our builds during 2024, but none of the historical builds ran during this year.
- **uname output:** some unreproducible builds embed build information such as the machine that the build was run on. While the real host name is hidden by the Nix sandbox, other pieces of information (such as the Linux kernel version or OS version, reported by `uname`) are still available.
- **Environment variables:** some builds embed some or all available environment variables in their build outputs. This typically causes unreproducibility because an environment variable containing the number of cores available to build on the machine is set by Nix.
- **Build ID:** some ecosystems (Go for example) embed a unique (but not deterministic) build ID into the artifacts.

Despite a well-known recommendation by the Reproducible Builds project to avoid embedding build dates in build artifacts

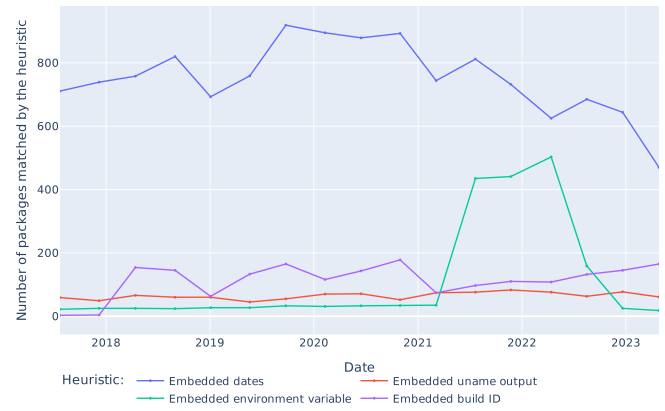


Fig. 14. Evolution of the number of packages for which we generated diffscopes that are matched by each of our heuristics, over time.

to achieve bitwise reproducibility, we find that 12 831 of our 86 317 packages with diffscopes contain a date, accounting for 14.8% of them. Still in the most recent nixpkgs revision, we find 470 instances of this non-reproducibility cause, showing that embedding dates is still an existing practice. Additionally, we find that embedded environment variables and build IDs each account for 2.2% of our unreproducible packages. Finally, we find `uname` outputs in 1097 of our unreproducible builds (1.3%), 721 of which also include a date as part of the `uname` output. Figure 14 shows the evolution of the number of packages matched by each heuristic over time. Altogether, a total of 19.7% of the packages for which we have generated a diffscope have an unreproducibility that can be explained by at least one of our heuristics.

We evaluate the precision of each heuristic by manually verifying 500 matched lines for every heuristic and counting false positives. For the `uname`, `build ID`, and `environment variables` heuristics, we report a precision of 100%. For the date heuristic, we obtain a precision of 97.8% (11 false positives).

#### E. RQ4: How are unreproducibilities fixed?

Our analysis of 100 randomly sampled reproducibility fixes showed that, in most cases, authors are not aware that they are fixing a reproducibility issue, or they do not mention it: in 93 instances, we did not find any trace of reproducibility being mentioned. In fact, in 76 cases we found out that the reproducibility fix was just a routine package update, and it is very likely that the package becoming reproducible had more to do with changes in the upstream software than with a conscious action from the package maintainer. Other fixes included internal nixpkgs changes that had reproducibility impact as a *side effect* of the main reason behind the change.

Our study of the 15 most impactful fixes suggests that those are more often done with the intent of fixing reproducibility: in more than half of them (8 out of 15) the author mentioned and documented the reproducibility issue being fixed and 9 of them were changes internal to nixpkgs. Some large-scale

reproducibility fixes were still package updates like toolchain updates.

## VII. DISCUSSION

This work brings valuable insights to the ongoing discussions about software supply chain security, reproducible builds, and functional package management.

*False myth: reproducible builds (R-B) do not scale:*

One strongly held belief about R-B is that they work well in limited contexts—either selected “critical” applications (e.g., cryptocurrencies or anonymity technology) or strictly curated distributions—but either do not scale or are not worth the effort in larger settings. Our results can be interpreted as counterproof of this belief: in nixpkgs, the largest general-purpose repository with more than 70k packages (in April 2023), more than 90.5% packages are nowadays bitwise reproducible from source. R-B do scale to general purpose, regularly used software.

*Recommendation: invest in infrastructure for binary caches and build attestations:* To reap the security benefits of the R-B vision [16] in practice, users need multiple components: (1) signed attestations by each independent builder, publishing the obtained checksums at the end of a build; (2) caches of built artifacts, so that users can avoid rebuilding from source; (3) verification technology in package managers to verify that built artifacts in (2) match the consensus checksums in (1). With few exceptions, the infrastructure to support all this does not exist yet. Now that the “R-B do not scale” excuse is out of the picture, we call for investments to develop this infrastructure, preferably in a distributed architectural style to make it more secure, robust, and sustainable.

*False myth: Nix implies bitwise reproducibility:* Contradictory to the previous myth, there is also a belief that Nix (or functional package management more generally) *implies* bitwise reproducibility. Our results disprove this belief: about 9.4% of nixpkgs packages are not bitwise reproducible. This is not surprising, because R-B violations happen for multiple reasons, only some of which are mitigated by FPM *per se*.

*Recommendation: use FPM for bitwise reproducibility and rebuildability:* Still, Nix appears to perform really well at bitwise reproducibility “out of the box”, and even more so at rebuildability (above 99%). Based on this, we recommend to use Nix, or FPM more generally, for all use cases that require either mere rebuildability or full bitwise reproducibility. At worse, they provide a very good starting point.

This work has investigated the causes of the lingering *non* reproducibility in nixpkgs, but not those of reproducibility; it would be interesting to know why Nix performs so well, possibly for adoption in different contexts. It is possible that bitwise reproducibility is an emerging property of FPM, or that it comes from technical measures during build like sandboxing, or that Nix is simply benefiting (more than other distributions?) from the decade-long efforts of the R-B project in upstream toolchains. Exploring all this is left as future work.

*QA monitoring for bitwise reproducibility:* The significantly higher (and stably so) bitwise reproducibility performances of the NixOS minimal ISO image (see Figure 13) suggests that quality assurance (QA) monitoring for build reproducibility is an effective way to increase it. The fact that Debian uses a similar approach with good results [16] is compatible with this interpretation. A deeper analysis of the relationship between QA monitoring and reproducibility rate is out of scope for this work, but it is quite likely that extending reproducibility monitoring to all packages will result in an easy win for nixpkgs (assuming bitwise reproducibility is seen as a project goal).

## VIII. THREATS TO VALIDITY

### A. Construct validity

We rebuilt Nix packages from revisions in the period July 2017–April 2023, for about 6 years. That is enough to observe arguably long-term trends, which also appear to be stable in our analysis; except for one temporary regression, which we analyzed and explained. Still, we cannot exclude significant differences in reproducibility trends before/after the studied period.

Due to the high computation cost, build time, and environmental impact of rebuilding general purpose software from source, we sampled nixpkgs revisions to rebuild uniformly (every 4.1 months) within the studied period, totaling 14 296 hours of build time. We cannot exclude trend anomalies *between sampled revisions* either, but that seems unlikely due to the stability of the observed long-term trends. More importantly, this means that we are less likely to catch short-spanned reproducibility regressions, which would be introduced and fixed within the same period between two sampled revisions. This can be improved upon by sampling and rebuilding more nixpkgs revisions, complementing this work.

When rebuilding a given package, we relied on the Nix binary cache for all its transitive dependencies. As we have rebuilt *all* packages from any sampled nixpkgs revisions, our package coverage is complete. But this way we have not measured the impact of unreproducible packages on transitive reverse dependencies, i.e., how many *additional* packages become unreproducible if systematically built from scratch, including all their dependencies? Our experiment matches real-world use cases and is hence adequate to answer our RQs, but it would still be interesting to know.

Also, during rebuilds we have not attempted to re-download online assets, but relied on the Nix cache. Hence, we have not measured the impact of lacking digital preservation practices on reproducibility and rebuildability. It would be interesting and possible to measure such impact by, e.g., first trying to download assets from their original hosting places and, for source code assets, fallback to archives such as Software Heritage [6] in case of failure.

### B. External validity

We have rebuilt packages from nixpkgs, the largest cross-ecosystem FOSS distribution, using the Nix functional pack-

age manager (FPM). Other FPMs with significant user bases exist, such as Guix [7]. Given the similarity in design choices, we do not have reasons to believe that Guix would perform any differently in terms of build reproducibility than Nix, but we have not empirically verified it; doing so would be useful complementary work.

Neither did we verify historical build reproducibility in classic FOSS distributions (e.g., Debian, Fedora, etc.), as out of the FPM scope. Doing so would still be interesting, for comparison purposes. It would be more challenging, though, due to the fact that outside the FPM context, it is significantly harder to recreate exact build environments from years ago.

## IX. CONCLUSION

In this work we conducted the first large-scale experiment of bitwise reproducibility in the context of the Nix functional package manager, by rebuilding 709 816 packages coming from 17 revisions of the nixpkgs software repository, sampled every 4.1 months from 2017 to 2023. Our findings show that bitwise reproducibility in nixpkgs is very high and has known an upward trend, from 69% in 2017 to 91% in 2023. The mere ability to rebuild packages (whether bitwise reproducibly or not) is even higher, stably around 99.8%.

We have highlighted disparities in reproducibility across ecosystems that coexist in nixpkgs, as well as between packages for which bitwise reproducibility is actively monitored and the others. We have developed heuristics to understand common (un)reproducibility causes, finding that 15% of unreproducible packages were embedding the date during the build process. Finally, we studied reproducibility fixes and found out that only a minority of changes inducing a reproducibility fix were done intentionally; the rest appear to be incidental.

## DATA AVAILABILITY STATEMENT

The dataset produced as part of this work is archived on and available from Zenodo [18, 19]. A full replication package containing the code used to run the experiment describe in this paper is archived on and available from Software Heritage [20].

## ACKNOWLEDGMENTS

This work would have not been possible without the availability of historical data from the official Nix package cache at <https://cache.nixos.org> with its long-standing “no expiry” policy. In the context of ongoing discussions to prune the cache, we strongly emphasize its usefulness for performing empirical research on package reproducibility, like this work. We also thank the NixOS Foundation for guaranteeing us that the revisions that we needed for this research would not be garbage-collected.

## REFERENCES

- [1] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. “Solar Winds Hack: In-Depth Analysis and Countermeasures”. In: *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. July 2021, pp. 1–7. URL: <https://ieeexplore.ieee.org/document/9579611> (visited on 11/08/2024).
- [2] Rahul Bajaj, Eduardo Fernandes, Bram Adams, and Ahmed E. Hassan. “Unreproducible builds: time to fix, causes, and correlation with external ecosystem factors”. en. In: *Empirical Software Engineering* 29.1 (Nov. 2023), p. 11. ISSN: 1573-7616. URL: <https://doi.org/10.1007/s10664-023-10399-4> (visited on 04/26/2024).
- [3] *Build path — reproducible-builds.org*. URL: <https://reproducible-builds.org/docs/build-path/> (visited on 10/21/2024).
- [4] *Buildbot*. URL: <https://buildbot.net/> (visited on 10/24/2024).
- [5] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. “On business adoption and use of reproducible builds for open and closed source software”. en. In: *Software Quality Journal* 31.3 (Sept. 2023), pp. 687–719. ISSN: 1573-1367. URL: <https://doi.org/10.1007/s11219-022-09607-z> (visited on 11/15/2023).
- [6] Roberto Di Cosmo and Stefano Zacchiroli. “Software Heritage: Why and How to Preserve Software Source Code”. In: *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan, September 25-29, 2017*. Ed. by Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto. 2017. URL: <https://hdl.handle.net/11353/10.931064>.
- [7] Ludovic Courtès. *Functional Package Management with Guix*. arXiv:1305.4584 [cs]. May 2013. URL: <http://arxiv.org/abs/1305.4584> (visited on 03/16/2023).
- [8] Ludovic Courtès, Timothy Sample, Stefano Zacchiroli, and Simon Tournier. “Source Code Archiving to the Rescue of Reproducible Deployment”. In: *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability, ACM REP 2024, Rennes, France, June 18-20, 2024*. ACM, 2024. URL: <https://doi.org/10.1145/3641525.3663622>.
- [9] *Cyber Resilience Act — Shaping Europe’s digital future*. en. Sept. 2022. URL: <https://web.archive.org/web/20231109015038/https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act> (visited on 11/15/2023).
- [10] Alexandre Decan, Tom Mens, and Philippe Grosjean. “An empirical comparison of dependency network evolution in seven software packaging ecosystems”. en. In: *Empirical Software Engineering* 24.1 (Feb. 2019), pp. 381–416. ISSN: 1573-7616. URL: <https://doi.org/10.1007/s10664-017-9589-y> (visited on 05/12/2021).
- [11] *diffoscope: in-depth comparison of files, archives, and directories*. URL: <https://diffoscope.org/> (visited on 11/07/2024).
- [12] Eelco Dolstra. “The purely functional software deployment model”. en. OCLC: 71702886. PhD thesis. S.l.: s.n., 2006.
- [13] Eelco Dolstra and Eelco Visser. “The Nix Build Farm: A Declarative Approach to Continuous Integration”. en. In: *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*. 2008.
- [14] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. “It’s like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1527–1544. URL: <https://ieeexplore.ieee.org/abstract/document/10179320/> (visited on 10/17/2024).
- [15] The White House. *Executive Order on Improving the Nation’s Cybersecurity*. en-US. May 2021. URL: <https://web.archive.org/web/20231114135442/https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/> (visited on 11/15/2023).
- [16] Chris Lamb and Stefano Zacchiroli. “Reproducible Builds: Increasing the Integrity of Software Supply Chains”. In: *IEEE Software* 39.2 (Mar. 2022), pp. 62–70. ISSN: 1937-4194. URL: <https://ieeexplore.ieee.org/abstract/document/9403390> (visited on 11/15/2023).
- [17] Damien Legay, Alexandre Decan, and Tom Mens. “A Quantitative Assessment of Package Freshness in Linux Distributions”. en. In: *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*. Madrid, Spain: IEEE, May 2021, pp. 9–16. ISBN: 978-1-66544-557-3. URL: <https://ieeexplore.ieee.org/document/9474659/> (visited on 11/07/2024).
- [18] Julien Malka. *Replication package for: Does Functional Package Management Enable Reproducible Builds at Scale? Yes. - Build metadatas and logs*. Zenodo, Jan. 2025. URL: <https://doi.org/10.5281/zenodo.14736078>.
- [19] Julien Malka. *Replication package for: Does Functional Package Management Enable Reproducible Builds at Scale? Yes. - Diffoscopes*. Zenodo, Jan. 2025. URL: <https://doi.org/10.5281/zenodo.14728623>.
- [20] [SW Rel.] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. *Code archive for the paper “Does Functional Package Management Enable Reproducible Builds at Scale? Yes.”* version 1.0, 2025. URL: <https://gitlab.telecom-paris.fr/julien.malka/does-functional-package-management-enable-reproducible-builds-at-scale>, SWHID: {swh:1:rev:93663673a99ac5df6d4a964665a542404a06ae26}.
- [21] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. “Reproducibility of Build Environments through Space and Time”. In: *Proceedings of the*

- 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 97–101. ISBN: 9798400705007. URL: <https://dl.acm.org/doi/10.1145/3639476.3639767> (visited on 10/20/2024).
- [22] Tom Mens and Alexandre Decan. *An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries*. arXiv:2409.18884. Oct. 2024. URL: <http://arxiv.org/abs/2409.18884> (visited on 11/07/2024).
- [23] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. “Reproducible Containers”. In: ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 167–182. ISBN: 978-1-4503-7102-5. URL: <https://dl.acm.org/doi/10.1145/3373376.3378519> (visited on 03/17/2023).
- [24] *NixOS Reproducible Builds*. en. URL: <https://reproducible.nixos.org/> (visited on 10/21/2024).
- [25] *NVD - CVE-2024-3094*. URL: <https://nvd.nist.gov/vuln/detail/CVE-2024-3094?ref=thetack.technology> (visited on 11/08/2024).
- [26] *Overview of various statistics about reproducible builds*. URL: <https://tests.reproducible-builds.org/debian/reproducible.html> (visited on 10/21/2024).
- [27] Manuel Pöll and Michael Roland. “Analyzing the Reproducibility of System Image Builds from the Android Open Source Project”. In: (2021). URL: [https://www.digidow.eu/publications/2021-poell-tr-reproducibilityaospsystemimages/Poell\\_2021\\_ReproducibilityAOSPSystemImages.pdf](https://www.digidow.eu/publications/2021-poell-tr-reproducibilityaospsystemimages/Poell_2021_ReproducibilityAOSPSystemImages.pdf) (visited on 10/17/2024).
- [28] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. “Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems”. In: *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 654–664. URL: <https://ieeexplore.ieee.org/abstract/document/10555868/> (visited on 10/17/2024).
- [29] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. “Automated localization for unreproducible builds”. en. In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 71–81. ISBN: 978-1-4503-5638-1. URL: <https://dl.acm.org/doi/10.1145/3180155.3180224> (visited on 10/17/2024).
- [30] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. “Root cause localization for unreproducible builds via causality analysis over system call tracing”. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ASE '19. San Diego, California: IEEE Press, Feb. 2020, pp. 527–538. ISBN: 978-1-72812-508-4. URL: <https://doi.org/10.1109/ASE.2019.00056> (visited on 09/04/2023).
- [31] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. “Automated patching for unreproducible builds”. In: ICSE '22. New York, NY, USA: Association for Computing Machinery, July 2022, pp. 200–211. ISBN: 978-1-4503-9221-1. URL: <https://doi.org/10.1145/3510003.3510102> (visited on 03/17/2023).
- [32] *Reproducible Builds — a set of software development practices that create an independently-verifiable path from source to binary code*. Nov. 2023. URL: <https://web.archive.org/web/20231113151826/https://reproducible-builds.org/> (visited on 11/15/2023).
- [33] *SOURCE\_DATE\_EPOCH — reproducible-builds.org*. URL: <https://reproducible-builds.org/docs/source-date-epoch/> (visited on 10/21/2024).
- [34] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. “in-toto: Providing farm-to-table guarantees for bits and bytes”. en. In: 2019, pp. 1393–1410. ISBN: 978-1-939133-06-9. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias> (visited on 09/04/2023).