A Decade of Software Reproducibility in the Nix Package Ecosystem

Julien Malka, Stefano Zacchiroli, Théo Zimmermann

Empirical Software Engineering manuscript No.
(will be inserted by the editor)

A Decade of Software Reproducibility in the Nix Package Ecosystem

Julien Malka · Stefano Zacchiroli · Théo Zimmermann

Received: date / Accepted: date

Abstract We report a large-scale empirical study of two aspects of software reproducibility—rebuildability and bitwise reproducibility—in the Nix package ecosystem. Using 29 evenly spaced historical snapshots over a decade of history of the nixpkgs repository (2015–2024) we attempted to rebuild and bitwise-compare the build outputs of tens of thousands of packages per snapshot. Our experiment produced a dataset of build metadata and logs for 1 321 000 package builds and preserved 166 523 diffoscopes for unreproducible outputs.

We find that functional package management enables extremely high rebuildability over time (near-universal ability to reconstitute historical build environments and rebuild software packages), while bitwise reproducibility has steadily improved and reaches a high point in recent years (up to 93% in 2024). Early years show substantially lower bitwise reproducibility, indicating that functional package management alone does not guarantee bitwise-identical outputs, and that the observed high level of bitwise reproducibility is not solely due to the package management approach. Common causes of unreproducibility, both in the rebuildability and bitwise reproducibility dimensions, include management of dates in build and test processes; we quantify their prevalence and other common causes using manual analysis of logs of rebuild failures and automated analysis of diffoscopes.

Julien Malka (Corresponding Author)
LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France
E-mail: julien.malka@telecom-paris.fr

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

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

Our dataset (build logs, metadata, and diffoscopes) and analysis scripts are released to support future research on debugging, repairing, and tracking reproducibility at scale. We discuss implications for package managers, distribution maintainers, and reproducible-builds practitioners, and identify actionable mitigations to increase reproducibility across large software distributions.

1 Introduction

Software reproducibility is a multifaceted concept with significant implications for software engineering practices. Depending on the kind of applications that practitioners are targetting, they might associate it with different properties of software builds. In its weakest form, it refers to the ability to recompile source code into an executable, even years after its initial creation [21, 19]. This property is essential for long-term software preservation, as it ensures that as long as the source code is accessible, the software can still be built and executed. In this article, we will refer to this form of software reproducibility as *rebuildability*. In its strongest form, the requirement goes further: recompilation must yield results that are exactly identical to the original, down to the level of individual bits, which also guarantees the two artifacts will behave identically under the same conditions. This is the concept of *reproducible builds*, which is promoted by the project of the same name [46]. In this article, we prefer the more explicit term *bitwise reproducibility* [6], to avoid the possible confusion between the notion of reproducible builds and weaker notions of reproducibility. Bitwise reproducibility is a promising way to avoid having to trust third-party vendors who distribute executable software in binary form. Indeed, if a piece of open source software is bitwise reproducible, a user may rely on a network of independent software (re)builders to reach a consensus on what the "right" result of building the software is, and verify its checksum before downloading built artifacts from a specific vendor. For an attacker wanting to compromise the software supply chain of that software component, it is no longer sufficient to compromise only one of the involved parties [22].

However, despite those advantages, neither rebuildability nor bitwise reproducibility are exempt of challenges: they both suppose as a prerequisite the ability to reproduce the environment containing the exact set of dependencies that was used to compile the project initially—the *build environment*. This assumption does not hold true in most cases, as independant evolutions of dependencies within software distributions might create cases where it is practically impossible to assemble a build environment with the expected dependency versions. Sometimes, it is even impossible to recover the needed information on what were the precise versions used at the time of the initial build. Even when reproducing the build environment is possible, sources of non-determinism in the build process might hinder bitwise reproducibility, to the point that a recent study [18] which interviewed 24 experts of reproducible builds (R-B) 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 bitwise reproducibility levels like Debian, which consistently achieves a reproducibility rate of more than 95% [37], 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.

Functional package management (FPM) is a deployment model that aims to address the challenges of build environment reproducibility by applying the principles of functional programming to package management [15, 9]. FPM packages are *pure functions* (in the mathematical sense) from build- and runtime dependencies to build artifacts, described as "build recipes" that are executed locally by the package manager when users request to install a package. Components are built in a sandboxed environment, disallowing access to unspecified dependencies, even if they are present on the system.

Nix [35] is the initial and most widely used implementation of the FPM model. It comes with a package repository called *nixpkgs*, which is the largest cross-ecosystem distribution of free and open source software (FOSS), totaling as of October 2025 over 100 000 packages.[1] It includes components from a large variety of software ecosystems, making it an interesting target to study reproducibility at scale. However, despite the potential for insightful distribution-wide metrics, nixpkgs limits its monitoring of bitwise reproducibility to the narrow set of packages included in the minimal and GNOME-based ISO images [36], where a reproducibility rate higher than 95% is consistently reported.

In a seminal ICSE NIER paper [29], we have shown that the functional package management model allowed for reproduction of build environments in 99.99% of cases, out of about 7 million Nix expressions sampled over time. These results paved the way for further investigations on the reproducibility properties enabled by functional package managers. In the same study, we further asked whether the property of being able to reproduce build environments was sufficient to achieve good rebuildability levels, and we started answering it by rebuilding a single 6-year-old revision of nixpkgs (for a total of 14 thousand software packages built) with a success rate of 99.94%. In the current paper, we go further on several aspects:

- We study reproducibility properties over a longer time span, one full decade from 2015 to 2024, on a sample of 29 evenly spaced revisions of nixpkgs. Note that building one *single* revision of nixpkgs means building *all packages* referenced by it, or about 100 *thousand* software packages in recent revisions (fewer for earlier ones);
- We consider two dimensions of reproducibility: rebuildability and bitwise reproducibility. We investigate for each revision the proportion of packages

[1] Based on the Repology rankings `https://repology.org`, accessed Oct. 2025.

that can be rebuilt, the proportion of packages that can be rebuilt bitwise reproducibly, and the causes of failures to achieve these properties.

Contributions In this work, we perform the largest scale empirical study of reproducibility of open source software going back in time, rebuilding historical packages from evenly spaced snapshots of the nixpkgs package repository taken every 4.1 months from 2015 to 2024. With this experiment, we answer the following research questions:

- **RQ1: Does functional package management allow rebuilding past packages reliably at scale** (even if not bitwise reproducibly)**?**
  - **RQ1.1: What is the evolution of rebuildable packages in nixpkgs between 2015 and 2024?** Is it harder (i.e., more rebuild failures) to rebuild older packages than newer ones?
  - **RQ1.2: What are the causes of rebuild failures?** Can we distinguish between transient failures (flakiness) and permanent failures? Can we identify common causes of failures from build logs?
  - **RQ1.3: What is the cascading impact** of rebuildability failures on the package dependency graph?
- **RQ2: Does functional package management allow bitwise reproducibility at scale?**
  - **RQ2.1: What is the evolution of bitwise reproducible packages in nixpkgs between 2015 and 2024?** How does the rate of bitwise reproducibility evolve over time? Are unreproducible packages eventually fixed? Do reproducible packages remain reproducible?
  - **RQ2.2: What are the bitwise unreproducible packages?** Are they concentrated in specific ecosystems? Are critical packages more likely to be reproducible?
  - **RQ2.3: Why are packages bitwise unreproducible?** Is large-scale identification of common causes possible?

Results Thanks to this large-scale experiment, we are able to establish that **bitwise reproducibility is achievable at scale**, with a reproducibility rate reaching 93% in 2024. However, going back as far as 2015, we observe that such high bitwise reproducibility levels are not a necessary consequence of the use of functional package management, since in the early years, we only get a reproducibility rate of 35%. The general upward trend in bitwise reproducibility over time happened despite a continuous increase in the number of packages in nixpkgs.

Looking at rebuildability, we find an extremely high rebuildability rate over the whole time span, to the exception of one notable regression in 2015, that we analyze in detail.

By investigating reproducibility failures, we are able to draw implications for practitioners, highlighting in particular the large importance of tests in rebuildability failures, and some common and easily avoidable causes of bitwise unreproducibility. Dates, in particular, are a major cause of both rebuildability

failures (because of tests relying on current dates) and bitwise unreproducibility (because of embedded build dates in artifacts).

As part of this work, we introduce a **novel dataset containing build logs and metadata** of over 1 321 000 package builds, and more than 205 000 occurrences of non-reproducibility with full artifacts including 166 523 "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 beyond what we have done, or applying more complex heuristics or qualitative research to the diffoscopes.

Extended version This paper is an extended version of a previous paper [28] published at the Mining Software Repositories (MSR) 2025 conference. Compared to the shorter conference version, the current version includes: a longer time span (2015–2024 instead of July 2017–April 2023); more focus on the rebuildability aspect, with a new quantitative and qualitative analysis of rebuild failures, and the analysis of the cascading impact of rebuildability failures on the package dependency graph; new figures throughout the paper, including when discussing bitwise reproducibility. The conference paper contained an additional analysis of commits fixing unreproducibility, which we removed from the current version, because we do not consider that extending it to the new time span would bring significant additional insights. We refer the interested reader to the conference version for that analysis.

Paper structure This paper is structured as follows. After presenting the related work in Section 2, we give some background that is required to understand the experiment in Section 3, whose methodology is then presented in Section 4. Some descriptive statistics about the dataset are presented in Section 5, then we answer RQ1 about rebuildability in Section 6, and RQ2 about bitwise reproducibility in Section 7. We discuss the implications of our findings in Section 8, and the threats to validity in Section 9, concluding in Section 10.

2 Related work

2.1 Software reproducibility

Hinsen introduced the term *software collapse* [19], to capture the phenomenon that software stops working eventually if not actively maintained, impacting negatively the reproducibility of experiments in computational sciences. Among the many causes of software collapse, the inability to *rebuild* it is a major culprit. Software reproducibility [21]—meaning the ability to reproduce it, the software, rather than the fact that software always behaves the same—helps in reducing the risk of software collapse. In this paper we study extensively two possible technical measures of software reproducibility—rebuildability and bitwise reproducibility—in the Nix package ecosystem.

In 2020, the ReScienceC journal initiated the Ten Years Reproducibility Challenge [47, 39], encouraging researchers to rerun—and hence, in most cases, *rebuild*—software associated with their own publications from at least a decade prior. While participants generally succeeded, many authors reported significant challenges in the process.

Currently, supporting reproducible science through software tooling is a prominent topic, with researchers investigating the potential of Docker for this purpose [4, 7], alongside functional package managers [53, 11]. One of the key promises of functional package management [15, 9] is the comprehensive description of build environments, effectively mitigating dependency issues that lead to build and test failures. This represents the theoretical framework; however, it has been empirically validated only in experiments of limited scale [29]. With this work we extend the scale of similar experiments, both in terms of scale and of considered reproducibility dimensions: both rebuildability and bitwise reproducibility.

2.2 Reproducibility of build environments and software rebuildability

The presence of moving parts in build environments has long been acknowledged as problematic by developers, leading to issues with non-reproducibility. A substantial yet fragmented body of research has explored the causes and impacts of specific moving parts of development and build environments.

CI failures Continuous Integration (CI) [17, 32], an essential component of DevOps practices, relies on the stability of build environments. Empirical studies on CI build failures have been conducted at a large scale [49, 42, 56]. It has been firmly established that, regardless of the programming language, the primary cause of CI build failures is often attributed to (bloated) dependency issues [52].

Dependency hell More broadly, the phenomenon known as "dependency hell" [13] significantly contributes to the non-reproducibility of development environments. For instance, Mukherjee et al. [33] examined this issue within the Python ecosystem, while Zampetti et al. [55] corroborated similar findings in a wider analysis of "CI smells." Abate et al. [1] demonstrated that failures in dependency resolution are a recurrent cause of package non-installability across software ecosystems.

Flaky tests Flaky tests [24, 38] also frustrate developers and are among the main causes of rebuild failures identified in our study (Section 6.2). The categories in our taxonomy of rebuild failures map to the root causes identified by Luo *et al.* [24] and consolidated by Parry *et al.* [38]. In particular, our *race conditions* correspond to their Concurrency category; our *kernel differences* and *hardware differences* correspond to their Platform Dependency category; and our *date-related* permanent failures correspond to their Time category.

Software preservation Ensuring reproducibility in build environments is also crucial for software preservation [51]. Previous research [30, 48] has suggested that build systems, recipes, and tools should be preserved just as diligently as source code [8] and binary executables. Nevertheless, preservation is rendered less meaningful if, after capturing all the necessary components, the build process cannot be executed to produce the intended results.

2.3 Reproducible builds (R-B)

R-B, which we refer to as bitwise reproducibility in this paper (reusing a terminology already found in the literature [6]), 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 [46] 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.

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 [22]. 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 [54]. 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.

Techniques for bitwise reproducibility In a series of articles [43, 44, 45], 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 bitwise reproducibility is proposed by Navarro *et al.* [34]. They propose "reproducible containers" that are built in a way that makes the build process fully deterministic, at the expense of performance.

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. These alternative reproducibility definitions are stronger than simple rebuildability, but weaker than the strict notion of bitwise reproducibility. For instance, *accountable builds* [40] aim to distinguish between differences that can be explained (accountable differences) or not (unaccountable differences). Sharma et al. [50] takes a similar approach and "canonicalize" Java build artifacts so that it is possible to determine their equivalence even in the presence of meaningless (from the Java bytecode point

of view) changes at the byte level. OSS-Rebuild[2] is an initiative announced by Google in July 2025, which also takes a similar approach: they will rebuild packages from several ecosystems (PyPI, npm, Crates.io) at large scale, record build attestations using the SLSA framework,[3] and allow to compare independently built artifacts with a "semantic" comparison that ignores irrelevant byte-level 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.

Empirical studies of R-B Some other recent academic works have empirically studied R-B in the wild. Two papers from 2023 [6, 18] 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. Although we derive our heuristics from empirical data rather than from pre-labeled entries in the Debian issue tracker, they match causes cataloged in existing taxonomies: our *date heuristic* corresponds to the Build Timestamp category described by Lamb and Zacchiroli [22] and by Bajaj *et al.*; our *build ID heuristic* corresponds to Bajaj *et al.*'s Build ID category; and our *uname heuristic* matches their Architecture Information category. Our contribution is to quantify the prevalence of these causes at scale across the entire nixpkgs distribution over time.

A few recent works perform experimental studies of R-B by rebuilding software, like we do, but with different methodologies. Randrianaina *et al.* [41] study the effect of compile-time configuration options on reproducibility in three C systems (Linux kernel, BusyBox, ToyBox). They build each configuration twice on the same machine in a fixed environment, with no time gap, to isolate which options introduce non-determinism. Benedetti *et al.* [3] rebuild packages from six ecosystems (npm, Maven, PyPI, Go, RubyGems, Cargo) and compare two local builds against each other using `reprotest`, which varies environmental conditions (time, locale, file ordering) on the same machine. They

[2] `https://github.com/google/oss-rebuild`

[3] `https://slsa.dev/`

explicitly do not compare against published registry artifacts. Their study is cross-ecosystem but uses a single snapshot with no time gap between builds.

Sharma *et al.* [50] rebuild Java/Maven artifacts and compare against published artifacts on Maven Central, which were built on different infrastructure—a methodology closer to ours. However, they focus on a single ecosystem at a single point in time. In contrast, our study is longitudinal (spanning 2015–2024), covers an entire distribution including its full dependency graph, and compares local rebuilds with historical builds performed on different infrastructure and at significantly different times. Ours is a stronger reproducibility criterion, as it reveals failures caused by long-term environmental drift that same-time, same-machine approaches cannot detect.

2.4 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. This was the case of Benedetti *et al.* [3] as well, who were able to observe widely different R-B rates across ecosystems in their study. Similarly, we observe differences in bitwise reproducibility between ecosystems represented in nixpkgs, although for Python packages we observe a much higher reproducibility rate than they do, perhaps due to the way nixpkgs builds Python packages.

Beyond the specific topic of R-B, several previous works have compared package ecosystems (*e.g.,* [12]). For an overview of recent research on package ecosystems, see Mens and Decan [31].

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.* [23] 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.

3 Background

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

3.1 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

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

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

  src = fetchFromGitHub {
    owner = "htop-dev";
    repo = "htop";
    rev = version;
    sha256 = "sha256-MwtsvdPHcUdegsYj9NGyded5XJQxXri1IM1j4gef1Xk=";
  };

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

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

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.

3.2 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.

3.3 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.

3.4 The nixpkgs continuous integration

Hydra [16] 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.

3.5 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 pre-existing build output for the derivation under test, which it will try to fetch from one of the configured binary caches, or fail if not available. 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.[4]

3.6 Diffoscope

Diffoscope [14] 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 *diffoscopes*—that can be either interpreted by humans or automatically processed.

4 Methodology

Our build and analysis pipeline is summarized in Figure 2.

[4] 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.

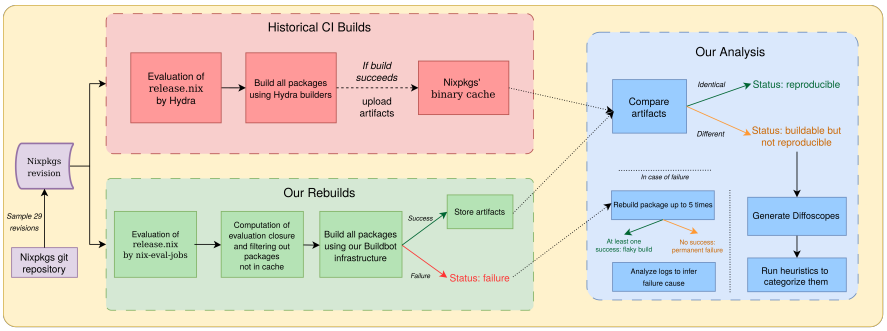

Fig. 2: Description of our build and analysis pipeline.

4.1 Reproducibility experiments

4.1.1 Revision sampling

We start from the 200 nixpkgs revisions from our ICSE NIER paper [29] 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. For the conference version of the current paper, 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 obtained a regularly spaced sample set of nixpkgs revisions, with one sampled revision every 4.1 months. To extend the time span of our study to the period 2015–2024, we added nixpkgs revisions starting from the two ends of the previous sample set, again picking one revision every 4.1 months, until we fully covered the new time span. This resulted in a total of 29 revisions, listed in Table 1. On average, each revision corresponds to building more than 46 thousand packages.

To perform our builds, we used a distributed infrastructure based on Buildbot [5], a Python CI pipeline framework. Our infrastructure has two types of machines: one *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 build jobs sent by the coordinator. During the initial part 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. We completed the experiment for the added revisions between January and October 2025 using bare-metal machines running NixOS, both for the coordinator and the builders. Note that to perform our bitwise reproducibility checks, we compare to the historical results coming from Hy-

Date	Commit Hash	Package Number
2015-03-13	e07ea5c	15729
2015-07-17	19e5fd6	15344
2015-11-11	225642e	17273
2016-03-15	fdcd875	19396
2016-07-17	62455c7	16486
2016-11-17	711a42e	16782
2017-03-21	02129a8	18281
2017-07-23	5328102	19631
2017-12-07	14df60d	22782
2018-04-14	4cfdced	23957
2018-08-31	1e1221c	31049
2019-01-03	b0f40b7	30575
2019-05-20	c00a043	28837
2019-09-24	a7e1051	32023
2020-02-09	eee784a	36794
2020-06-14	00e4481	38355
2020-10-29	38bff9a	41417
2021-03-05	f5f6dc0	45334
2021-07-21	1610540	46853
2021-11-22	98747f2	49919
2022-04-13	0343e34	53411
2022-08-16	6c6409e	57189
2022-12-20	ba6ba2b	76106
2023-04-27	8e3b64d	75584
2023-08-29	0c88788	81524
2023-12-31	d44d59d	90970
2024-05-02	94035b4	97783
2024-09-03	45508c1	112652
2024-12-27	7cc0bff	109717

Table 1: List of built revisions with commit hashes and number of packages. The rows highlighted in gray correspond to the revisions already built for the MSR 2025 conference version of this paper.

dra, that uses builders that, at the time, ran older versions of Nix than the ones we used on our builders.[5]

4.1.2 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

[5] Further details on the operating systems, Nix versions and kernel versions that we used on builders can be found in the replication package.

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. Derivations whose outputs are not present in the cache are excluded from our experiment, as there is no historical build output to serve as oracle for the bitwise reproducibility comparison.

4.1.3 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 3.5. 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 cache hence avoids incurring into issues such as source code disappearing from the original upstream distribution place. Investigating how much of nixpkgs can be rebuilt without relying on the cache for source code preservation is an interesting research question, recently explored for Guix [10], 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 store both the historical artifacts and our locally built ones, for comparison purposes.

4.1.4 Retrying failed builds

Transient build failures can happen due to various issues with our build infrastructure, such as network issues or a full disk. When such failures happen, we identify all the derivations that failed for a transient reason, and retry them later on, without considering the initial build result.

Next, we wish to analyze failed builds that are not due to transient issues on our system, including flaky builds that sometimes succeed and sometimes fail because of non-determinism in the build process. To that end, we retry all failed builds up to 5 times, and classify them as *flaky* as soon as they succeed once during the retries, and as *permanently failed* otherwise.

Note that with this approach, we necessarily miss some cases of flakiness if the first build was successful. However, detecting such cases would require building all packages multiple times, which was not feasible given the scale of

our experiment. Rebuilding only failed packages is viable, given that there are much fewer of them.[6]

4.2 Ecosystem identification and package tracking

To answer some of our research questions, 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 independently all the 197 ecosystems from the 29 nixpkgs revisions considered, ordered alphabetically, and deciding which ones to merge, 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).

4.3 Comparison with the minimal ISO image

As part of RQ2.2, we investigate the difference of bitwise reproducibility rate between critical packages whose reproducibility is monitored and the rest of the package set. We are interested in knowing whether observing the reproducibility rate of this subset of packages gives good enough insights on the overall reproducibility of nixpkgs. The minimal and GNOME-based ISO images are considered critical subsets of packages and benefit from a community-maintained reproducibility monitoring, which was initiated in 2019.[7] We study the minimal ISO image because it contains a limited amount of core packages.

[6] For the second oldest revision, 5119 packages fail to rebuild with the same `make` segfault (Section 6.1); as these all share a single root cause, we excluded these specific failures from the retry pipeline but retried the remaining non-segfault failures in that revision.

[7] The initial commit of the project github.com/grahamc/r13y.com (now superseded by github.com/NixOS/reproducible.nixos.org) tracking the reproducibility of the minimal ISO image is from February 2019.

We evaluate the Nix expression associated with the image, compute its install-time dependency closure (the set of packages included in the image) and match it with the packages of our dataset to infer their reproducibility status.

4.4 Analyzing causes of failed builds from build logs

To answer RQ1.2, we manually compare the build logs from failed and successful builds. Before manual inspection we run CiDiff [20], a tool for diffing continuous integration logs, on each pair of logs, to abstract over meaningless differences (e.g., timestamps, ordering) and help identify more quickly relevant differences between them. Because this is a manual analysis, we perform it on a smaller sample of nixpkgs revisions. Once again, we proceed with a dichotomic approach, starting from the oldest and the most recent revision, then picking the one in the middle of them, and so on. We stopped after having manually analyzed all the failed builds of 5 revisions, which corresponds to 373 failed builds.

For each revision, we analyze causes of flaky builds and causes of permanent rebuild failures separately. For permanent failures, we compare the local failed build log with the historical successful log from the cache. For flaky builds, we compare the log of the initial failed local build with the one of the first successful local build. We prefer local logs over historical logs in this case, because it guarantees that we always have the log available, even when the historical one is missing from the cache. When CiDiff fails to produce a diff of logs, we compare the two logs manually. In the case of permanent rebuild failures for which the log of the historical successful build is not available from the cache, we analyze the failed log only.

The first and last author performed this task as follows. They decided on the categorization by initially looking at diffed logs together. Once it was clear that the categorizations were solid for the logs encountered, they divided the work between them, but skipping any build where the cause would be ambiguous or would correspond to a new category, later reconvening to treat these cases and reach a consensus on the cause or the introduction of a new category.

To assess the reliability of this classification, both authors independently reclassified a random sample of 10% of the manually classified failures, yielding a Cohen's Kappa of 1.0 for the failure phase and 0.84 for the failure cause — a lower bound on the original classification's reliability, since ambiguous cases were resolved jointly during the original process.

4.5 Analyzing causes of bitwise unreproducibility using diffoscopes

Analyzing causes of bitwise 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 [44], but they rely on instrumenting the build, while we have to run the Nix builds unchanged to avoid introducing bias.

For each unreproducible output, we run diffoscope with a 5-minute timeout, yielding a dataset of 166 523 diffoscopes, corresponding to 165 976 packages[8] out of a total of 205 000 non-reproducible packages. 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 consider 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 reuse and rerun on the full diffoscope dataset the heuristics that we derived in the conference version of this paper [28] (from a smaller dataset of 86 476 diffoscopes). We refer the interested reader to the conference version for details on how the heuristics were obtained and validated.

5 Dataset

The main result of running the pipeline of Figure 2 is a large-scale dataset of historical package rebuilds, including (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 1 321 753 package builds coming from 29 nixpkgs revisions, built over a total of 21 480 hours. From those builds, 951 910 are coming directly from the `release.nix` file and can be tracked by name. They correspond to 84 315 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 10 045 in 2015 to 67 492 in 2024. Ecosystems can be present in multiple versions. On average, deduplicating packages in multiple ecosystem copies decreases their number by 20% while adding the evaluation closure of the `release.nix` file increases the number of jobs by 39%.

Figure 4 shows the evolution of the top 9 ecosystem sizes in nixpkgs, plus the base namespace. 63.1% of packages belong to an ecosystem, while the rest live in nixpkgs base namespace. The four largest ecosystems are Haskell, Python, Emacs, and Perl, which together account for 45.4% 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 2015 the deduplicated package set contained 6948 elements, 3263 of which were still present in 2024.

[8] In some cases, a package may have multiple unreproducible outputs.

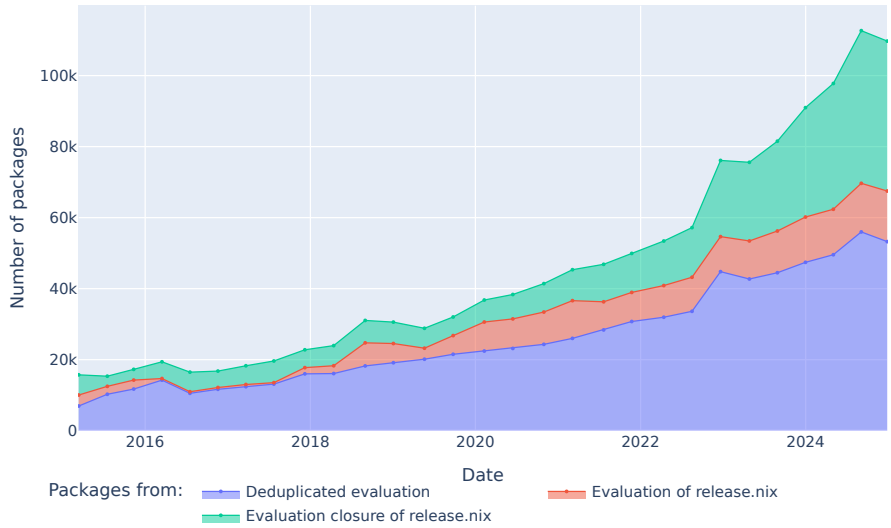

Fig. 3: Evolution of the number of packages in each nixpkgs revision (as defined by `release.nix`), in their evaluation closure and after deduplicating ecosystem copies.

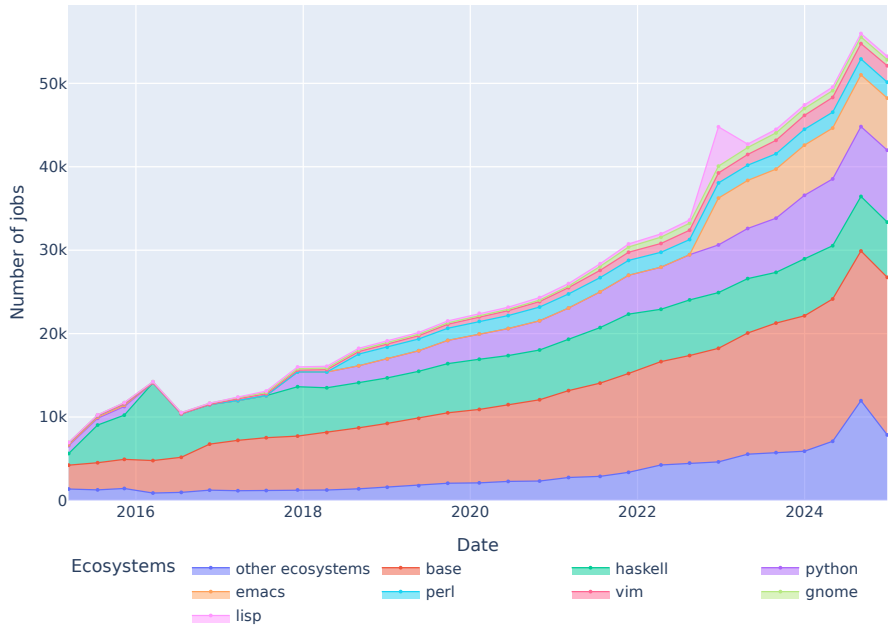

Fig. 4: Evolution of the size of the nine most popular software ecosystems in nixpkgs, the packages whose ecosystem is undetermined (base), and the packages from other ecosystems (only packages listed in `release.nix`).

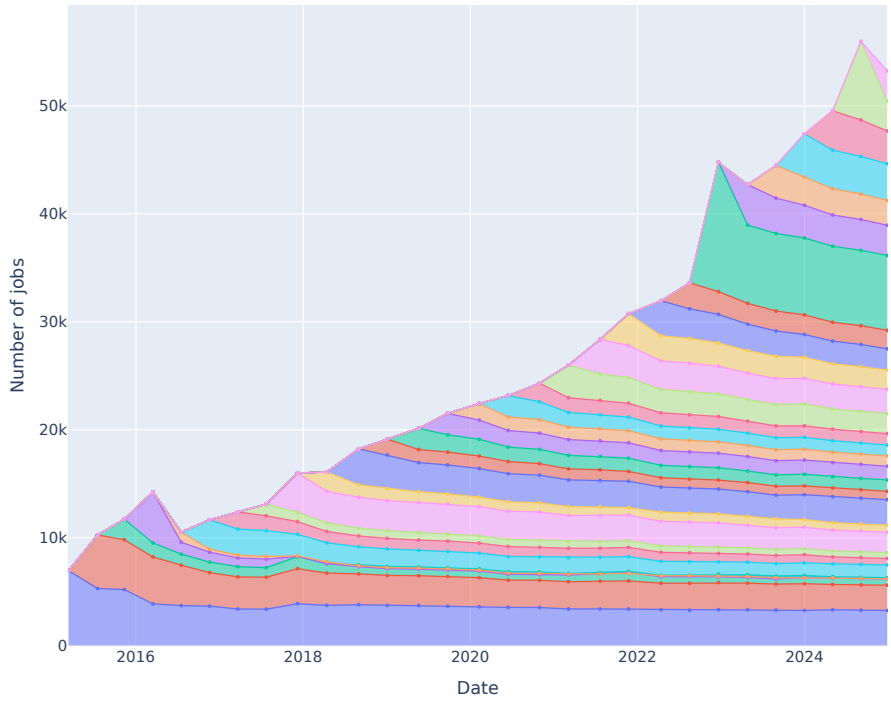

Fig. 5: Number of packages introduced by every revision of the dataset and their survival in the package set over time.

6 Rebuildability results

In this section, we present the results for our first research question: Does functional package management allow rebuilding past packages reliably at scale (even if not bitwise reproducibly)?

6.1 RQ1.1: What is the evolution of rebuildable packages in nixpkgs between 2015 and 2024?

Apart from a significant regression in 2015 (second revision, where reproducibility rate is of 67.24%), we obtain rebuildability levels between 99.34% and 99.95% (see Figure 6 and Figure 7), confirming the findings from our ICSE NIER paper [29]: Nix reproducibility of build environments allows for very high rebuildability rate over time. Note that for the revision already built in the ICSE NIER paper, only packages explicitly listed in `release.nix` were rebuilt, but now we also include those in the dependency graph.

The oldest revisions have more rebuild issues, and we observe in particular a major regression in the second oldest revision, which we investigate specifically. In that revision, 5119 packages fail to rebuild with the same exact error:

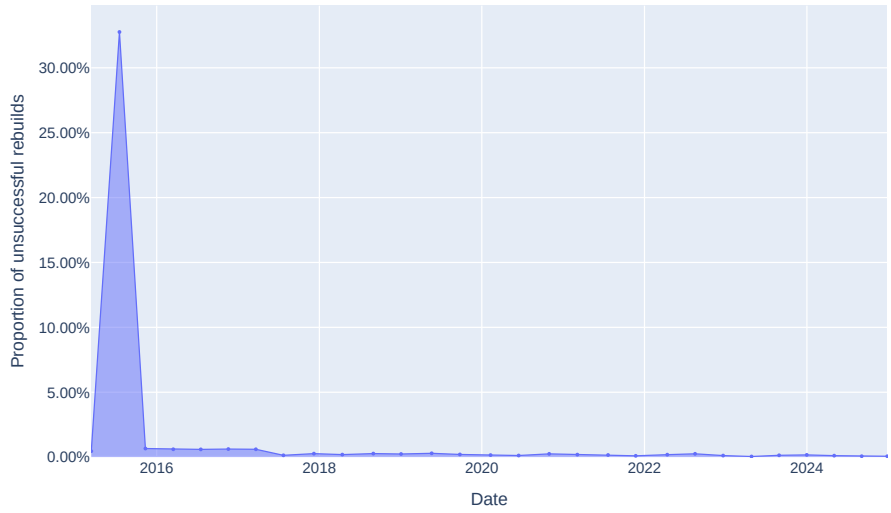

Fig. 6: Evolution of the proportion of rebuildability failures over time.

a segmentation fault in the version of `make` used in the nixpkgs standard build environment. Note that those packages built successfully in the nixpkgs continuous integration platform in 2015, which means that this segmentation fault was not happening at the time with that version of `make` and that a change exterior to the software in the build environment is the cause of this issue. Given that this error only happened for our second oldest revision means that the underlying issue was fixed between this revision and the next, despite the absence of symptom at the time (given that this is a rebuildability issue that we can only observe now). In order to understand the issue at play, we bisected the nixpkgs repository between revisions 2 and 3 to identify the commit that fixed the incorrect behavior of `make`. This operation yielded commit `3ede0a6` as a result, which applies a patch to `make` to prevent it from crashing when `ttyname` could not be read. This suggests a change in the build sandbox of Nix between the version used to perform the historical build (in which `ttyname` could be read) and the version used in our pipeline (in which `ttyname` could not be read, hence triggering the build failure).

Figure 8 breaks down rebuildability by ecosystem, excluding the second oldest revision.

6.2 RQ1.2: What are the causes of rebuild failures?

When discriminating rebuild failures between those whose subsequent rebuild attempts succeeded (*flaky* failures) and the others, we obtain that 1051 of the 8276 classified rebuild failures (12.7%) are flaky. Excluding the large rebuildability regression from the second oldest revision, the proportion of flaky

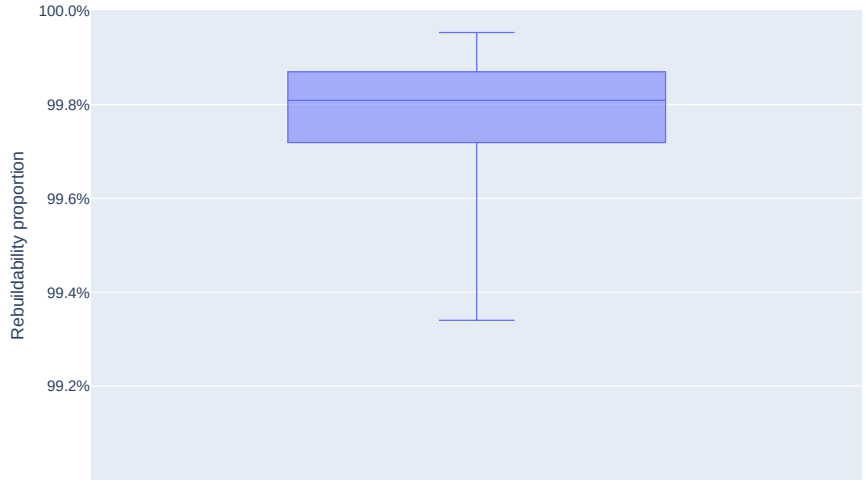

Fig. 7: Distribution of rebuildability rates in all revisions, to the exception of the second oldest revision. The box plot highlights, from bottom to top, the minimum, first quartile, median, third quartile and maximum rebuildability rates of the 28 considered revisions.

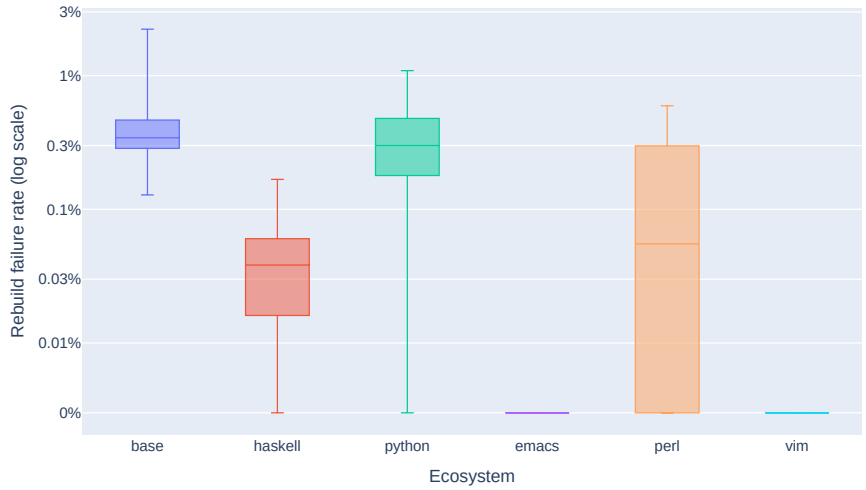

Fig. 8: Distribution of rebuild failure rates per ecosystem, excluding the second oldest revision. The y-axis is on a logarithmic scale.

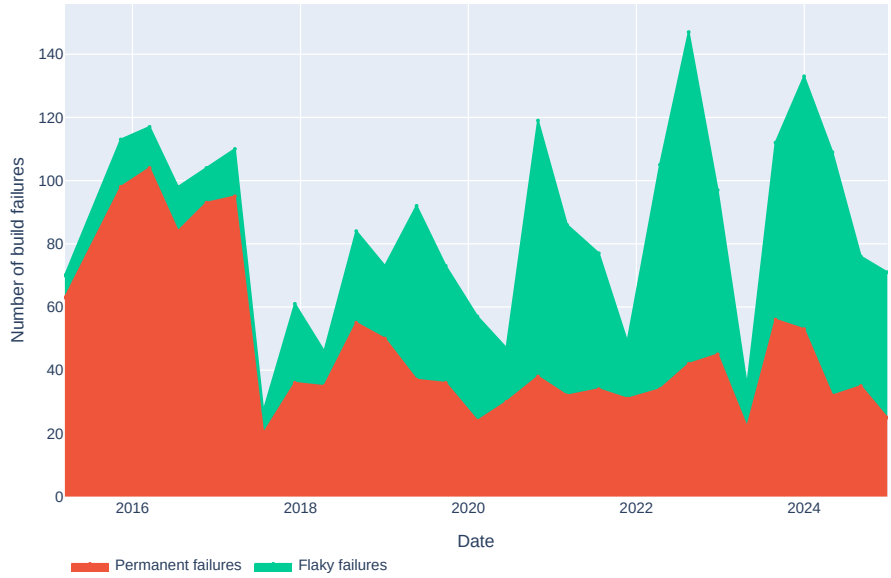

Fig. 9: Evolution over time of the absolute number of flaky and permanent rebuild failures across all revisions excluding the second oldest.

failures rises to 36.2% (1049 out of 2897). The detailed evolution of flaky and permanent rebuild failures is shown in Figure 9.

Flaky rebuild failures Flaky rebuild failures refer to rebuilds that fail initially but succeed upon being retried on the same machine, without any modification to the source code or build environment. Such behavior indicates either the presence of nondeterminism in the build process or insufficient resilience to transient, non-critical environmental fluctuations on the build host (like machine load).

One common manifestation of this phenomenon involves timeouts during test execution, typically caused by temporary resource contention on the build machine. For example:

```
Data generation is extremely slow: Only produced 1 valid examples in 1.86
seconds (0 invalid ones and 0 exceeded maximum size). Try decreasing size
of the data you're generating (with e.g. max_size or max_leaves parameters).
```

This case belongs to the broader category of *flaky tests*—failures occurring during the test phase when one or more individual tests fail in one run but pass in subsequent retries. For instance, in a failed build of the package `buildbot`, the log reports:

```
Ran 5673 tests in 66.450s
FAILED (skips=37, errors=4, successes=5635)
```

However, upon the first rebuild of the same package, the corresponding log reads:

```
Ran 5673 tests in 153.387s
PASSED (skips=37, successes=5636)
```

Flaky tests are the predominant cause of flaky rebuild failures: we identified them in 135 out of the 194 flaky failures analyzed.

Flakiness can also occur during the build or installation phases—where artifacts are compiled or copied into their final filesystem locations. We observed 59 such cases. Two main causes emerged from our analysis: *race conditions* and *kernel version differences*.

Race conditions arise when the ordering of build tasks affects the outcome. For example, a task `B` may depend on task `A` but the dependency is not properly declared in the build recipe. As a result, the build may succeed or fail depending on external factors such as CPU concurrency or system load. In one build of `elfkickers`, the process attempted to start a task requiring the object file `elfrw_sym.o` before it was compiled, resulting in the following error:

```
ar: elfrw_sym.o: No such file or directory
```

We identified 19 such cases.

Finally, discrepancies in the Linux kernel version can also induce nondeterministic build behavior. For example, we observed failures in 23 Lisp packages built with `SBCL`:

```
mmap: wanted 5308416 bytes at 0x1003618000, actually mapped at 0x7ffff6000000
```

In this case, the behavior of the `mmap` system call differs from `SBCL`'s expectations regarding the memory mapping address, leading to intermittent rebuild failures. These results illustrate how even low-level system variations can create nondeterministic effects that eventually impact build reproducibility.

As Figure 10 shows, a large majority (866 out of 1051, 82.4%) of flaky rebuilds succeed on the first retry, and only 11 (1.0%) require all five attempts before succeeding.

Figure 11 shows the evolution of flaky rebuild failure causes across the 5 manually studied revisions. Flaky tests remain the dominant cause throughout, while race conditions and kernel-related flakiness appear more sporadically. Figure 12 breaks down the flaky rebuild failure causes by ecosystem. Haskell and Python concentrate the majority of flaky tests, which is consistent with both ecosystems relying heavily on test suites during the build process. Kernel-related flakiness is most visible in the base nixpkgs namespace and in the Lisp ecosystem, reflecting the SBCL memory mapping issue discussed above.

Permanent rebuild failures Permanent rebuild failures refer to packages that could not be built successfully even after five consecutive retry attempts. However, these packages had built successfully in the past on the Nix project's continuous integration platform. This suggests that the cause of failure lies in changes external to the build environment (given that the build environment itself is fully specified and reproduced by Nix).

We observed that the majority of these failures (107 out of 179) occurred during the test phase, where arbitrary code is executed to verify that the built

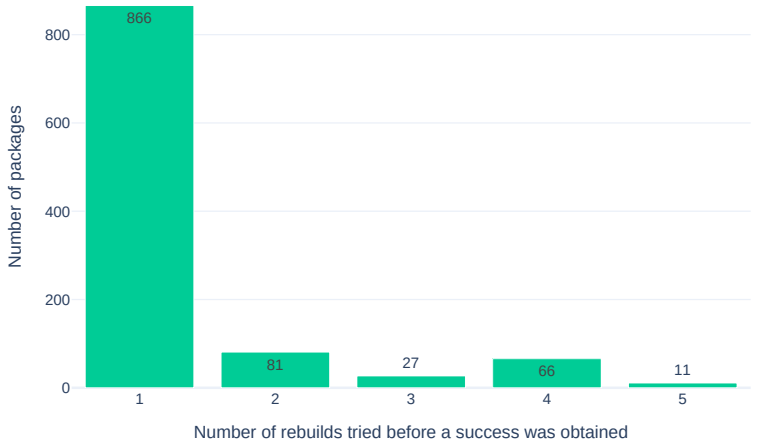

Fig. 10: Distribution of the number of rebuild attempts needed before a successful rebuild, across all 1051 flaky rebuild failures in our dataset.

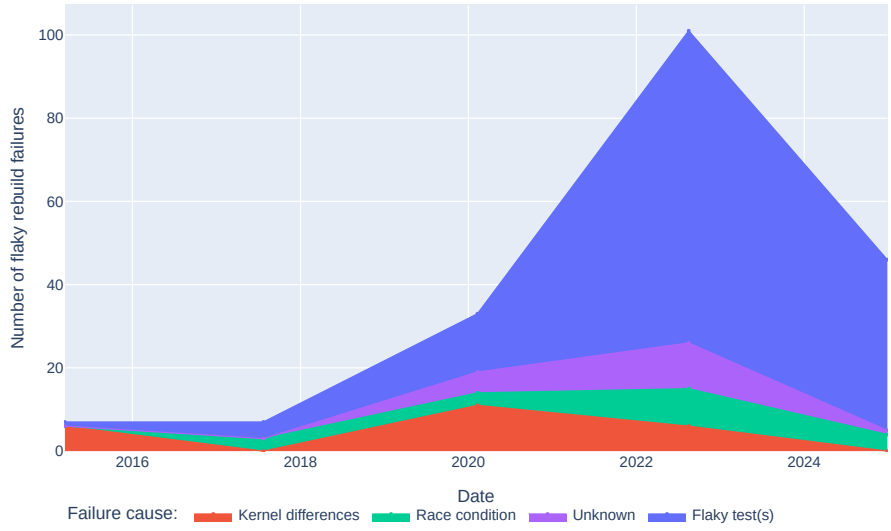

Fig. 11: Evolution over time of the causes of flaky rebuild failures in the 5 manually studied revisions.

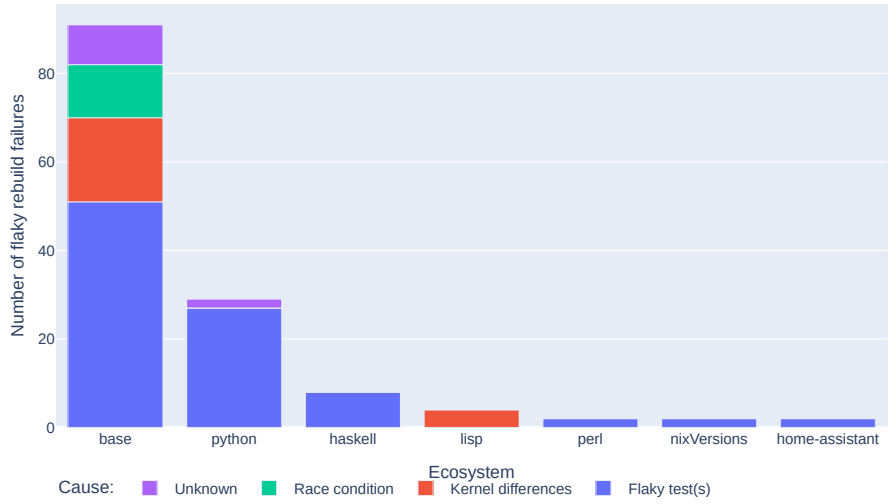

Fig. 12: Causes of flaky rebuild failures broken down by ecosystem (aggregated across the 5 manually studied revisions), over ecosystems with at least two flaky failures.

artifacts behave as expected. Tests in this phase often rely on implicit assumptions about the execution environment. One common assumption, identified in 44 cases, is the current date. For instance, some tests depend on certificates with fixed expiration dates, e.g.

```
Client certificate expired: Not After: 2021-11-27 15:28:58Z
```

Others verify output containing time-dependent information, such as copyright statements:

```
AssertionError: ‘‘Copyright (C) 2024, [retracted], Inc.’’ not found in ‘‘MIT
License Copyright (C) 2022, [retracted], Inc.’’
```

It is likely that such a test was crafted so that the maintainer would not forget to update the copyright year when releasing a new version of the software, but was not meant to be run in builds of distributions shipping the package.

Two other major causes of permanent rebuild failures that we could identify are kernel version differences and hardware differences.

Issues related to kernel version differences arise in both the test phase (9 cases) and the build phase (15 cases). The kernel version used in our build pipeline most often differed from that of the historical builds. In the test phase, this can lead to tests failing when they rely on specific kernel features or system APIs that have changed over time. For example, in one build of the `go` compiler, the test failed with the following error message:

```
creds_test.go:60: WriteMsgUnix failed with write unix @: invalid argument,
want EPERM
```

In the build phase, kernel version discrepancies can lead to outright build failures when, for instance, the configuration scripts check for specific kernel versions that are known to be compatible. An example of this is:

```
configure: error: Valgrind works on kernels 2.4, 2.6
```

Issues arising from hardware differences were predominantly observed during the test phase (14 cases), although they can also affect the build phase (4 cases). While our pipeline used the same CPU architecture as the historical builds (`x86_64-linux`), subtle variations in specific processor features (such as the microarchitecture) can lead to discrepancies, especially during the test phase where code is executed directly. For instance, we encountered the following assertion failure:

```
AssertionError: assert np.float64(0.9272952180016122) == 0.9272952180016123
```

This assertion is the result of a floating-point computation that yields a slightly different result than the one expected by the test, possibly due to variations in floating-point rounding (e.g., due to a library which would run slightly different code paths depending on the available CPU instructions). Because we used a range of different CPU models in our build infrastructure, we were able to find an older machine where this test passed, despite the fact that it consistently failed on newer hardware.

Conversely, we also found cases where newer hardware features were required, leading to failures on older CPU models. For example, in builds of the `jmp` and `chex` packages, which depend on the `jaxlib` package, tests failed with the following error message:

```
RuntimeError: This version of jaxlib was built using AVX instructions, which
your CPU and/or operating system do not support. You may be able work around
this issue by building jaxlib from source.
```

The likely explanation is that the software in the build environment (including `jaxlib`), which we downloaded from the Nix cache, were built on machines with a more recent CPU model than the one used in our build infrastructure for this specific build. We were able to confirm that this test passed successfully on a newer machine in our infrastructure, despite the consistent failure on older hardware. Note that this later example of rebuild failure would not have happened if we had built all dependencies from source instead of downloading them from the Nix cache, as the build process would (expectedly) have adapted to the CPU features of our build machine.

One notable example of failure due to hardware differences in the build phase is from a build process which fails when started with an excessive number of CPU cores:

```
Invalid value for the '-j' option, valid values are 1 through 64.
```

Overall, these cases highlight that even seemingly minor variations in the external environment—such as kernel or hardware configuration—can deterministically break previously successful builds, emphasizing the fragility of rebuildability in time.

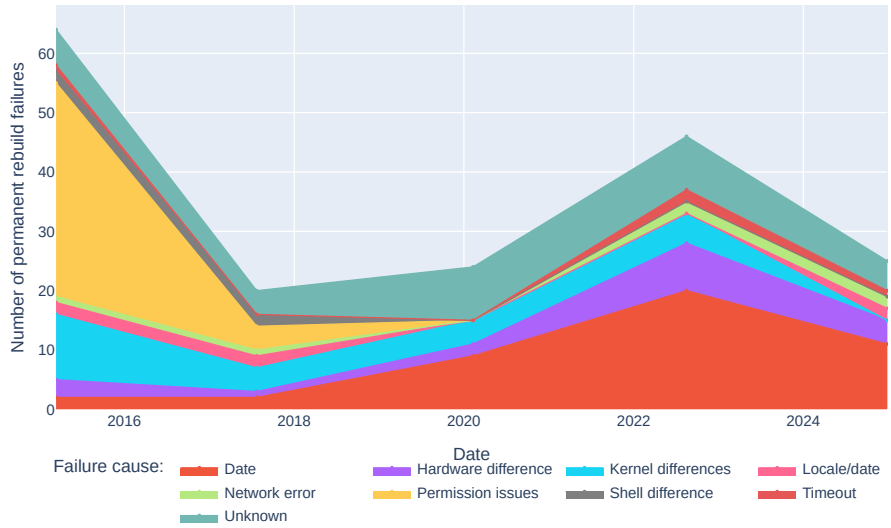

Fig. 13: Evolution over time of the causes of permanent rebuild failures in the 5 manually studied revisions.

Figure 13 shows how the causes of permanent rebuild failures evolve across the 5 studied revisions. Date-related failures (44 cases) and permission issues (40 cases) are the two most prevalent causes. Kernel and hardware differences appear in all studied revisions. Some causes, such as locale/date, permission issues and shell differences, only appear in the first two manually studied revisions, which suggests that these permanent rebuild failures are related to changes in the Nix sandbox between these old data points and today. Figure 14 breaks down the same causes by ecosystem.

6.3 RQ1.3: What is the cascading impact of rebuildability failures on the package dependency graph?

Our analysis pipeline has rebuilt each package independently and in parallel from other packages in the same revision. We relied on our ability to precisely reproduce the build environment of a given package by downloading all the dependencies from the Nix cache. The results provided in the previous sections therefore do not account for the cascading effects of rebuildability failures.

We revisit the rebuildability figures provided in Section 6.1 by exploring the package dependency graph. Note that this analysis concerns *rebuildability* (not bitwise reproducibility): we ask whether a package could be rebuilt at all today if its dependencies also had to be rebuilt from source, rather than fetched (prebuilt) from the binary cache. This analysis is performed by combining the per-package build results from our pipeline with a traversal of the dependency graph: for each package, we check whether all of its transitive

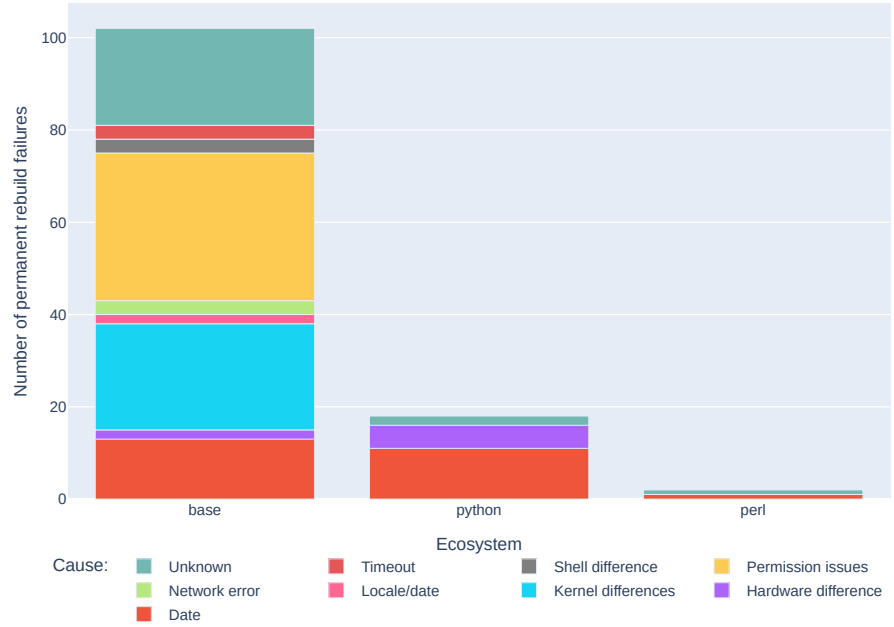

Fig. 14: Causes of permanent rebuild failures broken down by ecosystem (aggregated across the 5 manually studied revisions), over ecosystems with at least two permanent failures.

dependencies were successfully rebuilt. A rebuildability failure anywhere in a package's transitive dependency chain prevents rebuilding the package: without the binary cache, we cannot reproduce the build environment of a package whose dependency failed to build.

Accounting for cascading rebuildability failures, rebuildability rates decrease sharply, see Figure 15. Besides, in Figure 16 we observe the evolution of this rebuildability rate to be irregular, with 11 revisions with more than 90% packages still rebuildable, interleaved with 18 revisions with, on average, 35.6% cascading rebuildability failures. This disparity can be explained by the fact that small changes may have large effects inside the dependency graph, as shown by Figure 17: 14 failures of 8 individual packages out of 7216 rebuild failures account for 54.0% of the cascading rebuildability failures.

Table 2 lists these packages and the number of dependents impacted by their failure, distinguishing between cascading failures caused by flaky root causes (which could be resolved by retrying the build) and permanent root causes. The most impactful package, `dejagnu`, is a testing framework used transitively by a large portion of the package set; and its build failure is flaky. Overall, 54.3% of all cascading rebuildability failures are caused by flaky root causes, meaning that more than half of the cascading impact could be mitigated by simply retrying failed builds.

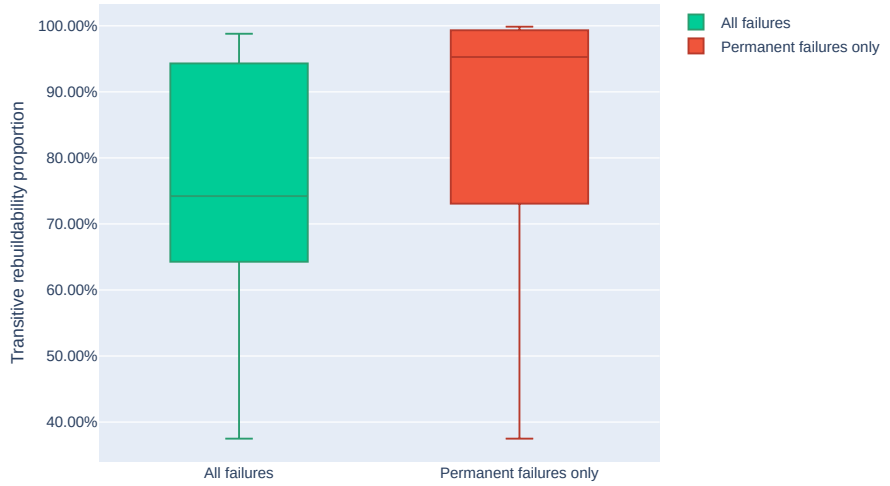

Fig. 15: Distribution of transitive rebuildability rates across the 29 revisions, when accounting for all rebuild failures (left) versus only permanent failures (right).

Fig. 16: Evolution over time of the proportion of unsuccessful rebuilds, distinguishing direct rebuild failures, cascading failures caused by permanent root causes, and cascading failures caused by flaky root causes.

Package	Version	Impacted dependents	Failure type
`dejagnu`	1.6.3	51 405	Flaky
	1.6.3	20 183	Flaky
`libuv`	1.44.2	17 684	Flaky
	1.28.0	5056	Flaky
`coreutils`	8.27	11 446	Permanent
`psutil`	5.7.2	9459	Flaky
`gnutls`	3.6.15	8274	Flaky
	3.6.13	8615	Permanent
`pyOpenSSL`	19.0.0	6887	Permanent
	19.0.0	6334	Permanent
	18.0.0	5462	Permanent
	17.2.0	6716	Permanent
`heimdal`	1.5.3	5640	Permanent
`ghc`	8.4.3	5383	Flaky

Table 2: Individual rebuildability failures that each cascade to more than 5000 dependents. The same package can appear multiple times across different revisions or versions. The "Failure type" column indicates whether the failure was flaky (succeeded on retry) or permanent.

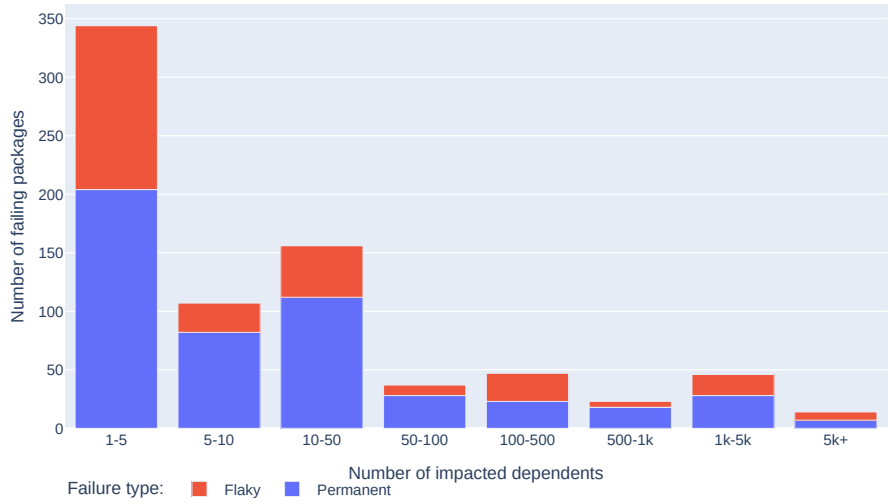

Fig. 17: Distribution of the number of impacted dependent packages for each rebuildability failure, distinguishing between flaky and permanent root causes.

Figure 18 plots the same data on a logarithmic scale, comparing all failures with permanent failures only. The heavy-tailed nature of cascading effects is visible in both cases, but the permanent-only distribution is noticeably sparser at higher impact levels, confirming that many high-impact cascading failures are avoidable through retries.

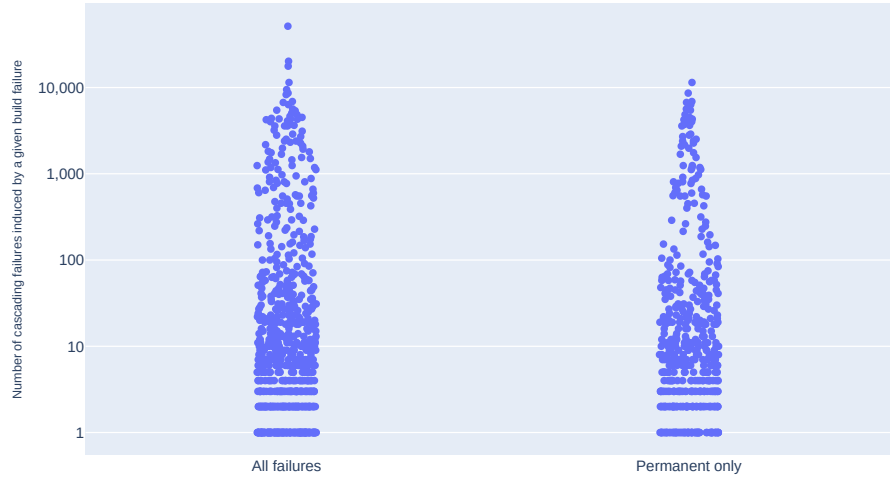

Fig. 18: Number of dependent packages impacted by each rebuild failure that cascades to at least one dependent, on a logarithmic scale. Left: all failures. Right: permanent failures only.

7 Bitwise reproducibility results

In this section, we present the results for our second research question: Does functional package management allow bitwise reproducibility at scale?

7.1 RQ2.1: What is the evolution of bitwise reproducible packages in nixpkgs between 2015 and 2024?

Our results show a major improvement of bitwise reproducibility in nixpkgs over time, from 35% in 2015 to 93% in 2024 (excluding the rebuildability regression in the second oldest revision, see Figure 19). The trends in Figure 20 show that the absolute number of bitwise reproducible packages has consistently gone up and followed the fast growth of the package set. The most notable 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 7.2 below.

Figure 21 shows for each revision the cumulative number of unreproducibilities introduced by that revision that get fixed over time. Each curve starts at zero at the revision where the unreproducibilities were introduced, and increases as those unreproducibilities are fixed in subsequent revisions. The visually dominant curve corresponds to the June 2020 reproducibility regression (reaching approximately 3000 fixes). Most other revisions introduce far fewer unreproducibilities, and their curves remain near the bottom of the figure. Across all revisions, the large slope between the two first points of each

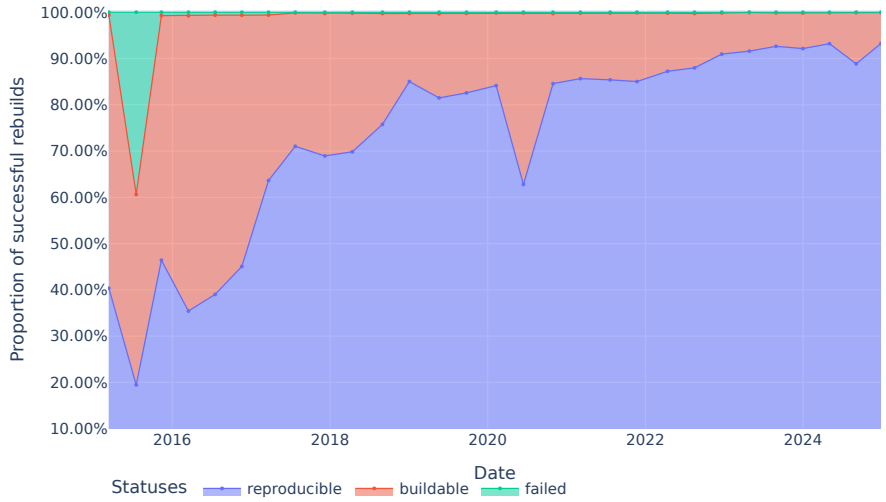

Fig. 19: Proportion of reproducible, rebuildable (but unreproducible) and non-rebuildable packages over time.

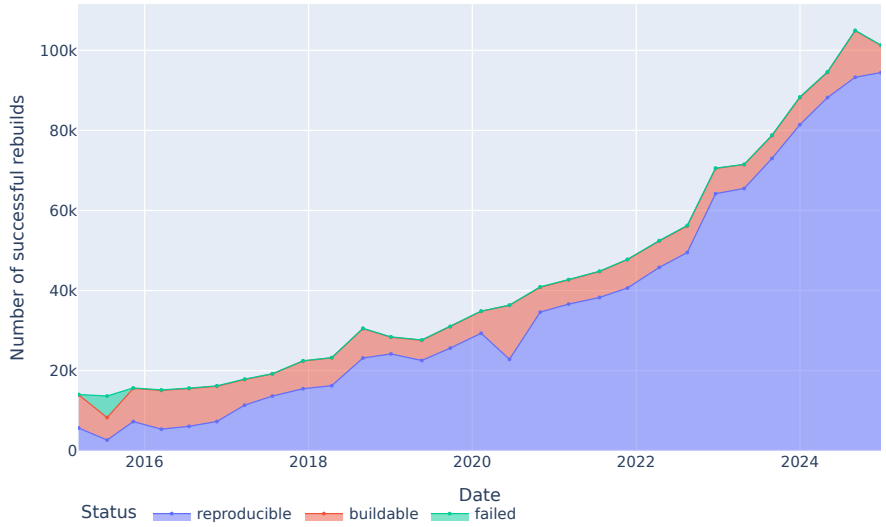

Fig. 20: Absolute numbers of reproducible, rebuildable (but unreproducible) and non-rebuildable packages over time.

curve indicates that most unreproducibilities introduced in a given revision are fixed in the next revision, on average 77% of them (59% when excluding the packages fixed after the 2020 reproducibility regression).

Excluding the June 2020 revision, Figure 22 depicts the average evolution of the package set between two consecutive revisions. In particular, only 0.53%

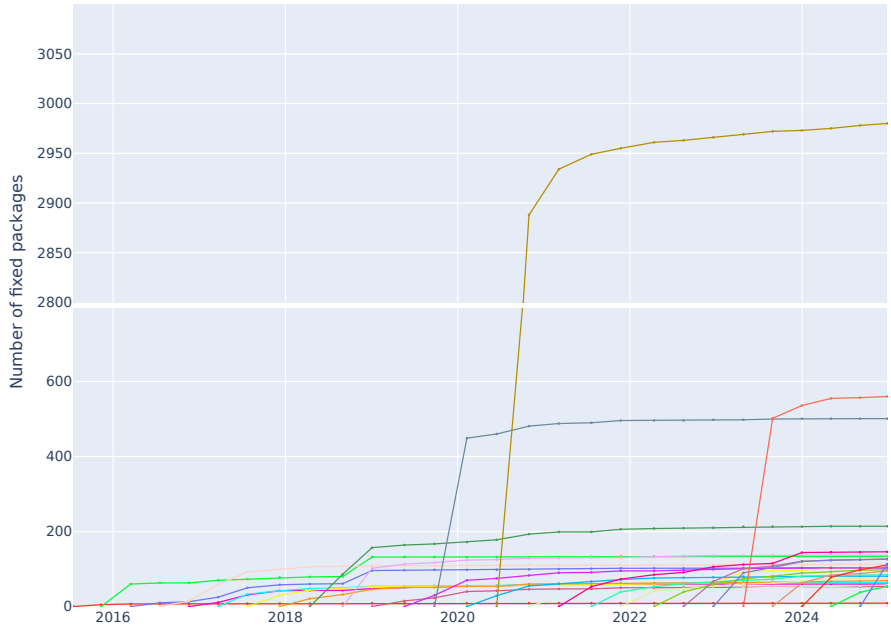

Fig. 21: Evolution of the cumulative number of fixes to non-reproducibilities, separated by the revision in which the non-reproducibilities were introduced.

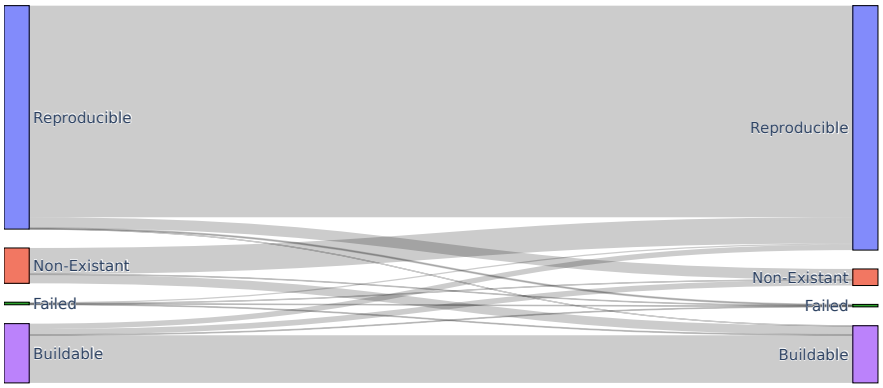

Fig. 22: Sankey graph of the average flow of packages between two revisions, excluding the revision from June 2020, considered as an outlier.

of the packages transition from *reproducible* to *buildable but not reproducible* status between two revisions, while 1.94% of the *buildable but not reproducible* packages become reproducible. The average growth rate of the package set is 1.06 and, on average, 73.90% of the newly introduced packages are reproducible.

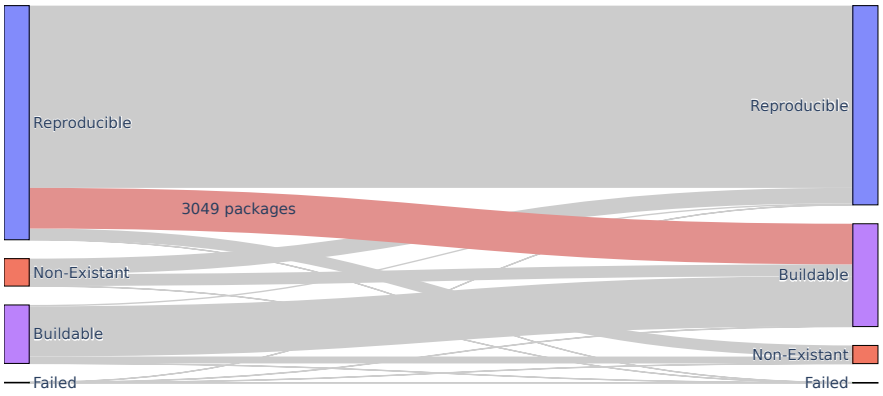

Fig. 23: Sankey graph of the flow of packages between the revision from June 2020 (bitwise reproducibility regression) and the next one.

Figure 23 contrasts this average behavior with the June 2020 bitwise reproducibility regression visible in Figure 19: unlike the typical inter-revision flow, this transition shows a large number of packages moving from *reproducible* to *buildable but not reproducible*, illustrating the magnitude of the regression. We investigate its root cause in Section 7.2.

7.2 RQ2.2: What are the bitwise unreproducible packages?

We observe large disparities both in trends and in reproducibility by package ecosystem (see Figure 24). While some ecosystems, like Perl, Emacs and Vim, have virtually always had all their packages reproducible, the Haskell ecosystem started very low (around 5% in 2015), significantly improved to around 60% in 2017, and then stagnated or even regressed slightly, down to around 48% in 2024. The Python ecosystem has globally enjoyed high reproducibility rates, to the exception of three revisions, and in particular one major regression in June 2020, dropping from 96% to 6% reproducibility, before recovering back to 96% bitwise reproducible in the next revision.

As can be seen in Figure 23, 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.[9]

To complement the per-ecosystem analysis, and understand where improvements to the overall reproducibility of nixpkgs could be focused, we analyze

[9] Details can be found in `https://github.com/pypa/pip/issues/7808`.

(a) nixpkgs base namespace

(b) Haskell

(c) Python

(d) Emacs

(e) Perl

(f) Vim

Fig. 24: Number of reproducible, buildable and failed packages in the five biggest ecosystems and the base namespace of nixpkgs. The holes in the plots correspond to revisions where no package from the ecosystem was present in the package set built by Hydra. This can be due to the absence of packages from that ecosystem in nixpkgs at that time, or to the absence of those packages from the `release.nix` file.

the ecosystems of provenance of unreproducible packages in our most recent revision (see Figure 25). We find that the Haskell ecosystem accounts for 70% of all bitwise unreproducible packages, which is in line with both its large size in nixpkgs and its low observed reproducibility rate. This low build reproducibility performance is an *upstream* issue in the Haskell compiler (GHC) itself, which does not produce deterministic object code, rather than a problem with the Nix infrastructure layer or the way Haskell packages are packaged in nixpkgs. A fix to this long-standing issue has been introduced in GHC 9.12.1 (the new `-fobject-determinism` flag, which is off by default, because while

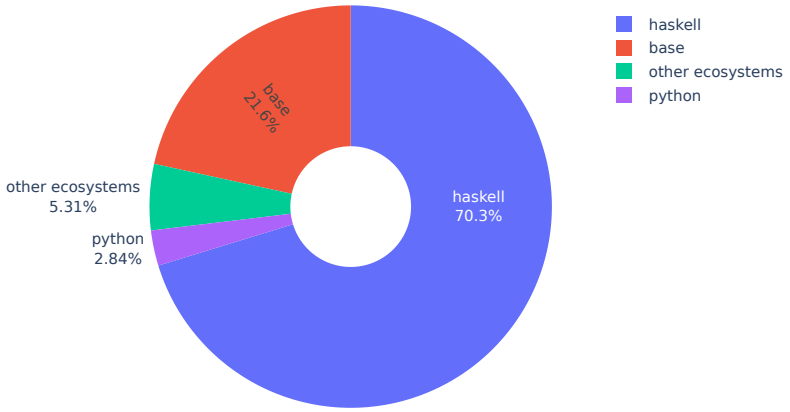

Fig. 25: Ecosystems of provenance of unreproducible packages in our most recent revision.

it improves reproducibility, it makes compilation slightly slower). As of October 2025, nixpkgs does not make use of this flag, but we have highlighted its existence to the nixpkgs maintainers monitoring bitwise reproducibility, and a preliminary experiment by a nixpkgs maintainer confirms that using the flag could improve the reproducibility of Haskell packages in nixpkgs.

Figure 26 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 last few years, with rates consistently higher than 95% starting from May 2019 (a few months after the monitoring began). 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.

7.3 RQ2.3: Why are packages unreproducible?

In the MSR 2025 conference version of this paper [28], 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

Fig. 26: 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 build artifacts (version file, log file, file names, etc.). To detect dates embedded in build outputs, we look specifically for the year of the build (2024 or 2025) in added lines, given that we never ran our builds during the same year as the historical builds (the revisions from 2024 were built in 2025).
- **`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 to achieve bitwise reproducibility, we find that 21 649 of our 165 976 packages with diffoscopes contain a date, accounting for 13.0% of them. Still in the most recent nixpkgs revision, we find 840 instances of this non-reproducibility cause, showing that embedding dates is still an existing practice. Embedded environment variables represent 4.3%

(a) Date heuristic.

(b) Uname heuristic

(c) Environment variable heuristic.

(d) Build ID heuristic.

Fig. 27: Evolution of the number of packages for which we generated diffoscopes that are matched by each of our heuristics, over time

of our unreproducible packages, although only 4 revisions account for the vast majority of instances, which suggests that this issue has more to do with the way packages are built in nixpkgs, rather than the packages themselves. Build IDs are found in 1.7% of our unreproducible packages. Finally, we find `uname` outputs in 1323 of our unreproducible builds (0.8%), 721 of which also include a date as part of the `uname` output. Figure 27 shows the evolution of the number of packages matched by each heuristic over time. Altogether, a total of 19.2% of the packages for which we have generated a diffoscope have an unreproducibility that can be explained by at least one of our heuristics.

8 Discussion

This work brings valuable insights to the ongoing discussions about software reproducibility, rebuildability, 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 counter proof of this belief: in nixpkgs, the largest general-purpose repository with more than 100k packages (in 2024), more than 93.2% 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 [22] 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 6.7% of nixpkgs packages are not bitwise reproducible in our last revision. We can even strengthen this claim with the additional revisions built further in the past in this extended version of our study: the lower proportion of bitwise reproducible packages in the early years of our dataset (as low as 35%) shows that FPMs alone are not enough to guarantee bitwise reproducibility. This is not surprising, because R-B violations happen for multiple reasons, only some of which are mitigated by FPM *per se*. The high bitwise reproducibility rates we observe in recent years are thus more likely due to the cumulative effect of multiple factors, including the adoption of R-B best practices in upstream toolchains, the fixes to reproducibility issues proposed by members of the R-B community (including from other distributions), and the efforts of the nixpkgs maintainers to monitor and fix reproducibility issues.

Recommendation: use FPM for bitwise reproducibility and rebuildability Still, Nix appears to perform really well at bitwise reproducibility, and even more so at rebuildability: our ability to consistently rebuild more than 99% of packages even after a decade is a great demonstration of Nix's abilities for software reproducibility (with the caveat that due to the criticality of some packages in the package dependency graph of nixpkgs, single failures may have ripple effects). This highlights the impact more adoption of FPMs could have on software engineering practices. Based on this, we recommend to use Nix, or FPMs more generally, for all use cases that require either mere rebuildability or full bitwise reproducibility. At worse, they provide a very good starting point.

QA monitoring for bitwise reproducibility The significantly higher (and stably so) bitwise reproducibility performances of the NixOS minimal ISO image (see Figure 26) 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 [22] 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).

9 Threats to validity

9.1 Construct validity

We rebuilt Nix packages from revisions in the period 2015–2024, for about 10 years. That is enough to observe arguably long-term trends, which also appear to be stable in our analysis; except for one notable regression in rebuildability, and one in bitwise reproducibility, 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 21 480 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? For the rebuildability question, we have studied this impact in Section 6.3, finding that rebuildability rates drop sharply when accounting for cascading rebuildability failures, although as highlighted in our analysis of rebuild failures (Section 6.2), this could often be mitigated by retrying the build (in case of flaky rebuild failures) or by disabling the test phase (in case of permanent rebuild failures occurring during tests). For bitwise reproducibility, it would be much more complicated to estimate the transitive impact of unreproducible dependencies, given that in most cases, packages that do not build bitwise reproducibly

should still behave the same (but we do not have a theoretical guarantee of that).

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 [8] in case of failure.

9.2 External validity

We have rebuilt packages from nixpkgs, the largest cross-ecosystem FOSS distribution, using the Nix functional package manager (FPM). Other FPMs with significant user bases exist, such as Guix [9]. 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.

10 Conclusion

In this work, we conducted the largest scale empirical study of bitwise reproducibility of open source software going back in time, in the context of the Nix functional package manager, by rebuilding 1 321 753 packages coming from 29 revisions of the nixpkgs software repository, sampled every 4.1 months from 2015 to 2024. Our findings show that bitwise reproducibility in nixpkgs is very high and has known an upward trend, from 35% in 2015 to 93% in 2024. The mere ability to rebuild packages (whether bitwise reproducibly or not) is even higher, stably around 99.8%.

We have highlighted disparities in bitwise 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 13% of unreproducible packages were embedding the date during the build process. Finally, we conducted an in-depth analysis of a subset of non-rebuildable packages and uncovered several recurrent failure causes such as test processes.

Declarations

Data availability statement

The dataset produced as part of this work will be archived on and available on Zenodo [25, 26] upon acceptance. A full replication package containing the code used to run the experiment describe in this paper, and a preview of the dataset is available as a git repository [27].

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.

Funding

No external funding was received for this study.

Conflict of Interest

The authors declare that they have no conflict of interest.

Author Contributions

Julien Malka conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.
Stefano Zacchiroli conceived and designed the experiments, authored or reviewed drafts of the article, and approved the final draft.
Théo Zimmermann conceived and designed the experiments, analyzed the data, authored or reviewed drafts of the article, and approved the final draft.

Ethical Approval

Not applicable.

Informed Consent

Not applicable.

Clinical Trial Number

Not applicable.

References

[1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. "Mining Component Repositories for Installability Issues". In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. Ed. by Massimiliano Di Penta, Martin Pinzger, and Romain Robbes. IEEE Computer Society, 2015, pp. 24–33. URL: https://doi.org/10.1109/MSR.2015.10.

[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] Giacomo Benedetti, Oreofe Solarin, Courtney Miller, Greg Tystahl, William Enck, Christian Kästner, Alexandros Kapravelos, Alessio Merlo, and Luca Verderame. "An Empirical Study on Reproducible Packaging in Open-Source Ecosystems". en. In: 2025.

[4] Carl Boettiger. "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 71–79. ISSN: 0163-5980. URL: https://doi.org/10.1145/2723872.2723882.

[5] *Buildbot*. URL: https://buildbot.net/ (visited on 10/24/2024).

[6] 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).

[7] Jürgen Cito and Harald C. Gall. "Using Docker Containers to Improve Reproducibility in Software Engineering Research". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 906–907. ISBN: 9781450342056. URL: https://doi.org/10.1145/2889160.2891057.

[8] 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.

[9] 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).

[10] 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 Replica-*

bility, ACM REP 2024, Rennes, France, June 18-20, 2024. ACM, 2024. URL: https://doi.org/10.1145/3641525.3663622.

[11] Ludovic Courtès and Ricardo Wurmus. "Reproducible and User-Controlled Software Environments in HPC with Guix". In: *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*. Ed. by Sascha Hunold et al. Vol. 9523. Lecture Notes in Computer Science. Springer, 2015, pp. 579–591. URL: https://doi.org/10.1007/978-3-319-27308-2%5C_47.

[12] 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).

[13] Stephanie Dick and Daniel Volmar. "DLL Hell: Software Dependencies, Failure, and the Maintenance of Microsoft Windows". In: *IEEE Annals of the History of Computing* 40.4 (Oct. 2018). Conference Name: IEEE Annals of the History of Computing, pp. 28–51. ISSN: 1934-1547.

[14] *diffoscope: in-depth comparison of files, archives, and directories*. URL: https://diffoscope.org/ (visited on 11/07/2024).

[15] Eelco Dolstra. "The purely functional software deployment model". en. OCLC: 71702886. PhD thesis. S.l.: s.n., 2006.

[16] 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.

[17] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. en. Google-Books-ID: PV9qfEdv9L0C. Pearson Education, June 2007. ISBN: 978-0-321-63014-8.

[18] 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).

[19] Konrad Hinsen. "Dealing With Software Collapse". In: *Comput. Sci. Eng.* 21.3 (2019), pp. 104–108. URL: https://doi.org/10.1109/MCSE.2019.2900945.

[20] Nicolas Hubner, Jean-Rémy Falleri, Raluca Uricaru, Thomas Degueule, and Thomas Durieux. "What Happened in This Pipeline? Diffing Build Logs with CiDiff". In: *Proceedings of the ACM on Software Engineering* 2.ISSTA (2025), pp. 2023–2044.

[21] Daniel S. Katz. "Citation and Reproducibility in Software". 2017. URL: https://indico.cern.ch/event/570249/contributions/2434087/a

ttachments/1400440/2150694/Citation_and_Reproducibility_in_Software-no-animations.pdf (visited on 10/10/2025).

[22] 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).

[23] 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).

[24] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. "An empirical analysis of flaky tests". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 643–653. ISBN: 978-1-4503-3056-5. URL: https://dl.acm.org/doi/10.1145/2635868.2635920 (visited on 09/06/2023).

[25] Julien Malka. *Published upon acceptance; very large dataset.*

[26] Julien Malka. *Published upon acceptance; very large dataset.*

[27] [SW] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann, *Code archive for the paper "Does Functional Package Management Enable Reproducible Builds at Scale? Yes."* 2025. URL: https://gitlab.telecom-paris.fr/julien.malka/a-decade-of-software-reproducibility-in-the-nix-package-ecosystem.

[28] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. "Does Functional Package Management Enable Reproducible Builds at Scale? Yes". In: *22nd IEEE/ACM International Conference on Mining Software Repositories, MSR@ICSE 2025, Ottawa, ON, Canada, April 28-29, 2025*. IEEE, 2025, pp. 775–787. URL: https://doi.org/10.1109/MSR66628.2025.00115.

[29] 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).

[30] Brian Matthews, Esther Conway, Jim Woodcock, Catherine Mary Jones, Juan Bicarregui, and Arif Shaon. "Towards a Methodology for Software Preservation". In: *Proceedings of the 6th International Conference on Digital Preservation, iPRES 2009, San Francisco, CA, USA, October 5-6, 2009*. 2009. URL: https://hdl.handle.net/11353/10.294040.

[31] 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).

[32] Mathias Meyer. "Continuous Integration and Its Tools". In: *IEEE Software* 31.3 (May 2014). Conference Name: IEEE Software, pp. 14–16. ISSN: 1937-4194.

[33] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. "Fixing dependency errors for Python build reproducibility". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, July 2021, pp. 439–451. ISBN: 978-1-4503-8459-9. URL: https://dl.acm.org/doi/10.1145/3460319.3464797 (visited on 09/06/2023).

[34] 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).

[35] *Nix & NixOS: Declarative builds and deployments*. en. URL: https://nixos.org/ (visited on 10/08/2025).

[36] *NixOS Reproducible Builds*. en. URL: https://reproducible.nixos.org/ (visited on 10/21/2024).

[37] *Overview of various statistics about reproducible builds*. URL: https://tests.reproducible-builds.org/debian/reproducible.html (visited on 10/21/2024).

[38] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. "A Survey of Flaky Tests". In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (Oct. 2021), 17:1–17:74. ISSN: 1049-331X. URL: https://dl.acm.org/doi/10.1145/3476105 (visited on 09/06/2023).

[39] Jeffrey M. Perkel. "Challenge to scientists: does your ten-year-old code still run?" en. In: *Nature* 584.7822 (Aug. 2020). Bandiera_abtest: a Cg_type: Technology Feature Number: 7822 Publisher: Nature Publishing Group Subject_term: Computational biology and bioinformatics, Computer science, Research data, Software, pp. 656–658. URL: https://www.nature.com/articles/d41586-020-02462-7 (visited on 09/06/2023).

[40] 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 (visited on 10/17/2024).

[41] 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).

[42] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. "An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. May 2017, pp. 345–355.

[43] 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).

[44] 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).

[45] 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).

[46] *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).

[47] ReScience. *Ten Years Reproducibility Challenge*. 2020. URL: http://rescience.github.io/ten-years/ (visited on 09/06/2023).

[48] Mahadev Satyanarayanan. "Saving software from oblivion". In: *IEEE Spectrum* 55.10 (Oct. 2018). Conference Name: IEEE Spectrum, pp. 36–41. ISSN: 1939-9340.

[49] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. "Programmers' build errors: a case study (at google)". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 724–734. ISBN: 978-1-4503-2756-5. URL: https://dl.acm.org/doi/10.1145/2568225.2568255 (visited on 09/06/2023).

[50] Aman Sharma, Benoit Baudry, and Martin Monperrus. "Canonicalization for Unreproducible Builds in Java". In: *CoRR* abs/2504.21679 (2025). arXiv: 2504.21679. URL: https://doi.org/10.48550/arXiv.2504.21679.

[51] Len Shustek. "What Should We Collect to Preserve the History of Software?" In: *IEEE Annals of the History of Computing* 28.4 (Oct. 2006). Conference Name: IEEE Annals of the History of Computing, pp. 112–111. ISSN: 1934-1547.

[52] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. "A comprehensive study of bloated dependencies in the Maven ecosystem". en. In: *Empir Software Eng* 26.3 (Mar. 2021), p. 45. ISSN: 1573-7616. URL: https://doi.org/10.1007/s10664-020-09914-8 (visited on 09/06/2023).

[53] Francesco Strozzi et al. "Scalable Workflows and Reproducible Data Analysis for Genomics". In: *Evolutionary Genomics: Statistical and Computational Methods*. Ed. by Maria Anisimova. New York, NY: Springer New York, 2019, pp. 723–745. ISBN: 978-1-4939-9074-0. URL: https://doi.org/10.1007/978-1-4939-9074-0_24.

[54] 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).

[55] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. "An empirical characterization of bad practices in continuous integration". en. In: *Empir Software Eng* 25.2 (Mar. 2020), pp. 1095–1135. ISSN: 1573-7616. URL: https://doi.org/10.1007/s10664-019-09785-8 (visited on 09/06/2023).

[56] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. "A large-scale empirical study of compiler errors in continuous integration". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 176–187. ISBN: 978-1-4503-5572-8. URL: https://dl.acm.org/doi/10.1145/3338906.3338917 (visited on 09/06/2023).