An Empirical Study on Reproducible Packaging in Open-Source Ecosystems

Giacomo Benedetti*, Oreofe Solarin†, Courtney Miller‡, Greg Tystahl§, William Enck§, Christian Kästner‡, Alexandros Kapravelos§, Alessio Merlo¶ and Luca Verderame*
*University of Genoa, †Case Western Reserve University, ‡Carnegie Mellon University, §North Carolina State University, ¶CASD - School of Advanced Defense Studies

Abstract—**The integrity of software builds is fundamental to the security of the software supply chain. While Thompson first raised the potential for attacks on build infrastructure in 1984, limited attention has been given to build integrity in the past 40 years, enabling recent attacks on SolarWinds, event-stream, and xz. The best-known defense against build system attacks is creating *reproducible builds*; however, achieving them can be complex for both technical and social reasons and thus is often viewed as impractical to obtain. In this paper, we analyze reproducibility of builds in a novel context: reusable *components* distributed as *packages* in six popular software ecosystems (npm, Maven, PyPI, Go, RubyGems, and Cargo). Our quantitative study on a representative sample of 4000 packages in each ecosystem raises concerns: Rates of reproducible builds vary widely between ecosystems, with some ecosystems having all packages reproducible whereas others have reproducibility issues in nearly every package. However, upon deeper investigation, we identified that with relatively straightforward infrastructure configuration and patching of build tools, we can achieve very high rates of reproducible builds in all studied ecosystems. We conclude that if the ecosystems adopt our suggestions, the build process of published packages can be independently confirmed for nearly all packages without individual developer actions, and doing so will prevent significant future software supply chain attacks.**

I. Introduction

Reproducible builds to secure build integrity. While the world runs on open-source software, software is rarely consumed as source. The software build process that transforms source code to its consumed artifact is a long-standing security risk. In his 1984 Turing Award Lecture, Thompson [1] described a process to create an undetectable backdoor in software by modifying the compiler that compiles a compiler, leaving no trace of the backdoor in source code. Nearly 40 years later, the 2020 attack on *SolarWinds* subverted the build system to produce binary artifacts signed with SolarWinds's official code signing keys [2], and the 2018 *event-stream* and 2024 *xz* supply chain attacks try to inject malicious code into deployed components that are not visible in the source code [3].

The best known defense against build system attacks is creating *reproducible builds* [4]: A build is reproducible if executing the build on two or more different machines (e.g., by different organizations) produces a bitwise-identical artifact when given the same source code, build environment, and build instructions. A reproducible build provides strong evidence that the build process was not tampered with and that the resulting artifact corresponds to the source code, which is particularly important for high-profile projects (e.g., Tor [5] and Bitcoin [6]) and essential digital infrastructure. Approaches such as CHAINIAC [7] build on top of reproducible builds to create collectively verified builds. For example, it is unlikely that Google, Microsoft, and Amazon's build processes will be simultaneously compromised, especially when the compromise of any subset of the parties is easy to identify.

Achieving reproducible builds is widely viewed as very hard [8], and currently, only a few developers invest effort into reproducible builds. There are many intricate reasons why a build may not be reproducible in practice [4] – for example, time, environment variables, and build location may be embedded in binary executables and archive files that package artifacts. In addition, there are many more potential sources of system differences and non-determinism in builds, such as different system locales, pseudorandom number generators used during compilation or code generation, and process scheduling. While the Debian project has spent a decade removing sources of unreproducibility and has achieved over 95% reproducible builds on AMD64 [9], reproducible builds remain a hard challenge – a recent interview study of practitioners invested in reproducible builds highlighted many technical and social challenges of resolving build unreproducibility [10].

Reproducible component builds to secure package ecosystems. In this paper, we consider reproducible builds in a novel context: the reusable *components* distributed as *packages* with package managers in popular software ecosystems, including npm (JavaScript), Maven (Java), PyPI (Python), Go, RubyGems (Ruby), and Cargo (Rust).[1] Today, most applications are built with reusable open-source components, and build processes usually rely on the archives distributed with package managers rather retrieving the original source code from repositories.

Attackers are actively exploiting the gap between the source code in a repository and the release of a package. In 2018 the account controlling the popular PyPI package `ssh-decorate` was hijacked to upload a version that collected users' SSH credentials to send them to a remote server [11] – an independent build of the package would have identified that the hash

[1]We use the terminology of *components* and *applications* common in the discourse of software supply chains. A component here can be any reusable software artifact, including libraries, frameworks, infrastructure tools, and non-code artifacts. Components are typically, but not necessarily distributed as *packages* with a *package manager*.

of the version in PyPI differed from the one built from the source. The *xz* attack [12] also exploited this gap: While the malicious code was obfuscated in the source code repository as a malformed compressed archive used as a negative test case, it was only enabled by an Autoconf file that was *not* in the repository. The released dist tarball of xz included build instructions generated by the malicious Autoconf file. Interestingly, the Debian Reproducible-Builds project *did not* detect the *xz* attack, because they used the released dist tarball as their source. Even though we do not study reproducibility of C/C++ dist tarballs, this is exactly the kind of scenario of manipulated distributed packages (source or binary) we address in this paper.

Although the primary focus of the discourse on reproducible builds is on applications rather than components, the limited prior research on component reproducibility paints a gloomy picture, reporting severe reproducibility issues for most components studied in npm and PyPI. Specifically, Vu et al. [8] attempted to match the code in over 2,000 popular PyPI packages with their original source code (without actually building the packages) and found a wide range of differences, most benign. Furthermore, Goswami et al. [13] studied the build reproducibility of over 3000 versions popular npm packages and found significant challenges resulting from version drift of build tools.

In our work, we explicitly adopt a much broader scope, comparing practices in multiple ecosystems and analyzing the role of package managers' tooling: Open-source infrastructure has formed distinct communities, interdependent *ecosystems*, often around languages, frameworks, and platforms, often with distinct practices and tools [14], [15], [16]. Among others, practices and tools for packaging and indexing artifacts differ widely between (a) compiled languages that share binaries and (b) interpreted languages that share archives of source code, sometimes transpiled, sometimes including some binary code extensions, sometimes including code in multiple languages – for instance, the package managers *npm* and *PyPI* have adopted entirely distinct toolchains. Different practices and tools may lead to widely different outcomes for reproducibility. Hence, *we specifically study the difference of component reproducibility across ecosystems and the influence of tooling choices in package managers.*

Research overview. For the purposes of this study, we consider *reproducible packaging* as producing a bitwise-identical artifact from the same source code, build environment, and build instructions. This analysis of reproducible packaging is distinct from, but a necessary foundation for, comparing a locally created artifact to a published artifact (see Section III-C). We perform a quantitative study applying `reprotest` [17] to a representative large random sample of 4000 packages in each of six packaging ecosystems (npm, Maven, PyPI, Go, RubyGems, and Cargo) and investigate the reasons for reproducibility issues . Specifically, we answer the following research questions:

RQ1 (Reproducible Package Builds): *How many packages are reproducible* as is *in each ecosystem?* We find that almost all packages in Cargo and npm are reproducible,[2] but only very few to none are in RubyGems, PyPI, and Maven. Go packages simply include a reference to a source code repository and do not include any artifacts – therefore component builds are not relevant for Go. This highlights the substantial differences between ecosystems and also seems to initially confirm the gloomy reports of vast reproducibility problems from previous research.

RQ2 (Causes of Unreproducibility): *To what extent are unreproducible package builds caused by the toolchain and fixable through toolchain changes?* Analyzing the causes of unreproducibility, we find that nearly all unreproducible package builds were caused by nondeterminism in the build process (e.g., paths, time, file permissions, locale) that can be controlled by the package manager's tooling, either through configuration or through small changes to the tools. With configuration or tooling changes in RubyGems, PyPI, and Maven,[3] almost all packages become reproducible in these ecosystems. This result paints a much more optimistic picture than prior reports and our initial findings: Our results highlight that small changes to ecosystem-wide tooling have a substantial lever to improve package reproducibility to the point that almost all packages are reproducible in each studied ecosystem, without having to convince every single package maintainer to take actions to ensure reproducibility.

RQ3 (Native Code Extensions): *To what extent does the presence of native code inside of packages influence build reproducibility?* Ruby, Python, and even JavaScript packages can include native code extensions. Analyzing specifically packages that contain native code extensions, even though the package managers in these ecosystems incorporate native code differently (e.g., compiling it into the package vs. including source code), our study shows that native code extensions rarely negatively influence reproducibility, with reproducibility rates in each ecosystem almost identical to that of packages without native code extensions. This is another positive finding, highlighting that native code extensions do not pose additional barriers in practice.

RQ4 (Reproducible Builds and Compilation): *To what extent does compilation of package dependencies into distributable artifacts of application affect the build reproducibility?* As components in ecosystems are usually used to build applications, we explore the downstream effects of component reproducibility (or a lack thereof) for applications relying on components in the ecosystem. We find that application builds using the studied packages are reproducible for 100% of the Go packages and greater than 90% of the Maven packages. The results for Cargo were more nuanced: due to timestamps used as part of cryptographic signatures, potentially all packages can cause downstream build reproducibility problems;

[2]For simplicity we use the term *reproducible* when the package manager tooling does not insert any reproducibility issue.

[3]As noted in Section IV-B1, Maven independently added timestamp-related build reproducibility fixes too their tooling during the preparation of the camera-ready version of this paper.

however, in practice, these cryptographic signatures will not change, and greater than 80% of the studied packages had reproducible builds. This is a final supporting piece that also points toward the message that package reproducibility may be a smaller obstacle than originally expected.

Recommendations. Overall, our results largely paint a positive picture for the studied software ecosystems. Our results indicate that package managers can take advantage of reproducible builds to protect against build system or account compromise by doing the following: First, the identified toolchain issues must be addressed by providing more options to control nondeterminism during the build process. Second, package manager tooling should adopt default configurations that create reproducible builds without having each maintainer configure their build. Finally, package managers *must* make `buildinfo` files available. While our study did not consider build tool version drift, prior work [13] identified this as a significant challenge for npm; without `buildinfo` files, it is very challenging to compare our independent build with the packages distributed by package managers.

Contributions. In summary, we contribute a large-scale quantitative analysis of the state of reproducible builds across six software ecosystems, contrasting the prevalence of reproducible package builds and infrastructure changes that can support increased reproducibility for each of the ecosystems respectively. Our results have implications for understanding how to improve the state of reproducibility on a macro-ecosystem level as well as concrete changes to support individual package users and developers.

Source code, data, and additional material for this paper are available in our replication package [18].

II. Background

A. Packaging Ecosystems

Software applications are routinely developed by reusing existing (often open-source) components. The application and its dependencies on components, which again may depend on other components, form a *software supply chain*. Components are usually distributed as *packages* with a *package manager* and corresponding *package repository*, such as *npm* and *PyPI*.

Groups of packages, their developers, and their users often form an interdependent *ecosystem*. Software ecosystems are frequently underpinned by a common technological platform or market [19]. A software ecosystem can be defined as a *packaging ecosystem* when it revolves around package managers for a specific programming language [16].

A package within an ecosystem contains the files necessary for other software to use its functionality. These may include source code and binary files, as well as tests and documentation. Among those files, there is usually a *specification file* containing information on how the package manager must build the package and other metadata. There are two kinds of specification files: static and dynamic. A static specification file contains metadata that the package manager uses during the build process. A dynamic file may contain arbitrary code or elements that are executed by the package manager during the build process.

In the npm, PyPI, and RubyGems package ecosystems, packages include primarily source code, but may also include *native code extensions*. Native code extensions provide an API for compiled code, usually written in C, typically to optimize for resource-intensive activities.

A packaging ecosystem typically comes with tooling to create and publish components as packages, such as `pip` for PyPI and `mvn` for Maven. Most ecosystems have widely used standard tools, but there may be competing tools such as `npm` and `yarn` in the npm ecosystem, which provide different command-line tools to interact with the same package repository. In addition, most tools are highly configurable – for example `pip` uses a frontend–backend approach, where `pip` delegates the actual packaging to a configurable build backend; in addition `pip` works on two interfaces using two different specification files, an interface (legacy) based on the `setup.py` file and another interface based on the `pyproject.toml` file. Similarly, `mvn` is plugin-based, ingesting a `pom.xml` specification file to control the actions of various plugins.

B. Distribution Model

Figure 1 shows a typical distribution model for packaging ecosystems. It consists of two processes: (1) package and upload components as packages to repositories, and (2) installation and build of packages for applications.

Components are usually developed and tested locally on a developer's machine or with some public build and continuous integration infrastructure. The source code of open-source components is publicly available, and anybody can suggest modifications. The package manager's toolchain can then be used to release the package as a distributable artifact. At this stage, the package manager takes the source code and the specification files, collects dependencies, collects and possibly compiles code, and generates a package. This process is controlled by the various configuration files in the repository and potentially additional command-line arguments. During the packaging stage, information about the build environment can be captured as metadata (e.g., timestamps and release version). The resulting package file is then *uploaded* in the ecosystem's package repository.

Consumers of a component, such as other components and applications, usually use the packaged version in the package repository, downloading and installing it through the package manager, assuming that the distributed package corresponds to the component's source code in its public repository [20]. The dependencies of an application can be integrated during the build phase and included in the final artifact, or they might be gathered and installed on the end user's system separately. The resulting application can then be distributed to end users internally or publicly through various channels, including direct downloads, application stores, and as packages in package ecosystems (e.g., common for developer applications like static analysis tools or test runners).

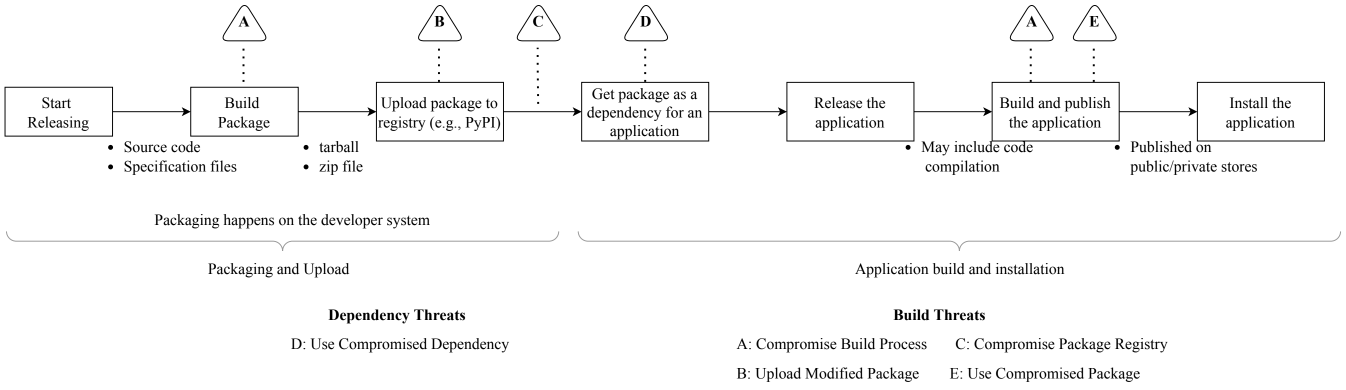
A
B
C
D
A
E
Start Releasing
Build Package
Upload package to registry (e.g., PyPI)
Get package as a dependency for an application
Release the application
Build and publish the application
Install the application
Source code
Specification files
tarball
zip file
May include code compilation
Published on public/private stores
Packaging happens on the developer system
Packaging and Upload
Application build and installation
Dependency Threats
D: Use Compromised Dependency
Build Threats
A: Compromise Build Process
C: Compromise Package Registry
B: Upload Modified Package
E: Use Compromised Package

Fig. 1. Workflow from source code to distribution.

Various threats can affect the distribution model (see Figure 1). According to Ladisa et al. [3], these threats can be implemented through multiple attacks. The build process can be compromised (A) due to weak configurations, vulnerabilities, or malicious components. A modified package may be uploaded to the registry (B) by compromising the host or maintainer systems or by hijacking a legitimate account. Similarly, a package registry can be compromised (C). Once distributed, compromised packages affect other packages or applications that use them (D). Both package and application build processes may use compromised packages (E). Reproducible builds are one countermeasure to ensure build integrity, ensuring packages are built from current, *unmodified* sources and dependencies.

C. Challenges of Reproducible Builds in OSS Supply Chains

The idea of reproducible builds has been broadly promoted to ensure an independently verifiable path from source to published artifacts where verifiably no additional[4] vulnerabilities or malicious code has been introduced. Identical results for every build of a given source allow multiple parties to come to a consensus and highlight any deviations from the expected build result.

Reproducible builds have two requirements that can be difficult to ensure: (1) The build process must be deterministic. (2) The build environment must be either recorded or pre-defined. Build tools and programming languages were not originally designed for reproducibility and contain many causes of nondeterminism that affect build reproducibility. Lamb and Zacchiroli [4] provide an overview of common sources of nondeterminism during the build process:

- *Build timestamps* are the main source of unreproducible builds. Many tools embed timestamps inside build artifacts, even though they may have limited practical value. The Reproducible Builds project proposed the `SOURCE_DATE_EPOCH` environment variable as a way to communicate a fixed timestamp to build systems [21].
- *File ordering* for the `readdir(3)` system call is not specified in the POSIX Unix standard, and differently ordered file lists may affect build artifacts. To avoid these issues, build systems should impose a deterministic order on any directory iteration encoded in its artifacts, e.g. via an explicit `sort()`.
- *Archive metadata* of `zip` archives and tarballs (i.e., `tar` archives) may contain timestamps and permissions for each file.
- *Randomness*, intentional or accidental, of any compilation or code generation step in the build may influence resulting binaries.

The reproducible-builds.org project provides educational materials, resources, and tools to support developers and software projects in making their build processes reproducible, including `reprotest` [17] to automate the process of building a package multiple times in diverse environments and `diffoscope` [22] to help find the differences between binary packages and directories.

Beyond technical challenges, recent research has studied the *perceptions* of reproducible builds and the *social challenges to their adoption*. Fourné et al. [10] analyzed enablers and blockers for adopting reproducible builds in the open-source community. They report that most industry practitioners *considers reproducible builds to be out of reach* – reproducible builds are seen as valuable but not essential. Other studies confirm that many developers are not aware of reproducible builds when developing their software [23], [24]. Butler et al. [25] identified the need for reproducible builds from a security point of view and the reasons for their limited adoption, minimal business impact, and limited awareness and perceived challenges.

Overall, adoption of reproducible builds is uneven, with dedicated efforts in Debian achieving substantial success [9], but minimal attention to reproducible builds in other areas.

With regards to reproducibility at the package level, Goswami et al. [13] examined the reproducibility of npm packages by comparing the build output of upstream repository code against artifacts stored on the npm registry. Vu et al. [8] did not directly focus on reproducible builds, but

[4]Malicious code already hidden in the original source or tools used in the build may be built reproducibly.

they argued for reproducible builds as a security solution to phantom artifacts, highlighting how difficult it is to achieve reproducible builds. In parallel with our study, Kenshani et al. [26] investigated the feasibility of automatically generating `.buildspec` files from metadata available in Maven packages. The build obtained by the automatically generated `.buildspec` file is then compared to the build hosted in Reproducible Central, which keeps trace of reproducibility for part of the packages in the Maven ecosystem. As additional result of their study, they conducted an explorative analysis trying to find common causes of reproducibility issues. They conclude that reproducibility of Maven packages can be achieved by trivial adjustments to the POM.xml file. We confirm this speculation in Section IV-B1.

In another line of work, Randrianaina et al. [27] studied the impact of configuration options on reproducible builds in highly-configurable systems, i.e., Linux, Toybox, and Busybox. By fixing the build environment they were able to understand how such options affect the build reproducibility. We use a similar technical approach in this study to focus on reproducibility issues related to the package manager.

Automatic approaches to support reproducible builds were proposed by Ren et al. with three tools: RepLoc [28], RepTrace [29], and RepFix [30], which localize and fix sources of nonreproducibility for Debian applications. Here, we focus more broadly on builds challenges across package ecosystems, including different practices and policies adopted by communities [14], and different evolution of dependency networks in different ecosystems [31], [32], [33], [34], [16].

III. Research Design

In this paper, we conduct an in-depth empirical study of reproducible builds in six packaging ecosystems to answer the four research questions we outline in the introduction:

RQ1 How many packages are reproducible *as is* in each ecosystem?

RQ2 To what extent are unreproducible package builds caused by the toolchain and fixable through toolchain changes?

RQ3 To what extent does the presence of native code inside of packages influence build reproducibility?

RQ4 To what extent does compilation of package dependencies into distributable artifacts of application affect the build reproducibility?

We use the following high-level research design: For each packaging ecosystem, we randomly generate a large, representative sample of packages and attempt to build them under varying environmental conditions from source code in their open-source repositories. This allows us to quantitatively study reproducibility rates across ecosystems. Based on our findings, we then explore sources of unreproducibility and corresponding interventions in package manager toolchains to quantify how tooling changes affect reproducibility rates.

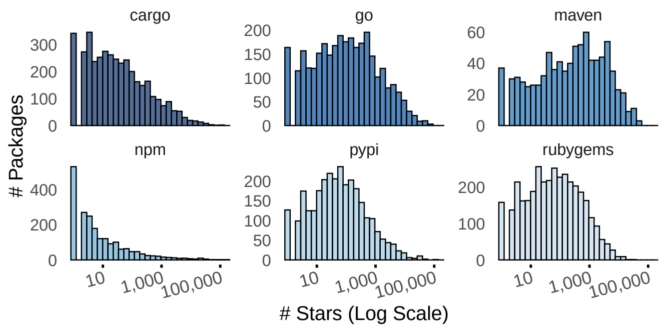

Fig. 2. The samples used in our analysis allow us to obtain the practices of the average developer dealing with reproducible builds. Thus, most of packages in the samples have a very low popularity, however, our samples also contain some very popular packages.

A. Analyzed Packaging Ecosystems

We focus on ecosystems that revolve around package managers for specific programming languages.

To select the package ecosystems used in our analysis, we searched for ecosystems representing a large community with over 100,000 packages each; we then used an information-oriented selection strategy (maximum-variation cases, following case study research logic [35]), to identify a set of ecosystems with different characteristics – e.g., native code extensions, dynamic specification files – and with different strategies in their distribution model – e.g., compilation output: binary/bytecode, plugin-based build, package indexing. This process yielded the following six ecosystems: Cargo, Go, Maven, npm, PyPI, and RubyGems. We argue that this set of ecosystems represents the current software landscape. Their position regarding reproducible builds is critical for global software supply chain security posturing.

B. Sample of Packages

Given the size of the ecosystems and the prohibitively high cost of repeatedly building packages, it is infeasible to analyze all 100,000 to 5 million packages in each ecosystem. Instead, we analyze a large, representative, random sample from each and make statistical generalizations.

a) Packages (RQ1, RQ2): To generate our sample of packages for each ecosystem, we begin with a list of all packages in the packaging ecosystem indexed by ecosyste.ms. In order to minimize the chance of build issues during the analysis, we filter packages by checking for the presence of ecosystems' specification files — e.g., `package.json` for npm and `setup.py` / `pyproject.toml` for PyPI—, discarding packages without them. We then generate a random sample of 4000 packages from this pool. We intentionally did not select only the most popular projects, so that we can generalize our findings to the entire population of packages in the ecosystem, rather than merely reporting numbers about the most popular packages. As a representative sample, it is expected that many of the analyzed packages have few stars[5], but some are very

[5]Stars are collected from ecosyste.ms API

popular in terms of stars, as shown in Figure 2. Having such a distribution of popularity allows us to obtain practices of average developer. With the relatively large sample size, results derived from this sample afford high generalizability with less than 1.54% margin of error at 95% confidence levels by standard sample size calculations. The precise margin of error varies slightly between ecosystems due to their different population sizes and population proportion of reproducibility, but the differences are negligible.

b) Packages with native code extensions (RQ3): To concentrate on the effect of native code on reproducible builds, we generated a sample of packages with native code extensions and a second sample of packages without native code extensions. To generate these samples, we first started by dividing the packages from our RQ1 sample according to whether native code extensions were present and designated them to the appropriate sample. This resulted in a sample of 398 (PyPI), 394 (RubyGems), and 129 (npm) packages with native code extensions and a sample of complementary cardinality without native code extensions. However, because we wanted these samples to contain 4000 packages so they were comparable to the RQ1 sample, we continued to draw fresh random samples from the whole package population until we found 4000 packages for each of the samples. The detection of native code extensions in packages is ecosystem-dependent: It is based on the presence of specific files (e.g., the extconf.rb file for RubyGems) or configurations (e.g., the ext modules in the setup.py file for PyPI). The margin of error is again less than 1.54% with 95% confidence. The complete list of criteria can be found in the replication package [18].

c) Compilation Process Impact (RQ4): Similarly to RQ3, to explore the impact of the complication process on reproducible builds, we generate a sample of packages with packages that can be compiled. For Cargo, Go, and Maven, we collected large random samples of 4000 packages each by applying the following filters:[6] For Cargo, the Cargo.toml file must contain a binary target to allow the package manager to compile the project. For Go, an entry point function, i.e., the main function, must be present among the project files. For Maven, the maven-compiler-plugin must be listed in the pom.xml specification file.

C. Reproducibility analysis for packages (RQ1–3)

For each package in our samples, we identify the corresponding GitHub repository and clone the most recent version of the code. We then use the ecosystem's default command to build the package (e.g., `pip wheel .`). The list of package managers and commands that we used is available in our replication package [18]. When the open-source repository is missing or the build fails the package is reported as missing and discarded from the analysis since it does not offer information on its build reproducibility. In those cases, a new package matching the same criteria is randomly selected from the whole package population to replace the discarded one. We discarded about 800 packages for each ecosystems before reaching the required number of 4000. The build failures were caused by missing system libraries and mistakes in specification files.

[6]The other ecosystems, i.e., npm, PyPI, and RubyGems, do not have a standard process to generate a compiled artifact.

The package build reproducibility of all packages is tested using `reprotest` on the same machine. Fixing the build infrastructure specifications — e.g., toolchain versions, dependencies, operating system — makes it possible to focus on the impact of reproducibility issues related to the package manager. The tool is set up with a build command and a build output for each ecosystem. For example, `reprotest -variations=+time 'pip wheel -w dist  <package_path>' 'dist/*.whl'`: the time variation is applied to the build of packages for PyPI and the resulting wheel files are tested for differences. `reprotest` runs one time for each variation. Our replication package contains the full catalog of the used variations. We consider a build as reproducible when `reprotest` does not report reproducibility issues for any of the tested variations. Failing variations are recorded to subsequently investigate the causes.

We explicitly do not compare the compiled artifact with the artifact uploaded to the package manager for two reasons. First, identifying which specific revision of the source code repository was used to produce the released package is nontrivial and the subject of extensive research [8], [36], [37], but entirely orthogonal to our exploration of whether a package can be reproduced from the same source code. Second, environmental dependencies such as compiler versions, system libraries, and operating system configurations can vary significantly across different build environments, leading to potential discrepancies in the compiled artifacts, which is again orthogonal to our research on whether the package manager build process influences a package build reproducibility. For example, a recent comparison of an independent build of packages on ftp.debian.org found that only around 30% of the published packages could be reproduced [38], despite the fact that over 95% of packages can be built reproducibly using `reprotest` [9]. Hence, we only compare build outcomes under different environments from the latest revision of the source code in the package's repository.

The results of the tests show which reproducibility issues have the biggest impact on build reproducibility. We look for the reasons for unreproducibility behind the package manager implementations. The majority of the analysis is done manually, by looking at particular portions of code that appeared to be the cause of an unreproducibility problem. We created several automated scripts to search for possible unreproducibility causes when reproducibility issues were not triggered by the package manager directly. For example, one such script looked for dynamic dates in specification files.

D. Reproducibility analysis for compilation (RQ4)

We examined Cargo, Go, and Maven since they offer a proper compilation method. The first two ecosystems yield binary files, while the last one generates an archive with compiled Java bytecode. We employed the same methodology as for the other research questions to examine the influence of compilation. The configuration of the `reprotest` tool uses the same variations used for RQ1–3, but it receives different build commands and builds outputs (such as `cargo build`). We use the assumption that the compiler requirements are known and that reproducibility problems are not the result of the compiler since we are interested in examining how package manager operations affect reproducible builds.

E. Limitations and threats to validity

Evaluating build reproducibility includes technical details varying by the level of abstraction taken into consideration. For the purpose of this study (package manager impact on reproducible builds), we designed our methods to identify reproducibility issues at the build infrastructure level. By excluding reproducibility issues originated by the system level, e.g., architecture specifications, we may miss unreproducible builds caused by the package manager because of platform-specific causes.

While build reproducibility has a binary result (reproducible or not), identifying causes of reproducibility issues is not as straightforward. We used a specific tool, `reprotest`, in an attempt to identify the causes of reproducibility issues; however, `reprotest` may be subject to unexpected bugs and design issues that have the potential to impact our results. To mitigate this risk, we carefully reviewed the `reprotest` implementation and currently open issues on the code repository, identifying a minor potential source of false positive unreproducible builds. The tool may cause failure in the handshake with the package registry because of widely varying system time. We reduced the range of this variation. After patching this (potential) implementation issue, we assume that the tool's results can be considered scientifically accurate.

We can statistically generalize our results to report trends for reproducible builds for the entire ecosystems with tight error margins. This method analyzes the general, representative behavior of package builds, but we may miss the behaviors of individual developers; for example, the behaviors of a few developers of high-impact packages may differ from those of the general population.

IV. Results

We present results of our experiments by research question.

A. RQ1: Reproducible Package Builds As Is

Attempts to reproduce packages *as is*, without any changes to package manager toolchains, yields wildly different results across ecosystems as shown in Figure 3: A first group of ecosystems achieves reproducible builds for nearly all the sampled packages. Specifically, Cargo and npm have 100% reproducible package builds. A second group of ecosystems achieves very low percentages of reproducible package builds. Specifically, Maven, PyPI, and RubyGems have 2.1%, 12.2%, and 0% of reproducible package builds, respectively.

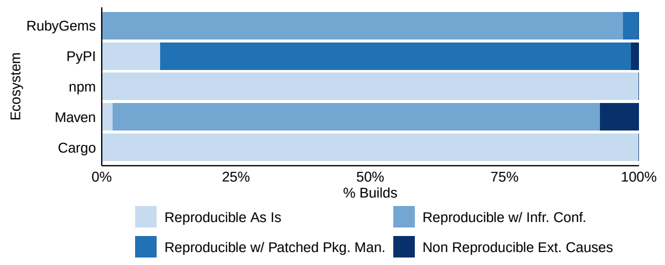

Fig. 3. This figure shows the reproducibility of package builds across different ecosystems, categorized as: i) reproducible as is, ii) reproducible with infrastructure configuration as requested by the package manager, iii) reproducible with a patched package manager, and iv) unreproducible due to external issues not related to the package manager.

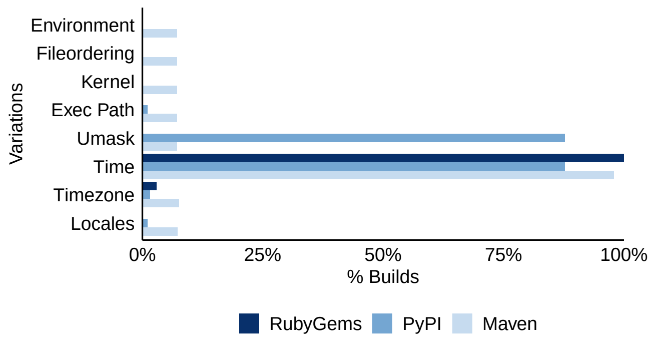

Fig. 4. Percentage of builds per ecosystem that were unreproducible because of different variations.

Takeaway: Rates of reproducible builds vary widely between ecosystems, with some ecosystems having all reproducible packages whereas others have reproducibility issues in nearly every package.

B. RQ2: Tooling Sources of Reproducibility Issues and Fixes

Fortunately, the situation is not as bleak as it might seem when analyzing the sources of reproducibility issues. In Figure 3, we report that even in package ecosystems with low package reproducibility as *as is*, most packages are reproducible if considering package manager toolchain configurations or small patches to package manager tooling. For example, just by changing the configuration options of the package manager toolchain, Maven and RubyGems increase from 2.1% to 92.6% and from 0% to 97.1% reproducible package builds, respectively. As we discuss in Section IV-B2, our analysis led us to discover small changes that can be made to PyPI's package manager that increase reproducible package builds from 12.2% to 98%. Thus, what might be perceived as very low package build reproducibility is, in fact, quite promising. In Section IV-B1, Section IV-B2, and Section IV-B3 we detail how changing the build configuration and tooling drastically increases the reproducibility of package builds, and how some causes of unreproducibility cannot be easily solved, contributing to answer RQ2.

1) Reproducible with Infrastructure Configuration: We applied solutions proposed by the selected ecosystems to study how infrastructure configuration affects the build reproducibility. By reviewing the `reprotest` logs, we found that timestamps and permission masks are the largest contributing factors (see Figure 4). In particular, timestamp metadata leads to unreproducible builds for 92.4% of Maven, 87.77% of PyPI, and 97.1% of RubyGems packages.

All five studied ecosystems use archives to distribute package artifacts. However, Cargo and npm drastically reduces (actually removes in our sample) the presence of reproducibility issues caused by hard-coding fixed values in their package managers' code. The other three ecosystems use a different approach: Because they use the current time or file timestamps unless developers set a specific build time by configuring the package manager, using the package manager as-is makes builds unreproducible.

The packaging infrastructure can be configured to help remove sensitivity to timestamp metadata. For example, the PyPI and RubyGems packaging infrastructure can set the build time using the `SOURCE_DATE_EPOCH` environment variable. However, the variable value must be communicated along with the package, e.g., via a `.buildinfo` file [39]. In contrast, Maven allows the developer to use the `outputTimestamp` POM specification file property to set a fixed timestamp in the package metadata. Unfortunately, this solution suffers from two major limitations: (1) the developer has to set it in the pom.xml file – it is not by default;[7] and (2) Maven plugins have to implement this feature to make it effective.

In general, the studied package managers do not require or suggest that configuration of the build infrastructure is necessary. Among the three ecosystems, only Maven's documentation clearly explains how to achieve build reproducibility. This lack of information for other ecosystems, combined with the challenges of communicating build parameters (e.g., `SOURCE_DATE_EPOCH`), is likely a large factor for unreproducible package builds.

Finally, PyPI package build reproducibility is largely impacted by both timestamp *and* umask values. Umask values cannot be addressed via packaging infrastructure configuration and are discussed in Section IV-B2. That said, different package manager tools operate differently. Recall Section II, `pip` is a frontend to deal with multiple building backends. It refers to the specification file to invoke the right backends. We found that the `flit` and `hatch` building backends fix the archive metadata similar to Cargo and npm. Since 12% of PyPI builds use either `flit` or `hatch`, they are reproducible as is. However, most of the remainder of the PyPI ecosystem suffers from archive metadata reproducibility issues.

[7]During the preparation of the camera-ready version of this paper, an automation was proposed and implemented in Maven 4.0.0-beta (issues.apache.org/jira/browse/MNG-8258) that sets a default timestamp. Packages that only have this timestamp reproducibility issue should build reproducibly in the future.

Takeaway: Configuring packaging infrastructure to control timestamp metadata makes over 90% of Maven and 97% of RubyGems package builds reproducible.

2) Reproducible with Patched Tooling: While controlling timestamp metadata via infrastructure configuration has a significant impact on package build reproducibility, it does not address all issues. In this subsection, we investigate how patching the packaging tools can address *umask* and other contributing factors. As shown in Figure 4, umask is also a large contributing factor for PyPI. As for timestamp reproducibility issues, most of the issues caused by umask affect the archive metadata. The source of much of the remaining unreproducibility are dynamic metadata.

Recall from Section II that dynamic metadata allows RubyGems and PyPI developers to define a metadata value in the specification file programmatically. Developers define dynamic metadata using the ecosystem's language or specific properties offered by package managers. The value is then interpolated within the build environment. We identify five root causes in the PyPI and RubyGems package managers: file ordering, locales, umask, time, and timezone. Examples of these causes can be found in our replication package.

The code creating dynamic metadata is defined in packages. Contacting package maintainers to modify their code to produce reproducible package builds would be very time-consuming and may not result in changes to the build specifications. For example, a recent interview study of practitioners working on reproducible builds found that project maintainers are not always receptive to making changes simply to make a build reproducible [10]. We propose patching packaging tools to enable reproducible package builds. The patched tools can be found in the replication package. Our key insight is that *packaging tools can set default environments and post-process dynamic metadata to ensure build determinism.*

As shown in Figure 3, these patches have a drastic impact on the package build reproducibility of PyPI, increasing the percentage of reproducible builds from 12% to 98%. As suggested by the `reprotest` variation data shown in Figure 4, most of this increase was the result of addressing umask determinism. RubyGems also received a meaningful impact, increasing the percentage of reproducible builds from 97% to 99.9%. While this increase is small, the ability to achieve almost 100% reproducible package builds is extremely valuable for the ecosystem.

Takeaway: Package managers can set default environments and post-process dynamic metadata to provide reproducible package builds without changing code in individual packages.

3) Unreproducible Packages: As shown in Figure 3, not all package builds could be made reproducible by configuring package infrastructure or patching the package managers. We randomly sampled some of these packages to investigate causes of reproducibility issues.

Recall that PyPI has multiple backends to produce a package, and these backends use either a `setup.py` or a `pyproject.toml` specification file to interface with the build process. Developers using `setup.py` can run arbitrary code during the build. Reproducibility issues caused by this code are not easily addressable. For example, statements using the `os.path.expanduser` function make the build unreproducible because of the `home` variable. This kind of actions can hardly be managed by the build infrastructure because: (i) the infrastructure cannot alter external libraries, such as the `os` library in our case, and (ii) the package's developer set up an ad-hoc approach, interfering with it can easily break the build process. In contrast, using `pyproject.toml` should address many of the problems caused by dynamic metadata. However, some build backends, e.g., `poetry`, allow developers to call pre-build scripts, declaring them inside of the specification file. These scripts may cause unreproducible builds and cannot be easily patched at the tooling level. We found that 1.6% of the builds for PyPI in our results are affected by these kinds of issues.

RubyGems is subject to similar issues. The cause of unreproducible builds is the run of arbitrary commands launched through the specification file. For example, the package version tag can be programmatically set in the specification file by using Ruby Time functions — e.g., `VERSION = "0.0.#{Time.now.to_i}"`. The Ruby Time library does not use the `SOURCE_DATE_EPOCH` variable, making time a reproducibility issue. This event shows how this kind of behavior can be dangerous for reproducibility and how it can hardly be addressed through the build infrastructure.

Maven's challenges are different. Section IV-B1 discussed that plugins need to support the `outputTimestamp` property declared in the POM.xml specification file. Older plugin versions and custom plugins may not use it. This requires a developer to carefully inspect the plugin used in the package build pipeline since, depending on the pipeline design, a single plugin can impact the build reproducibility [40]. We found that 7% of the builds for Maven are affected by these kinds of issues.

C. RQ3: Native Code Extensions

As discussed in Section II, PyPI, npm, and RubyGems allow packages to include native code. This subsection studies how the inclusion of native code impacts the reproducibility of package builds (RQ2). Considering the original dataset where packages are not distinguished based on native code extensions, the incidence of native code extensions is 9.96% for PyPI, 9.85% for RubyGems, and just 3.24% for npm. Due to the low relative infrequency of native code in these package ecosystems, as discussed in Section III-B, we created an additional dataset that exclusively contain packages with native code. Our comparison uses infrastructure configuration (Section IV-B1) and our packaging tool patches (Section IV-B2) to isolate the impact of native code extensions.

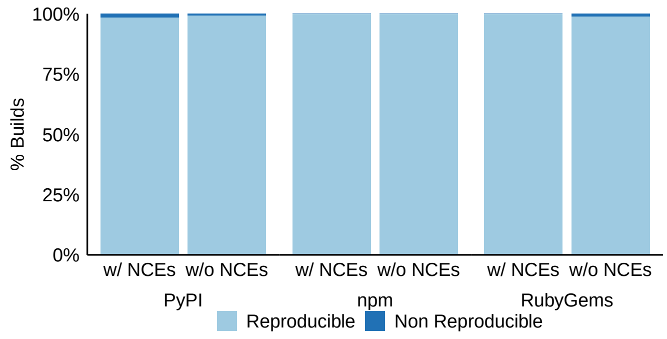

Fig. 5. Reproducible package builds that take into account extensions for native code extensions (NCEs). In order to concentrate on reproducibility issues caused by native code extensions, these findings are obtained using patched package managers.

Overall we find that, although native code extensions are a more or less common feature in interpreted languages, reproducible builds appear to be unaffected by them.

1) Quantitative Results: Comparing the reproducibility of package builds of the dataset without native code extensions to the dataset with native code extensions, in Figure 5, we find that the difference between the two datasets negligible: PyPI has 98.43% reproducible builds with native code and 99.28% without, npm has 100% with native code and 100% without, and RubyGems has 99.94% with native code and 98.87% without.

Unreproducible builds are caused by the same reproducibility issues discussed in Section IV-B3. In the end, npm, PyPI, and RubyGems show the same reproducibility issues independent of the presence of native code extensions. Using patched package managers solves those issues as we saw in the previous section. We hence conclude that native code extensions do not impact the build's reproducibility.

2) Causes of Unreproducibility: Since C code has multiple reproducibility issues that can affect the build process, we originally expected native code extensions to affect the reproducibility of the package build. Since we do not have any specific pointer to the causes of reproducibility in the analyzed ecosystems, we systematically searched for common reproducibility issues in the C code in 1000 packages randomly picked in the sample, e.g., time-dependent macros. We did not find any occurrence of such issues from this search. Note that our analysis used the same build configuration and tools. If no assumptions are made regarding system configurations, such as compiler specs, there will be fewer reproducible builds. Build specification files (e.g., `.buildinfo` files) are not incorporated into the build processes of any of these three ecosystems.

In PyPI, the Meson backend is largely adopted to compile native code extensions. Meson states to achieve reproducible builds[8] by addressing multiple issues, e.g., rpaths. In npm, the gyp module is similarly utilized for managing native code. This additional layer in compilation can be the reason behind the high number of reproducible builds. However, it does

[8]https://github.com/mesonbuild/meson

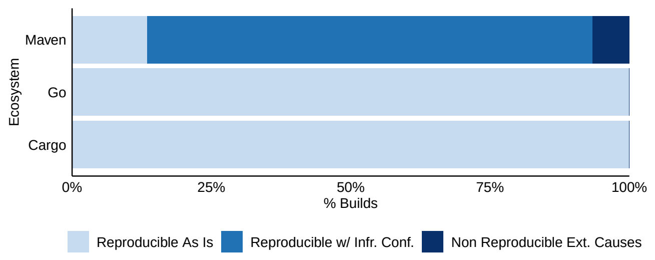

Fig. 6. Reproducible Builds obtained by compilation. We used unpatched package managers since the compilation process requires different configurations w.r.t. packaging.

not justify the almost complete reproducibility of builds. In RubyGems the system default compiler is directly invoked to compile extensions. Even without additional layer, there is a high number of reproducible builds.

It is difficult to completely understand the reasons why native code extensions in builds rarely affect reproducibility. However, it is possible to connect it to two main factors. First, native code extensions usually deal with specific tasks with a limited amount of code [41], reducing the chances of an reproducibility issue. Second, native code extensions require more expertise to be set up and properly work than regular code. Developers with higher skills in programming are possibly more aware of and invested in reproducible builds [25], [10].

Takeaway: Reproducible builds are not affected by the presence of native code extensions in the package. This means that reproducible builds could be easily achieved by solving issues related to the system configurations, such as the compiler specs.

D. RQ4: Reproducible Builds and Compilation

Some packaging ecosystems allow for packaging dependencies together with the developed code in a binary: The dependencies are inserted into the build artifact during the build process. We limited this analysis to Cargo, Go, and Maven, as they are the only ecosystems implementing compilation in their official package managers.

1) Quantitative Results: We find, as shown in Figure 6, that Cargo, Go, and Maven builds are still reproducible after the dependencies have been inserted into a reproducible application. That is, compiling dependencies into the binary in the build process does not introduce reproducibility issues. Cargo, Go, and Maven builds are almost completely reproducible, with 100%, 100%, and 92.5%, respectively. For Maven, the results were obtained using the package manager with the `outputTimestamp` property configured.

2) Causes of Unreproducibility: For Cargo and Go we randomly sampled 1000 packages to understand which are the causes of unreproducible builds, and searched for common reproducibility issues in those packages' code. For Maven, we checked whether packages with unreproducible builds were also unreproducible against packaging to understand if any specific reproducibility issue is caused by the compilation process. All three ecosystems have initiatives working towards reproducible builds that can explain the high rates of reproducibility we find.

Cargo developers explicitly label reproducibility issues in their GitHub repository[9] and have addressed several issues in the past, such as stripping absolute paths – a well-known issue for compilation. This deliberate effort towards reproducibility made `rustc` (the Rust compiler used by Cargo) almost deterministic, apart from system specifications such as the linker and compiler versions.

Also Go takes care to remove potential causes of reproducibility issues during compilation, for example, by trimming absolute paths and by removing build identifiers. The many issues and community discussions on reproducible builds within Go highlight the interest of this community.[10] Moreover, in order to facilitate reproducible builds across different systems, Go strives to establish reproducible builds for the toolchain [42].

Javac (the Java compiler) used in Maven is known to be deterministic [43]. Our results confirm this. The reproducibility issues in Maven packages are caused by archive metadata since the compiled Java classes are inserted into a Jar archive, as happens for packaging. Maven builds are affected by the same reproducibility issues discussed in Section IV-B1, affecting archive metadata and not the compiled artifacts. As for packaging, configuring the build infrastructure addresses most of the unreproducible builds. The remaining unreproducible builds are caused by the same reproducibility issues discussed in Section IV-B3.

Takeaway: Build reproducibility is unaffected by compilation. Build infrastructures do not introduce reproducibility issues thanks to the community efforts working on them.

V. Discussion and Recommendations

Reproducible builds are a notoriously hard problem. Few developers pay active attention to reproducible builds and those that do usually face a laborious, tedious, and incremental process [10]. With software being usually constructed using many reusable components, ensuring reproducible builds for components is an important building block for the community to move toward reproducible builds for applications and infrastructure.

Our results can be interpreted in two ways:

- On the negative side, given that components in many ecosystems are distributed as archives of source code, often with little build-time processing, it is shocking/frustrating how few components are packaged in a repro-

[9]https://github.com/rust-lang/rust/labels/A-reproducibility

[10]https://github.com/golang/go/issues?q=is:issue+is:open+"reproducible+builds"

ducible way. Almost no RubyGems, PyPI and Maven packages are reproducible.

- On the positive side, most reproducibility issues are rooted in the packaging tools of the ecosystem and can be addressed with small changes to the default settings of those tools. In contrast to usual discussions of reproducible builds for applications, only very few components in our study had reproducibility issues that cannot easily be addressed, even for components with native code extensions. This signals that reproducible builds for components are within easy reach.

Recommendations to practitioners. After patching packaging tools, the remaining unreproducible builds in our study were caused by ad-hoc solutions applied by developers. We recommend that developers avoid extending specification files with custom scripts, even if the ecosystem permits such behavior. Relying on custom configurations can easily compromise reproducibility, as we have seen (Section IV-B3).

Recommendations to maintainers of packaging tools. We make three recommendations: (1) secure defaults, (2) build information, and (3) warning when unreproducible. First, in line with past work on usable security [44], [45], [46], we emphasize the importance of secure defaults – in this case, defaults that eliminate common reproducibility issues. Reaching every single developer is inefficient and hopeless, but with simple changes to the packaging infrastructures most components will be packaged reproducibly without any manual effort from developers. Second, packaging tools should automatically record information that enables build reproducibility. This includes the used compiler and version, the exact versions of build dependencies, and build options. Third, packaging tools should issue warnings if packaging steps rely on undeclared infrastructure (e.g., JavaScript transpilers installed on the local computer but not declared as a dependency) and nondeterminism in builds (e.g., by explicitly running reprotest-like experiments during the build).

Recommendations to researchers. Most of the findings of this paper enable further research. Our aim was to provide a comprehensive study of reproducible builds in different ecosystems. While easily achievable, reproducible builds can be compromised by additional issues that require more examination. Further research may also consider developing methods that recreate build information for published packages as a stop-gap solution until package maintainers include this information by default. Recently, AROMA [26] proposed such an approach for Java packages in Maven.

VI. Conclusion

Overall, *our study results make us optimistic* about the future of reproducible builds. For software components, reproducible builds are much closer than we expected, and a few small changes to infrastructure tools have a large lever to get the community to a future where most packages are fully reproducible. Barriers to ensuring that nearly all published components are exactly reproducible from the public source code are minimal. We can focus our attention more on other last-mile issues, such as linking published components on package managers to specific commits of their publicly available source code, ensuring the provenance of code changes to the public source code [47], and ensuring that packages are consumed in a reproducible manner when building applications (e.g., locking floating dependencies) [48].

Acknowledgments

Solarin's work was supported through an NSF REU site (#2150217). Miller's and Kästner's work was supported in part by the National Science Foundation (#2206859). Tystahl's, Enck's, and Kapravelos's work was supported in part by the National Science Foundation (#2207008).

References

[1] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, pp. 761—-763, 1984.

[2] T. Herr, W. Loomis, E. Schroeder, S. Scott, S. Handler, and T. Zuo, *Broken trust: lessons from Sunburst*. Washington, DC: Atlantic Council, 2021.

[3] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1509–1526. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00010

[4] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, Mar. 2022.

[5] M. Perry, "Deterministic builds part two: Technical details," https://blog.torproject.org/deterministic-builds-part-two-technical-details/, Oct. 2013.

[6] devrandom, "Gitian: a secure software distribution method," https://github.com/devrandom/gitian-builder, 2011.

[7] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds," in *Proceedings of the 26th USENIX Security Symposium (Sec'17)*, Aug. 2017, pp. 1271–1287.

[8] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, "Lastpymile: identifying the discrepancy between sources and packages," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 780–792. [Online]. Available: https://doi.org/10.1145/3468264.3468592

[9] H. Levsen *et al.*, "Reproducible Debian overview," https://tests.reproducible-builds.org/debian/reproducible.html, 2023.

[10] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar, "It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2023, pp. 1527–1544.

[11] C. Cimpanu, "Backdoored python library caught stealing ssh credentials," https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, May 2018.

[12] D. Goodin, "What we know about the xz utils backdoor that almost infected the world," ars Technica, Apr. 2024, https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/.

[13] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, "Investigating the reproducibility of npm packages." Institute of Electrical and Electronics Engineers Inc., 9 2020, pp. 677–681.

[14] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3447245

[15] K. Manikas, "Revisiting software ecosystems research: A longitudinal literature study," *Journal of Systems and Software*, vol. 117, pp. 84–103, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216000406

[16] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381–416, 2 2019.

[17] H. Levsen, C. Lamb, and M. Rizzolo, "reprotest," https://salsa.debian.org/reproducible-builds/reprotest, 2016. [Online]. Available: https://salsa.debian.org/reproducible-builds/reprotest

[18] G. Benedetti, O. Solarin, C. Miller, G. Tystahl, W. Enck, C. Kästner, A. Kapravelos, A. Merlo, and L. Verderame, "Replication package: An empirical study on reproducible packaging in open-source ecosystems." [Online]. Available: https://osf.io/vmnsh/?view_only=c8e26e10cbf145839bfa6820de76792d

[19] I. van den Berk, S. Jansen, and L. Luinenburg, "Software ecosystems: a software ecosystem strategy assessment model," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 127–134. [Online]. Available: https://doi.org/10.1145/1842752.1842781

[20] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1513–1531. [Online]. Available: https://doi.org/10.1145/3372297.3417232

[21] reproducible-builds.org, "SOURCE_DATE_EPOCH standard," https://reproducible-builds.org/specs/source-date-epoch/, 2020.

[22] ——, "diffoscope in-depth comparison of files, archives, and directories." https://diffoscope.org/, 2014. [Online]. Available: https://diffoscope.org/

[23] Y. Shi, M. Wen, F. R. Cogo, B. Chen, and Z. M. Jiang, "An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3361–3377, Sep. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9465650/

[24] X. De Carné De Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference*. New Orleans Louisiana USA: ACM, Dec. 2014, pp. 16–25. [Online]. Available: https://dl.acm.org/doi/10.1145/2664243.2664288

[25] S. Butler, J. Gamalielsson, B. Lundell, C. Brax, A. Mattsson, T. Gustavsson, J. Feist, B. Kvarnström, and E. Lönroth, "On business adoption and use of reproducible builds for open and closed source software," *Software Quality Journal*, vol. 31, no. 3, pp. 687–719, Sep. 2023. [Online]. Available: https://link.springer.com/10.1007/s11219-022-09607-z

[26] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch, "Aroma: Automatic reproduction of maven artifacts," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3643764

[27] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, "Options matter: Documenting and fixing non-reproducible builds in highly-configurable systems," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 654–664. [Online]. Available: https://doi.org/10.1145/3643991.3644913

[28] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 71–81. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3180224

[29] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, Nov. 2019, pp. 527–538. [Online]. Available: https://ieeexplore.ieee.org/document/8952375/

[30] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang, "Automated patching for unreproducible builds," in *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 200–211. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510102

[31] A. Decan, T. Mens, M. Claes, and P. Grosjean, "On the Development and Distribution of R Packages: An Empirical Analysis of the R Ecosystem," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*. Dubrovnik Cavtat Croatia: ACM, Sep. 2015, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/2797433.2797476

[32] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Klagenfurt, Austria: IEEE, Feb. 2017, pp. 2–12. [Online]. Available: http://ieeexplore.ieee.org/document/7884604/

[33] ——, "On the topology of package dependency networks: a comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*. Copenhagen Denmark: ACM, Nov. 2016, pp. 1–4. [Online]. Available: https://dl.acm.org/doi/10.1145/2993412.3003382

[34] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita: IEEE, Mar. 2016, pp. 493–504. [Online]. Available: http://ieeexplore.ieee.org/document/7476669/

[35] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative Inquiry*, vol. 12, no. 2, pp. 219–245, 2006. [Online]. Available: https://doi.org/10.1177/1077800405284363

[36] J. F. Shobe, M. Y. Karim, M. B. Zanjani, and H. Kagdi, "On mapping releases to commits in open source systems," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 68–71. [Online]. Available: https://doi.org/10.1145/2597008.2597792

[37] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 14–14.

[38] H. Levsen, "Reproducible builds - rebuilding what is distributed from ftp.debian.org," MiniDebConf Toulouse, Nov. 2024. [Online]. Available: https://toulouse2024.mini.debconf.org/talks/4-reproducible-builds-rebuilding-what-is-distributed-from-ftpdebianorg/

[39] Debian, "Buildinfofiles," https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles, 2018. [Online]. Available: https://wiki.debian.org/ReproducibleBuilds/BuildinfoFiles

[40] A. Maven, "Configuring for reproducible builds," https://maven.apache.org/guides/mini/guide-reproducible-builds.html, 2024.

[41] R. Monat, A. Ouadjaout, and A. Miné, "A Multilanguage Static Analysis of Python Programs with Native C Extensions," in *Static Analysis*, C. Drăgoi, S. Mukherjee, and K. Namjoshi, Eds. Cham: Springer International Publishing, 2021, vol. 12913, pp. 323–345, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-88806-0_16

[42] R. Cox, "Perfectly reproducible, verified go toolchains," https://go.dev/blog/rebuild, 2023.

[43] Compiler-dev, "Javac determinism," https://mail.openjdk.org/pipermail/compiler-dev/2023-December/025215.html, 2023.

[44] B. Gates, "Memo from bill gates," https://news.microsoft.com/2012/01/11/memo-from-bill-gates/, 2012.

[45] M. Stanek, "Secure by default - the case of tls," 2017.

[46] P. Gorski, L. Lo Iacono, S. Wiefling, and S. Möller, "Warn if secure or how to deal with security by default in software development?" 08 2018.

[47] G. Rousseau, R. Di Cosmo, and S. Zacchiroli, "Software provenance tracking at the scale of public source code," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2930–2959, Jul. 2020. [Online]. Available: https://link.springer.com/10.1007/s10664-020-09828-5

[48] A. Gaurav, "Enable repeatable package restores using a lock file," https://devblogs.microsoft.com/nuget/enable-repeatable-package-restores-using-a-lock-file/, 2018.