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

Unreproducible builds: Time to fix, causes, and correlation with external ecosystem factors

Rahul Bajaj · Eduardo Fernandes · Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract *Context:* A reproducible build occurs if, given the same source code, build instructions, and build environment (i.e., installed build dependencies), compiling a software project repeatedly generates the same build artifacts. Reproducible builds are essential to identify tampering attempts responsible for supply chain attacks, with most of the research on reproducible builds considering build reproducibility as a project-specific issue. In contrast, modern software projects are part of a larger ecosystem and depend on dozens of other projects, which begs the question of to what extent build reproducibility of a project is the responsibility of that project or perhaps something forced on it. *Objective:* This empirical study aims at analyzing reproducible and unreproducible builds in Linux Distributions to systematically investigate the process of making builds reproducible in open-source distributions. Our study targets build performed on 11,528 and 597,066 Arch Linux and Debian packages, respectively. *Method:* We compute the likelihood of unreproducible packages becoming reproducible (and vice versa) and identify the root causes behind unreproducible builds. Finally, we compute the correlation between the reproducibility status of packages and three ecosystem factors (i.e., factors outside the control of a given package). *Results:* Arch Linux packages become reproducible a median of 30 days quicker when compared to Debian packages, while Debian packages remain reproducible for a median of 68 days longer once fixed. We identified a taxonomy of 16 root causes of unreproducible builds and found that the build reproducibility status of a package across different

R. Bajaj
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: r.bajaj@queensu.ca

E. Fernandes
The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense, Denmark
E-mail: edmf@mmmi.sdu.dk

B. Adams
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: bram.adams@queensu.ca

A.E. Hassan
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: hassan@queensu.ca

hardware architectures is statistically significantly different (strong effect size). At the same time, the status also differs between versions of a package for different distributions and depends on the build reproducibility of a package's build dependencies, albeit with weaker effect sizes. *Conclusions:* The ecosystem a project belongs to, plays an important role w.r.t. the project's build reproducibility. Since these are outside a developer's control, future work on (fixing) unreproducible builds should consider these ecosystem influences.

Keywords Reproducible build · Supply chain attack · Build environment · Software package · Release management · Software security

1 Introduction

Most open-source software (OSS) projects are distributed as part of larger software ecosystems, such as Linux distributions. Such distributions ship OSS projects in the form of packages [19,45] that other developers can build on (reuse) [2]. Popular Linux distributions such as Arch Linux and Debian [2] consist of thousands of inter-dependent packages, thereby forming a *software supply chain*.

Despite its benefits, ensuring the security of a supply chain while reusing packages may be particularly challenging. Even with the best effort of developers in following well-established security principles, just by accidentally updating to a tampered version of a dependency (either manually or automatically), any software project can be prone to supply chain attacks. For instance, recent supply chain attacks such as SolarWinds [22] and Mimecast[1] have affected thousands of users and caused significant financial losses. Supply chain attacks have been observed even in the context of significant software organizations, such as Apple, PayPal, and Shopify.[2]

As a means to increase the integrity of supply chains in terms of security, the software industry has been gradually adopting reproducible builds processes [17]. A reproducible build occurs if given the same source code, build instructions, and build environment (i.e., installed build dependencies), compiling a software project, bit-by-bit generates the same build artifacts. Reproducible builds are expected to prevent supply chain attacks by ensuring that the binaries provided to the software customers are equivalent to the binaries of other customers and the binaries provided by the developers.

The Reproducible Builds initiative is an initiative that promotes the importance and best practices of reproducible builds, as well as provides tools to automatically scan a given project in terms of the reproducibility of its build process. This initiative has been adopted by 33 OSS projects[3] such as Arch Linux and Debian. For example, in the case of Debian, builds are performed on each source package[4] in two distinct build environments[5] with similar source code, build instructions, and build environment. Next, the build artifacts (i.e., software binaries) generated from each independent build are compared bit by bit. If the build

[1] `https://www.mimecast.com/blog/important-update-from-mimecast/`

[2] `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`

[3] `https://reproducible-builds.org/who/projects/`

[4] A source package like `glibc`, when built can produce multiple binary packages like `libc6` and `libc6-dev`.

[5] `https://tests.reproducible-builds.org/debian/index_variations.html`

artifacts are equivalent, the reproducibility status of the package is said to be reproducible. Otherwise, the status is said to be unreproducible, and the package might be prone to supply chain attacks.

Full reproducibility for a project is a challenging endeavor, possibly impacted by many factors, such as the nature of release cycles [26]. In cases where release cycles are longer, it becomes harder to publish patches and address issues promptly. On the other hand, shorter release cycles, although facilitating faster development, can potentially lead to regressions. The focus within such time-constrained scenarios is primarily directed towards adding or enhancing software features, often at the expense of ensuring reproducible builds [20]. It is evident that stakeholders prioritize the swift development and expansion of functionalities, with reproducibility taking a backseat.

Nevertheless, empirical knowledge on how much time it takes for an unreproducible package to become reproducible (and vice versa) is nonexistent. Additionally, the current knowledge on possible root causes of unreproducible builds [17] is based on anecdotal experience rather than empirical evidence. Finally, a central unanswered question is the extent to which the build reproducibility of a package is only the project's "fault" or whether a build process can be forced to be unreproducible due to other projects in the ecosystem. For instance, if a project's build depends on another project whose build is unreproducible, could the former build still be reproducible? We refer to this and other ecosystem factors as external factors because they are external to the project's own community.

In this paper, we introduce a mixed-methods empirical study [11] aimed at analyzing reproducible and unreproducible builds in OSS projects. We carefully conducted quantitative and qualitative analyses targeting the Arch Linux[6] and Debian[7] distributions, which share a considerable number of common packages. Overall, we analyze 11,528 Arch Linux packages and 597,066 Debian packages in three phases. First, we perform survival analysis [3] to investigate the probability of unreproducible packages becoming reproducible (and vice versa) over time. Second, we build on an initial list of six root causes developed by Lamb et al. [17] to perform a manual labeling [41] of 396 issues related to reproducible builds to expand our current knowledge on possible root causes of unreproducible builds, as well as to compute their frequency and impact across multiple packages. Our goal is to assist developers in reasoning about the root causes before fixing unreproducible packages. Third, we compute the χ^2 correlation [33] between three external ecosystem factors and the reproducibility status of packages: architecture, build dependencies, and distributions.

As a complement, we also re-execute our three analyses mentioned above per package domain (sections)[8] in Debian to understand if the survival analysis, root causes, and correlations vary depending on the domain to which a package belongs, e.g., *devel* (development utilities) vs. *math* (mathematical computations). Inspired by previous work [1], we manually classify each package domain as either system-level, i.e., when a package domain covers essential functionalities of Debian, or application-level, i.e., when a package domain covers user-facing applications and utilities. We summarize below our study results and their implications.

[6] `https://reproducible.archlinux.org/`

[7] `https://wiki.debian.org/ReproducibleBuilds`

[8] `https://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections`

- The **survival analysis** reveals that initially unreproducible Arch Linux packages are fixed a median of 30 days quicker than Debian packages. On the other hand, once packages became reproducible, Debian packages remained reproducible for a median of 68 days longer when compared to Arch Linux packages. In summary, developers need assistance to act quicker in fixing unreproducible packages, thereby preventing supply chain attacks that might occur after the packages are built unreproducible.
- The **manual analysis** expands an initial list of six root causes of unreproducible packages developed by Lamb et al. [17] to a taxonomy of 16 root causes. Build Timestamp has the highest number of issues reported (155), while Build Path is the most influential one (5,008 packages affected) – five times more influential than the second one, i.e., Randomness (837 packages affected). The fact that Build Timestamp is less of an issue than in the past [17,23] shows that resolutions for that issue have had a positive impact, and that a wider range of (possibly unsolved) build reproducibility issues should be expected in practice. Our taxonomy results have garnered interest from the reproducible builds community, leading to their inclusion on the official website for reproducible builds documentation.[9][10]
- The **correlation analysis** reveals that the reproducibility status of packages depends strongly on the hardware architecture supported by the distribution. Therefore, all packages must be built differently to run on specific hardware architectures. Additionally, build reproducibility varies with external factors like the build reproducibility of a package's build dependencies (weak effect size). Finally, the distribution on which builds are performed has a negligible correlation with the reproducibility status of packages. This indicates that developers should build packages on all distributions taken into consideration to determine whether or not the package will build reproducibly.
- The **domain-oriented analysis** suggests that initially unreproducible packages of application-level package domains have a survival probability 12.9% higher than packages of the system-level domains. However, system-level packages usually become unreproducible again quicker than application-level ones, implying that they are more likely to be affected by supply chain attacks for longer. Additionally, 18.46% of Debian packages are impacted by multiple root causes, which might make these packages harder to fix. Overall, we suggest that developers must prioritize packages from system-level domains. Remarkably, the *devel* domain significantly impacts the package reproducibility based on the weak to strong correlation with all three external factors – hardware architecture (strong), distribution (moderate) and build dependencies (weak). Since packages in the *devel* domain consist of compilers and libraries, these packages implement core functionalities of the system.

We organized the remainder of this paper as follows. Section 2 provides case study data information for our work. Section 3 introduces the study methodology, including the research questions and study steps. Sections 4, 5, and 6 presents our study results for each research question. Section 7 discusses threats to the study's validity. Section 8 emphasizes the novelty of our study compared to the existing literature on the subject matter. Section 9 concludes this paper and suggests future

[9] `https://reproducible-builds.org/docs/plans/`
[10] `https://reproducible-builds.org/docs/env-variations/`

work. Finally, we provide the supplementary material[11] of our study, including the data collection, data processing and filtering scripts, and data spreadsheets.

2 Case Study Data

2.1 Selected Linux Distributions

Six Linux distributions adopted the Reproducible Builds initiative at the time of data collection in October 2021: Alpine, Arch Linux, Debian, Fedora, FreeBSD, and OpenSUSE. The Reproducible Builds [12] initiative publishes build data for each distribution package that constitutes a project. Builds are triggered by Jenkins servers[13] and the build results are logged into a publicly available database.[14] These results serve as a valuable resource for tracking the reproducibility status, enabling the assessment of whether it is improving, worsening, or remaining unchanged for the given project.[15][16]

We captured the database on October 11, 2021, and all data was stored in comma-separated values (CSV) format. However, only three of the six projects' build data is publicly available: Alpine, Arch Linux, and Debian. Considering that we aimed at performing survival analysis (see Section 3.2) for a two-year time frame, we discarded Alpine because it had less than two years of build data available.

Arch Linux and Debian employ a software package-based distribution model, whereby software packages are synchronized only when a verified official version is released upstream, not with every new upstream commit. These packages are versioned over time and organized into multiple suites. In the case of Debian, packages are organized in suites based on their production-level stability assessment, e.g., development-phase or stable. The Experimental and Unstable suites serve as testing grounds for new packages before they are deemed fit for the Stable suite. Tested packages are promoted to become part of the next stable release series, which typically occurs periodically, e.g., every two years.[17] On the other hand, Arch Linux package are closely tied to software repositories that serve as categories, each with its own purpose, e.g., the Core repository contains essential packages for booting the operating system.[18] Table 1 introduces the four Arch Linux suites, i.e., Core, Community, Extra, and Multilib, and four Debian suites – i.e., Experimental, Stable, Testing, and Unstable.

In the particular case of Arch Linux, each suite corresponds to a software repository, as Arch Linux is not versioned in terms of major releases. Instead, each package in the software repository is upgraded over time to the latest available

[11] `https://github.com/SAILResearch/replication-21-rahul_bajaj-reproducible_builds-code`
[12] `https://reproducible-builds.org/`
[13] `https://jenkins.debian.net/userContent/about.html#_reproducible_builds_jobs`
[14] `https://tests.reproducible-builds.org/reproducible.sql.xz`
[15] `https://tests.reproducible-builds.org/debian/reproducible.html`
[16] `https://tests.reproducible-builds.org/archlinux/archlinux.html`
[17] `https://wiki.debian.org/DebianReleases?action=show&redirect=DebianRelease#Introduction`
[18] `https://wiki.archlinux.org/title/official_repositories`

Table 1: Debian Releases and Suites

Distribution	Suite	Description	Releases
Arch Linux	Core	Packages required for performing essential tasks, e.g. booting Arch Linux, building packages, and repairing file systems.	N/A
	Community	Packages created and maintained by trusted users of the Arch Linux community. These packages might be promoted to the Core repository.	N/A
	Extra	Akin to the Experimental suite in Debian, the Extra repository consists of packages that are currently not suitable for being placed in the core repository.	N/A
	Multilib	The Multilib repository consists of packages built on 32 bit architecture but are also compatible with 64-bit architecture.	N/A
	Arch Full	Refers to the entirety of the Arch Linux distribution, encompassing all its main software repositories, i.e., Core, Extra, and Community.	N/A
Debian	Experimental	Packages that are not fit to become Unstable.	`experimental`
	Stable	Packages distributed to users. Each major release receives a codename, e.g. `bullseye` for Debian 11	`bookworm` (Debian 12) `bullseye` (Debian 11). `buster` (Debian 10) `stretch` (Debian 9)
	Testing	Packages in the process of becoming Stable.	`testing`
	Unstable	Packages on which active upstream development contributions have been made. Before becoming Stable, these packages are submitted to Testing.	`sid` or `unstable`

version while the old version is removed from the repository. This release strategy allows Arch Linux users to use the latest versions of packages available in the repository. Unlike Arch Linux, Debian is versioned in significant releases that are released periodically, e.g., every two years.[19]

In Debian, all packages undergoing active development changes are a part of the Unstable suite, which is thoroughly tested before being promoted to the Stable suite. The Unstable suite consists of a large number of packages (37,858 packages). Testing is the only suite not analyzed in this paper because, as a transition phase between Unstable and Stable, we lack build data for analysis due to its purpose of testing the packages.

In the remainder of this paper, unless explicitly stated otherwise, our comparison primarily focuses on Arch Full – representing the entirety of the Arch Linux distribution, which includes the Core, Extra, and Community suites – and Debian's Unstable suite. Note that Arch Full excludes the Multilib packages, which are tailored for 32-bit architecture, as our study is dedicated to packages developed for the `x86_64` architecture, a 64-bit system. This choice of comparison stems from the commonalities between the two: both represent the most current, rolling-release versions of their respective distributions. This ensures that we are comparing the

[19] `https://wiki.debian.org/DebianReleases`

most up-to-date and actively developed packages of these two influential Linux ecosystems.

2.2 Build

Table 2 presents the number of packages studied for the different suites and hardware architectures for the Arch Linux and Debian distributions. In the case of Arch Linux, all the suites are built on the `x86_64` architecture, while for Debian, all suites are built on four hardware architectures: `amd64`, `arm64`, `armf64`, and `i386`.

Table 2: Number of Packages by Suite of each Linux Distribution

Distribution	**Architecture**	**Suite**	**Number of Packages**
Arch Linux	`x86_64`	Community	8,529
		Extra	2,510
		Multilib	280
		Core	209
Debian	`amd64`	Unstable	37,858
		Bullseye	33,172
		Buster	30,547
		Stretch	27,003
		Bookworm	16,036
		Experimental	8,118
	`arm64`	Unstable	36,304
		Bullseye	32,971
		Buster	30,551
		Stretch	25,514
		Bookworm	14,465
		Experimental	5,802
	`armhf`	Unstable	37,323
		Bullseye	33,032
		Buster	30,518
		Stretch	26,194
		Bookworm	19,857
		Experimental	7,045
	`i386`	Unstable	36,940
		Bullseye	32,780
		Buster	30,538
		Stretch	26,062
		Bookworm	11,669
		Experimental	6,777

The downloaded CSV file with the Reproducible Builds database consists of 18,356,214 rows of build data. Each row in the dataset represents the build result of a particular package. *Architecture*, *build timestamp*, *package name*, *suite* and *status* are key columns in the dataset. The *Architecture* column specifies the supported hardware architecture on which a package is compiled.

The *package name* and *suite* columns refer to the package name and the release cycle from which the packages belong. The *build timestamp* column presents the time on which the build was performed. Finally, the *status* column has either of these values: reproducible, failed to build from source (FTBFS), dependency wait (depwait), failed to build reproducible (FTBR), error 404 (E404), and timeout. We considered all statuses other than reproducible as unreproducible since the package as a whole has a reproducibility issue.

2.3 Build Dependency Data

Performing builds on software repositories is far from trivial, as it involves identifying and installing all the necessary build dependencies, such as compilers, interpreters, and libraries. When available and used correctly [27], these build dependencies allow for turning a package's source code into the required software binaries, packaged in the distribution's package format. Problems with build dependencies (e.g., if one or more are missing) can lead to a defective binary [25]. In Debian, these dependencies are labeled as `Build-Depends:` in the control files.[20].

To investigate build dependencies, we make use of the `build_depends` column in the `all_sources` table of the Ultimate Debian Database (UDD) [30], which mirrors the `Build Depends:` in the control files found in Debian's source packages. The UDD contains comprehensive information pertaining to all Debian source packages, encompassing vital details such as package domain, package name, build dependencies, binary packages produced, and maintainer information.

2.4 Package Domain Data

The Debian community has categorized its source packages according to their domain (purpose).[21] For instance, the *devel* domain refers to development utilities, including compilers and libraries, while the *games* domain refers to libraries that enable playing games for users. We obtain the package domain data from the `section` column in the *all_sources* table of the Ultimate Debian Database (UDD) [30]. The *all_sources* table identifies 42,086 unique source packages, which are mapped to 59 package domains (excluding the non-free package domains) at the time of collecting the data. Out of the 38,064 unique source packages that are built to test for reproducibility only 956 (2.5%) unique packages identified in Debian, lack package domain information assigned to them.

Considering the functionality that packages in a domain perform, we manually classified the 59 package domains as either system-level (i.e., packages related to programming languages, development tools, and build systems, all of which play indispensable roles in software development and system maintenance) or application-level (i.e., packages catering to a wide array of end-user applications). We performed this manual classification to identify packages requiring prioritized fixes compared to other packages.

Table 3 presents 14 of the 59 package domains discussed throughout the paper. The first column categorizes the package domains as either system-level or application-level, while the second column lists the ID for each package domain as presented on the Debian website.[22] The third and fourth columns list and describe the purpose behind each package domain. Finally, the fifth column presents some examples of packages by domain. Examples of system-level packages include `grub-core-boot-bin` (*admin* domain), i.e., a boot loader for choosing which operating system to boot when multiple options are available, and `2ping` (*net* domain), which identifies established connections and determines the directional packet loss

[20] `https://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps`

[21] `https://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections`

[22] `https://packages.debian.org/unstable/`

from the connection initiated. Examples of application-level packages are `calc` and `gnuplot` (both from *math* domain), which are essential to perform tasks such as complex mathematical computations and graph plotting the results, respectively. Note that we do not have package domain information for Arch Linux, only for Debian.

Table 3: Package Domains in Debian

Type	ID	Purpose	#Source Packages	Examples of Packages
System-level	*admin*	Manage system resources, user accounts, deployments, configurations, and maintenance	653	adduser cron puppet
	devel	Software development support: build systems, compilers, environments, and libraries	670	automake autotools-dev cmake
	interpreter	Runtime libraries for programming languages such as Python, Ruby and Perl	147	erlang pycurl pyopenssl
	net	Daemon and client utilities that enable the system to communicate with the Internet	947	2ping netstat-nat samba
	perl	Perl software development support: tools, libraries, etc.	2705	carton dh-make-perl
	ruby	Ruby software development support: tools, libraries, etc.	307	bundler cucumber jruby
	utils	General purpose: file or disk manipulation, backup and archive tools, and system monitoring	964	dracut grep sed
Application-level	*fonts*	Text font creation and editing	242	fonts-freefont unifont
	games	Gaming on the Debian distribution support	613	0ad 3dchess fortunes-mario
	gnome	Desktop environment and utilities for user-friendly interactions	270	gnome-bluetooth gnome-menus gnome-panel
	graphics	Tools and libraries for image processing, visualization, and graphical design	349	nip2 pixelize pngtools
	math	Mathematical computations support	212	calc euler gnuplot
	science	Basic scientific tasks support, such as astronomical data processing and performing bitwise operations	475	chemical-structures morse-simulator pyoptical
	sound	Sound processing: mixers, players, recorders, and CD players	531	audiolink beep easytag

3 Study Methodology

3.1 Research Questions

RQ1: *What is the probability of an unreproducible package becoming a reproducible package (and vice versa)?* – Turning a package's build reproducible can require many changes to its build system. Thus far, there is no empirical evidence of the amount of maintenance effort required by the developers to fix an unreproducible package for future builds. However, once a package can be built reproducibly, changes performed to the package source code or the build environment, in principle, could render packages unreproducible again. Therefore, it is vital to understand how likely build reproducibility can regress and how quickly reproducible packages become unreproducible again. This analysis highlights the extent to which developers must frequently monitor packages' reproducibility status.

RQ2: *What are the root causes of packages being unreproducible?* – We identify two main objectives for RQ2. The first objective is to systematically expand an initial list of possible root causes of unreproducible builds reported by previous literature [17]. The second objective is to understand the frequency of each root cause leading to unreproducible builds by verifying whether the root causes presented by previous literature (relying on an expert's perspective) match the empirical data. Finally, we compute the number of affected packages for each root cause to reveal the overall impact of each root cause on the packages. This information provides insights into the possibility of packages being affected by multiple root causes, which might be the reason for introducing supply chain attacks.

RQ3: *What external factors correlate with the reproducibility status of packages?* – Thus far, the literature [14, 17, 23] considered build reproducibility specific to an individual project. However, in the ecosystem settings of open-source distributions, there are strong socio-technical dependencies between packages [9]. Similar to how a package might "inherit" a vulnerability by depending on a vulnerable library package [44], theoretically, a package's build could be unreproducible because a package it depends on at build time is not reproducible. In other words, the reproducibility status of a package might not be fully controllable by an individual project; instead, it might be impacted by external software ecosystem factors. This is an example of the three ecosystem-related external factors potentially impacting a package's build reproducibility that we analyze in this RQ. The three external factors are inspired by previous work on software ecosystems [9]: the *distribution* in which a package is released; the *build dependencies* of a package; and the hardware *architecture* on which the package is compiled.

3.2 Survival Analysis Steps (RQ1)

We conducted survival analysis [3] to examine the duration until an unreproducible package becomes reproducible (and vice versa). Consistent with the discussion in Section 2.1, any reproducibility status other than "reproducible" was classified as unreproducible. Survival analysis [3, 10] constitutes a statistical methodology aimed at determining the time until the occurrence of a specific event while taking into account so-called censoring, i.e., data points for which the event of interest did not occur during the observed time window. By using this approach, we estimate the duration until the event of interest takes place and investigate the factors influencing its progression. In one of our scenarios, we define the event as the build achieving reproducibility, while in the other scenario, we define the event as a reproducible build reverting to an unreproducible state.

Survival analysis allows for the estimation of survival functions [10], including the probability of a build remaining unreproducible over time, thereby providing valuable insights into the risk or rate of transitioning to reproducibility. The survival analysis provides the probability $P(T > t)$ of a subject surviving beyond time t. Notably, the survival probability of a subject at time $t = 0$ is 1; however, at the limit of $t = \infty$, the subject's probability of survival diminishes to zero. Survival analysis inherently accounts for censoring, which ensures that all data, even if not fully observed, contributes meaningfully to the analysis, making the method robust and comprehensive. In the subsequent sections, we describe the five-step framework specifically designed to facilitate the analysis of Research Question 1 (RQ1).

3.2.1 Define the analysis

We propose two analyses to understand the time packages take to change their reproducibility statuses at the granularity of various suites across (2) distributions and (59) domains, respectively. To estimate the efforts required by the developers to fix unreproducible packages in the short term and to determine the efficacy of such efforts in the long term, for Analysis 1, we compute the survival probability of packages at both 30 days (1 month) and 360 days (1 year). For Analysis 2, we compute probability at 360 days to explore the reproducibility of packages in a one-year time frame. We describe below the procedures followed for executing each analysis.

Analysis 1: This analysis focuses on how quickly packages in different releases change their reproducibility statuses within 30 and 360 days. With the help of the survival probabilities obtained from all suites of both distributions (Arch Linux and Debian), we track the change in reproducibility statuses across these distributions. Our focus is predominantly on the Arch Full suite and the Debian Unstable suite, with the intent to compare their respective survival probabilities. This comparison, given the rolling release models adhered to by these suites, can be deemed equitable (see Section 2.1). Afterward, we investigate why specific suites have greater probabilities of becoming reproducible and remaining reproducible.

Analysis 2: Our aim for this analysis is to understand whether developers' efforts to make builds reproducible differ between package domains. Furthermore, this analysis focuses on how quickly packages from the system-level package domains become reproducible and remain in the same state compared to the packages of the application-level package domains.

3.2.2 Compute the reproducibility status of the packages

In our study, we define two such events of interest, i.e., **E1** and **E2**, and any packages not meeting **E1** and **E2** are considered censored for that corresponding analysis. For event **E1**, the event of interest is met when an initially unreproducible package becomes reproducible. For event **E2**, the event of interest is met when a reproducible package (for whom event **E1** has occurred) becomes unreproducible again. As such, the set of packages for which event **E2** occurred is a subset of the set of packages for which event **E1** occurred.

3.2.3 Filter packages

Before performing survival analysis, we first need to filter and cluster the package data.

Analysis 1: Table 4 presents the number of packages under analysis for each Linux distribution for Analysis 1. The first column distinguishes the distributions, while the second column discriminates the events of interest - **E1** (i.e., an unreproducible package becomes reproducible for the first time) and **E2** (i.e., a reproducible package becomes unreproducible again for the first time). The third column presents the total number of unique packages before filtering **E1** and **E2**. The fourth and fifth columns show the resulting packages after applying the two filters discussed below. The sixth column shows the number of censored packages, i.e., packages that do not perform the event in the given 720 days (2-year) time frame.

Table 4: Filtered Packages for Analyses 1 to Perform Survival Analysis

Distribution (Suite)	**Event**	**#Unique Packages**	**Filter 1**	**Filter 2**	**#Censored Packages**
Arch Linux (Community suite)	E1	8,529	8,492	3,238	514
	E2	3,238	N/A	2,724	131
Arch Linux (Extra suite)	E1	2,510	2,502	1,282	77
	E2	1,282	N/A	1,205	1
Arch Linux (Miltilib suite)	E1	280	280	171	9
	E2	171	N/A	162	35
Arch Linux (Core suite)	E1	209	208	104	3
	E2	104	N/A	101	1
Debian (Unstable suite)	E1	37,858	37,515	9,856	2,311
	E2	9,856	N/A	7,545	953
Debian (Bullseye suite)	E1	33,172	33,149	2,095	935
	E2	2,095	N/A	7,193	736
Debian (Buster suite)	E1	30,547	30,436	2,227	1,169
	E2	2,227	N/A	1,058	483
Debian (Stretch suite)	E1	27,003	26,931	5,373	1,288
	E2	5,373	N/A	4,085	1,703
Debian (Bookworm suite)	E1	16,036	16036	1,077	939
	E2	1,077	N/A	138	134
Debian (Experimental suite)	E1	8,118	8,073	2,432	1265
	E2	2,432	N/A	1,167	525

To understand the filtering of packages, let us consider the community suite of Arch Linux from Table 4. The same filtering applies to all other suites from both distributions. To study the event **E1**, let us consider 8,529 unique Arch Linux packages from Table 4. Then, we use Filter 1 to discard packages with multiple builds performed on the same day due to configuration errors, network latency, or incorrectly set environment variables in the build system. After applying Filter 1, we found 8,492 Arch Linux packages with a unique build status on the day build is performed. Next, Filter 2 discards all packages that initially were reproducible. After applying Filter 2, we found that 3,238 Arch Linux packages were initially unreproducible, and were used to perform survival analysis of event **E1**.

To study the event **E2**, we want to consider only those packages for which event **E1** has occurred (third column). This is because we want to understand how much time packages would remain reproducible before becoming unreproducible again. Considering that Filter 1, already filtered out packages for event **E1** that had multiple builds on the same day, there is no need for further filtration of such packages in the context of **E2**, therefore we mark Filter 1 for event **E2** as not applicable. Next, Filter 2 discards those packages for which **E1** does not occur. For instance, in the case of Arch Linux, we consider the initially unreproducible 3,238 packages and subtract the 514 censored packages, i.e., packages that do not perform the event in the given time frame. After applying Filter 2, we obtain 2,724 Arch Linux packages for which event **E1** has taken place.

Analysis 2: We apply the same Filter 1 and 2 as described for Analysis 1. Once filters are applied, we make use of UDD to extract the package domain information and associate each unique package with its domain information. The domain information is used to categorize the packages in either system-level or application-level domains based on their functionality (see Section 2.4). Considering only the Debian Unstable suite from Table 4, for event **E1**, after applying Filer 2 out of 9,856 packages we found 7,549 (3,017 system-level; 4,532 application-level) packages with domain information. On the other hand, for event **E2**, after applying Filter 2, out of 7,545 packages we found 6,231 (2,595 system-level; 3636 application-level) packages with domain information from the UDD database.

3.2.4 Calculate time until event

For the survival analysis, besides defining the events of interest, we need to compute the so-called *time until event.* This is the time (in #days) required for an event **E1** and **E2** to occur for a given data point. Let us use the package `dracut`, which is part of both Arch Linux and Debian distribution, to explain how we compute *time until event.* Figure 1 depicts the timeline of builds performed on `dracut`, where each node represents one single build. Within each node, we provide the reproducibility status computed for a build: r for reproducible and u for unreproducible. Below the nodes, we depict the build times (t_i, t_j, and t_k) sorted in ascending order. We compute **E1** as follows: $E1 = t_j - t_i$, where t_i indicates the time at which a package performs its first build (i.e., when reproducibility status $= u$) from the start of the recorded build data. t_j is the time of the first build for which reproducibility status $= r$ (i.e., when developers fixed the package). We compute **E2** as follows: $E2 = t_k - t_j$, where t_j is the time of the first build for which reproducibility status $= r$ and t_k is the time when the package becomes unreproducible again.

3.2.5 Perform Survival Analysis

For each analysis, we estimate the survival probability using the non-parametric Kaplan-Meier estimator [15]. The Kaplan-Meier estimation is computed following Equation 1, where t_i is the time duration up to event-occurrence point i, d_i is the number of event occurrences up to t_i, and n_i is the number of subjects that survive just before t_i. Values of n_i and d_i are obtained from the aforementioned ordered data obtained from Section 3.2.4. We use the python package *Lifelines*[23] for estimating the Kaplan-Meier Survival probabilities.

[23] `https://lifelines.readthedocs.io/`

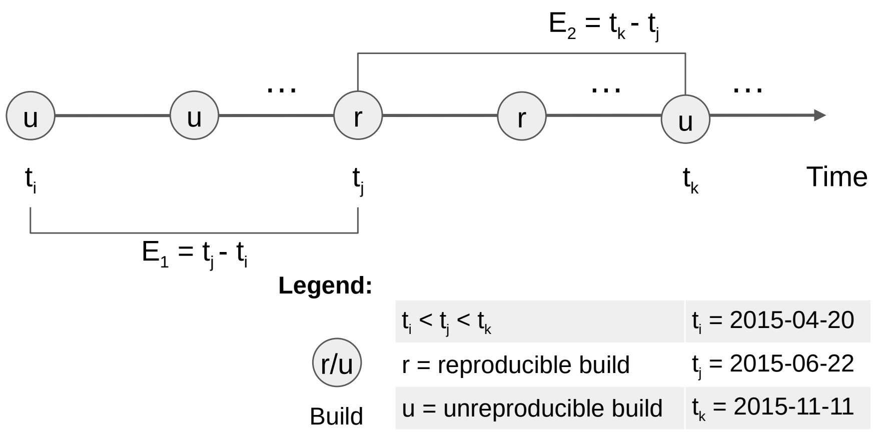

Fig. 1: Process to Calculate Time Until Event

$$S(t) = \prod_{i:t_i \leq t} [1 - \frac{d_i}{n_i}] \tag{1}$$

3.3 Manual Labeling Steps (RQ2)

As discussed in the introduction, the reproducible build process includes testing each package to verify its binaries against the source code. When Debian developers perform package builds to determine whether a particular package is unreproducible, they create a bug report on the unreproducible package by using Debian's reportbug[24] program. Developers then assess the bug report to identify the root cause behind the package's unreproducibility and figure out a fix.

To better understand the root causes of unreproducible packages, we considered performing a card sort on the 397 issues available on the Reproducible Builds website[25] at the time of collecting the data (November 11, 2021). The scraped data consists of issue names and reports the number of affected packages, affected package names, and the popcon score,[26] which is a popularity contest that enables identification of the issues that are relatively more popular amongst the Debian community. Notably, each issue can be linked to multiple affected packages. For example, the issue titled *gcc captures build path*[27] was associated with 1,832 affected packages at the time of our data collection.

[24] https://packages.debian.org/stable/utils/reportbug
[25] https://tests.reproducible-builds.org/debian/index_issues.html
[26] https://popcon.debian.org/
[27] https://tests.reproducible-builds.org/debian/issues/unstable/gcc_captures_build_path_issue.html

3.3.1 Data Collection

In order to support our study, we relied on the entire data set of 397 issues, with a total of 8,179 affected packages posted by developers on the Reproducible Builds website at the time of our analysis.

3.3.2 Identifying initial catalog of root causes

To prime our card sort analysis, we searched for references in the literature providing initial catalogs of root causes for build reproducibility issues. We found one recent experience report [17] from the industry that reported six possible root causes of unreproducible builds: Archive Metadata, Build Timestamp, Build Path, Filesystem Ordering, Randomness, and Uninitialized Memory. However, no systematic empirical evidence of these root causes was provided. We used these six categories as a foundation for our categorization.

3.3.3 Independent labeling

We started the manual categorization of the 397 issues. Initially, using the negotiated agreement technique [28], we categorized the first 10% of the issues together while the rest were categorized independently. The first two paper authors were assigned to engage in the manual card sorting of issues. Each of the two paper authors read the content of the issues and assigned a category without any external influences from the other author. To analyze each issue at a granular level, first, we examined the title and description of each issue. Furthermore, we inspect any external links, such as discussion forums and emails, attached to the issue by developers. We assigned a question mark ("?") when we could not assign any of the six existing categories.

3.3.4 Compute agreement & solve conflicts

Once we concluded the independent classifications, we computed the Cohen's Kappa agreement coefficient [24] for the two rater's classifications (Section 3.3.3). We obtained 68% of agreement, i.e., moderate agreement according to the common interpretation guidelines [24]. This agreement seems reasonable, considering that our study is the first to provide empirical evidence for the root causes of unreproducible builds.

We then held a meeting to discuss the disagreement instances one by one, so we could reach a consensus on the final labels for each issue. For each bug report where both authors agreed on uncertainty ("?"), we classified such issues into a new category. During this meeting, we also figured out that six issues involved more than just one root cause, so we decided to assign multiple labels to them. If we could not figure out the root cause associated with an issue, we assigned the Unknown category. Moreover, if a consensus still needed to be reached, we contacted experts via the Reproducible Builds mailing list.

This was the case for one issue affecting the Tomcat package.[28] The first two authors did not reach agreement on the classification of this issue, tentatively

[28] `https://tests.reproducible-builds.org/debian/issues/unstable/bundle_name_in_java_manifest_mf_issue.html`

creating a new category "Package Name" for this issue. The experts replied to us with a detailed explanation indicating that the unreproducible build results were due to the unreproducible outcome of data stored in Java's HashMap data structure.[29] Based on this, we could correctly re-categorize the issue under the Randomness root cause. Additionally, in another instance, we classified the root cause Locale as a subset of the Encoding branch. However, an expert reviewing our taxonomy argued that Locale is not related to the Encoding branch and should be its own standalone root cause category.

3.3.5 Build taxonomy

After our final decision about the categories assigned to all 397 issues under analysis, we held two meetings to build the taxonomy of root causes of unreproducible builds. In the first meeting, the first two paper authors compare the individually created taxonomies and resolve conflicts to generate a single taxonomy. In the second meeting, we take inputs from the third author to finalize the taxonomy. During the meeting, we discussed the hierarchical relationships between categories to create the taxonomy with multiple abstraction levels, from major to minor categories.

3.3.6 Frequency Analysis of Root Causes

Since the previous literature is based on industrial experience (see Section 8), we computed the frequency of issues for each root cause in order to validate the claims made by previous literature [17,23]. Furthermore, the frequency determines which underlying root causes are more widespread in real-world situations, so developers can focus their efforts on these fundamental causes.

3.3.7 Impact Analysis of Root Causes

We computed the number of packages affected by each root cause. This determines the intensity of a particular root cause's impact on the packages of a distribution. Additionally, we identified 3,566 packages out of the 8,179 packages affected by at least one root cause for which a package domain is associated with them (Section 2.4). After that, we computed the number of root causes affecting the packages for each package domain so that we could select the top-five package domains affected by the most root causes. This aims to understand if system-level package domains are affected by more root causes when compared to application-level package domains.

3.4 Correlation Analysis Steps (RQ3)

To examine external factors that may be related to unreproducible packages, we perform Chi-Square (χ^2) tests for independence [33]. The χ^2 test for independence evaluates the correlation between categorical variables having two (in our case) or more values. For example, gender and education are categorical variables where

[29] `https://github.com/bndtools/bnd/issues/5183`

male and female and levels of education, such as undergraduate and graduate, are the possible values, respectively. For this research question, we examine the correlation between three external factors (one at a time) and the reproducibility status of packages. We describe below the steps necessary to perform the (χ^2) test.

3.4.1 Collect data

Different packages are built at different instants; for each package, the reproducibility status of the most recent build is considered for computing the correlation. We explain below the definition of the external factors we have considered. The following paragraphs explain how we collected the data necessary to investigate each external factor, also showing the total number of packages available for analysis in each case.

Table 5 lists, inspired by the work of Decan et al. [9] on the socio-technical dependencies between packages, the external factors taken into consideration for our study. Furthermore, the table describes the detailed steps (third column) performed to collect data related to each external factor.

Table 5: External Factors Under Analysis

External Factor	**Definition**	**Computation**
Distribution	Collection of inter-dependent software constituting an operating system.	For Arch Linux's Community suite and Debian's Unstable suite, identify 4,108 unique packages with the same name in both suites, then obtain the reproducibility status of the packages in both distributions for the latest build performed.
Build Dependencies	Dependencies such as compilers, interpreters, and libraries that are essential to build packages.	1) We utilize the Ultimate Debian Database (UDD) as discussed in Section 2.2 to retrieve 15,076 build dependencies. The `build_depends` column in the `all_sources` table of the UDD contains a list of packages that a particular source package depends on during a build process. 3) Not all build dependencies found (15,076) were built to check reproducibility and, hence, did not have a reproducibility status. Thus, we identify 14,638 build dependencies with reproducibility status.
Architecture	Hardware architecture on which package builds are performed.	For the Unstable suite, we identify 36,303 unique packages with the same name in both architectures (armhf and arm64), then obtain the reproducibility status of the packages for both architectures.

3.4.2 Define null hypotheses

The null hypothesis for the χ^2 test of independence is that the reproducibility status of packages is independent from a given external factor. We define three null hypotheses:

1. HA_0: There is no relation between reproducibility status of packages between distributions.
2. HB_0: Reproducibility status of a package does not depend on the reproducibility status of its build dependencies.
3. HC_0: Reproducibility status of packages does not depend on the architecture for which their build is performed.

3.4.3 Compute contingency tables

A contingency table is a cross tabulation of the frequency of unreproducible or reproducible builds across two categorical variables (e.g., Arch Linux and Debian). Table 6 illustrates a generic contingency table for our analysis. The values in the contingency table are referred to as the observed values $O_k = \{01, 02, 03, 04\}$.

Table 6: Cross Tabulation for Hypothesis

External Factor			
		Categorical Variable 1	
		Unreproducible	Reproducible
Categorical	Unreproducible	01	02
Variable 2	Reproducible	03	04

3.4.4 Perform χ^2 test for independence

For each external factor, we use its corresponding contingency table (from Section 3.4.3) as input to the function `chi2_contingency` from the SciPy library.[30] Equation 2 computes the χ^2 statistic value, from which we obtain the `p-value` (p). In the equation, $E_k = \{e_1, e_2, e_3, e_4\}$ represents the theoretical expected values (one for each observed value), $k = 4$ because there are four observed values total for each analysis, and $d = 1$ the degree of freedom d equals to the number of categorical variables minus one.

$$\chi^2 = \frac{1}{d} \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k} \tag{2}$$

Regarding the interpretation of results, $p < 0.01$ means that the null hypothesis is false and, therefore, there is a strong relationship between the two categorical variables. As an alternative hypothesis, i.e., when the null hypothesis can not be rejected, the categorical variables are independent of each other.

In case we reject the null hypothesis, we compute the Cramer's V [35] estimator to understand the strength of the statistical difference better. The Cramer's V [35] estimator is computed following equation 3, where χ^2 is the calculated χ^2 statistic

[30] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html`

value, n is the sample size, and M is the minimum number of rows or columns. We use SciPy's statistical functions[31] for computing the Cramer's V estimator:

$$V = \sqrt{\frac{\chi^2}{n(M-1)}} \tag{3}$$

As seen in Table 7, Cramer's V [35] has a range of values from 0 to 1, with 0 denoting no connection and 1 denoting perfect association.

Table 7: Interpretation of Cramer's V estimator

Strength of association	**Mimimum band**	**Maximum band**
Negligible	>= 0	<0.1
Weak	>=0.1	<0.2
Moderate	>=0.2	<0.4
Relatively strong	>=0.4	<0.6
Strong	>=0.6	<0.8
Very strong	>=0.8	<= 1

3.4.5 Perform domain-oriented analysis

While the analysis in Section 3.4.4 helps us to understand if changes in the reproducibility status of packages are likely to depend on external factors, e.g., distribution, the obtained statistical difference might be biased to specific package domains. It is vital to identify package domains with a stronger correlation since they create a bias towards the reproducibility status of packages with respect to the external factor into consideration.

To identify essential package domains for prioritization, we perform χ^2 tests for independence (Section 3.4.4) again on all packages belonging to the same package domain for each external factor. Then we look at all package domains with a $p < 0.01$, indicating that the null hypothesis is false and that the package domain and the external factor have a significant correlation. After that, we compute the Cramer's V estimator, now for each package domain, to identify package domains that have a stronger correlation when compared to other package domains.

4 Analysis of Reproducibility Status (RQ1)

Using the approach outlined in Section 3.2, we initially conduct survival analysis using **Analysis 1** concerning events E1 (becoming reproducible) and E2 (reverting to unreproducibility) at the distribution level. Subsequently, we carry out a similar analysis at the domain level with **Analysis 2**, addressing both system-level and application-level packages.

[31] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.contingency.association.html`

4.1 Survival Analysis Across Both Distributions

The initially unreproducible Arch Linux packages become reproducible a median of 30 days earlier than Debian packages.

Figure 2 presents a scatter plot for the survival probability of unreproducible packages for all suites across distributions. The x-axis and y-axis represent the survival probability at 30 and 360 days, respectively. Each dot represents a suite belonging to a specific distribution – orange for Arch Linux and blue for Debian (Section 2.1).

Considering the x-axis, all probabilities at the 30 days mark are consistently lower for Arch Linux than those for Debian. Similarly, the y-axis indicates that almost all probabilities at 360 days are lower in Arch Linux – except for the Unstable suite (Debian; 18.3%), for which we have a tie with the Community suite (Arch Linux; 18.78%).

We particularly observe that the packages within official Debian releases, including Bookworm, Bullseye, Stretch, and Buster, strictly adhere to fixed release cycles with scheduled release dates and planned updates (see Section 2.1), after which no more (reproducibility) changes are performed. This observation explains the relatively high survival probabilities of unreproducible packages for these mentioned suites. In contrast, the reproducibility of packages in Debian's Unstable suite tends to evolve more dynamically, given its rolling release approach, and is, therefore, more similar to the Arch Full suite's reproducibility behavior.

Studying specific suites in more detail gives a clearer understanding. We found that half of all packages in Arch Linux become reproducible at least 92 days earlier than their Debian counterparts (in a period up to 720 days). This trend continues when comparing the Arch Full suite of Arch Linux to the Unstable suite of Debian, where Arch Full packages become reproducible at least 30 days earlier (again, observed over a 720-day period). These results strongly suggest that, on an overall basis, the time it takes for reproducible fixes to become available to users is more swift in Arch Linux when compared to Debian.

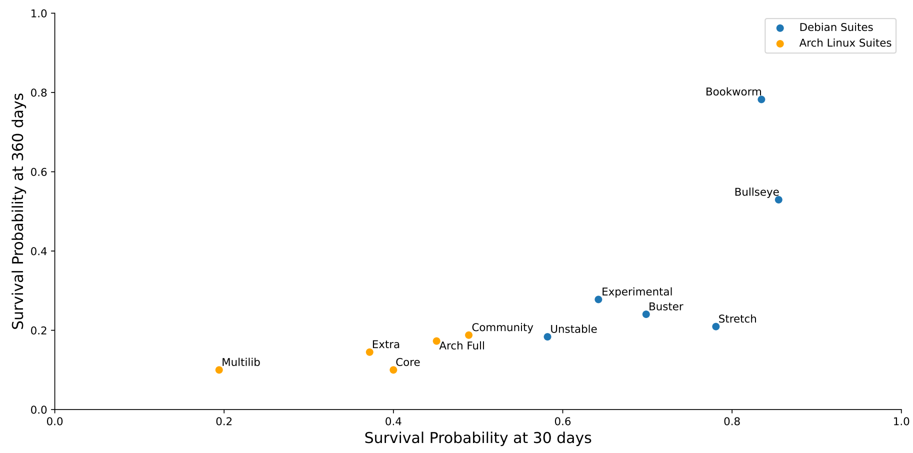

Fig. 2: Survival of Unreproducible Packages across Suites

We hypothesize three reasons for this result: (1) the type of release cycle followed in Arch Full, (2) the number of packages that comprise the distribution, and (3) the lower number of maintainers when compared to Debian. Regarding (1), as mentioned in Section 2.1, Arch Linux does not have a fixed release cycle; instead, package updates are made available to the distribution quickly after upstream package releases have been made. In particular, the time taken for an Arch Linux package to be released is between a few hours to a few weeks, which is less than Debian.[32] Debian's Unstable suite, while also following a rolling-release model, serves as a staging area for packages before they transition to the Testing suite. In Debian's Unstable suite, packages are subjected to rigorous checks, ensuring they remain free of release-critical issues for a specified duration before advancing to the Testing suite. In contrast, Arch Linux, while conducting tests on its packages, releases them at a notably accelerated pace, which might result in unforeseen (reproducibility) issues. However, this hypothesis requires further investigation.

Regarding (2), Arch Linux is a light-weight open-source operating system distribution, which means that it has fewer pre-compiled packages (12,984).[33] compared to Debian (148,000).[34] Having more binary packages in Debian indicates a need for more thorough testing of packages that interact with other packages, thereby leading to feature interactions. As a result, Debian packages may be prone to more test failures, mainly when one feature relies on another to function effectively. Consequently, we suspect that more dependence on such packages might lead to a chain of unreproducible packages. This might increase the time required by the developers to figure out the root causes and propose fixes.

Regarding (3), it is noteworthy to mention that Arch Linux presently relies on 63 maintainers[35] who undertake the crucial responsibilities of package release and maintenance, while Debian, in contrast, boasts a considerably larger team comprising 263 maintainers[36] to carry out similar tasks. Empirical investigations have consistently revealed that software development teams with a greater number of members tend to encounter extended development cycles, knowledge transfer intricacies, and overall project delays [12, 21]. One reason for these challenges might be the exponentially increasing communication channels when the number of developers increases [4]. Given the established fixed-release cycles of Debian, it is possible that these potential delays contribute to longer periods during which Debian packages continue to be unavailable in a reproducible manner to end users.

On the other hand, it is important to acknowledge that empirical studies [7] also indicate that larger development teams have the capacity to deliver enhanced software quality, cf. Linus's Law [34]: "given enough eyeballs, all bugs are shallow", a notion that aligns with the following findings.

Once developers have fixed an unreproducible package, Debian packages remain reproducible for a median of 68 days longer than Arch Linux packages.

Figure 3 presents a scatter plot for the survival probability of reproducible packages for all suites across distributions. The x-axis represents the survival prob-

[32] https://wiki.archlinux.org/title/arch_compared_to_other_distributions
[33] https://archlinux.org/packages/
[34] https://wiki.archlinux.org/title/arch_compared_to_other_distributions#Debian
[35] https://archlinux.org/people/trusted-users/
[36] https://nm.debian.org/public/people/dm_all/

ability at 30 days, while the y-axis corresponds to the probability at 360 days. Each dot represents a suite belonging to a specific distribution (orange for Arch Linux and blue for Debian). The dots are labeled according to the corresponding suites, as previously discussed in Section 2.1.

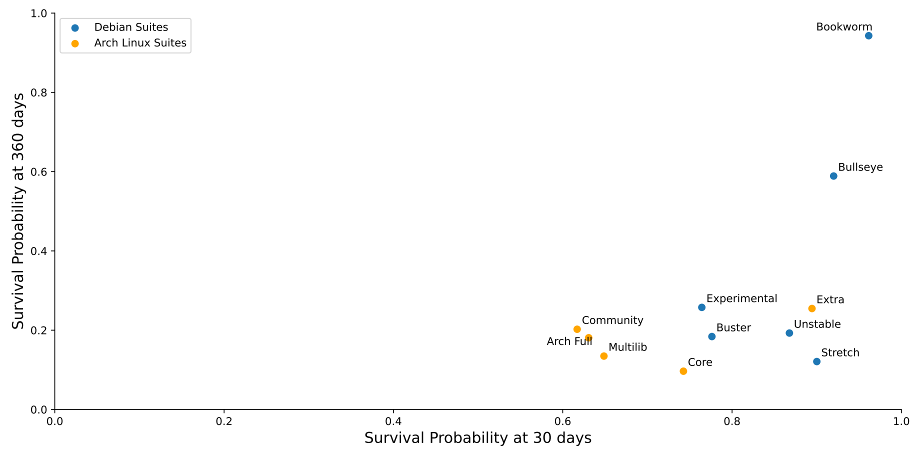

Fig. 3: Survival of Reproducible Packages across Suites

The survival probabilities for 30 days are higher than 61.7% for all suites regardless of the distribution. On the other hand, for 360 days, the survival probability drops drastically (more than 25.75%) for almost all suites in both distributions, except for the Bookworm and Bullseye suites for Debian – they present probabilities equal to 94.29% and 58.89%, respectively. This result is reasonable because these suites belong to the stable suites (Table 1), which follow a fixed-release model. As such, they undergo rigorous testing and are accessible freely for customer use. Surprisingly, the Extra suite from Arch Linux has an 89.44% probability of maintaining the reproducible state of its packages at the 30 days mark. In terms of duration, in-depth analysis of the suites showed that half of the packages from Debian's Unstable suite survive at least 68 days longer than those from the Arch Full suite in Arch Linux (up to 720 days).

One of the reasons why Arch Linux packages tend to become unreproducible more quickly once they are reproducible is because of the design decisions taken by the stakeholders of this distribution. For instance, the Arch Linux team opted to support only the latest version of packages as an attempt to reduce the maintenance effort associated with having multiple packages to deal with in the future.[37] However, to support the latest version of packages, their build dependencies must also be updated to support smooth build execution. Packages of interest might yield unreproducible builds if their build dependencies are unreproducible. The relation between factors like design decisions taken by the stakeholders of the dis-

[37] `https://wiki.archlinux.org/title/Frequently_asked_questions#Why_is_there_only_a_single_version_of_each_shared_library_in_the_official_repositories?`

tributions or reproducibility of build dependencies build reproducibility are further explored by RQ3 in Section 6.2 and Section 6.3.

We suspect that the maturity of a distribution and its packages might impact the packages to remain reproducible. More traditional distributions, i.e., those maintained for longer, are less likely to have unreproducible packages because more testing iterations were performed on these packages. We further discuss this assumption in Section 6.2.

4.2 Survival Analysis on Domain Clusters

Within a year, the survival probabilities in initially unreproducible application-level package domains are 12.9% higher than for packages in system-level package domains. The findings of **Analysis 2** of Section 3.2 suggest that the packages from the system-level package domains are the ones that stay unreproducible for the shortest time, which may be reasonable considering that packages that belong to the system-level package domain are more essential to the system's functioning and hence might be fixed more quickly. Furthermore, we found that initially unreproducible system-level packages become reproducible a median of 37 days earlier than packages that belong to the application-level domain.

Contradictory to the results of Figure 4, the survival probability of initially unreproducible packages in some of the most system-level package domains such as *interpreters* (28.57%), *net* (33.33%), and *otherosfs* (46.6%) is 2 to 30% greater than probabilities for application-level package domains such as *games* (26.01%), *mail* (31.8%) and *video* (17.39%). This is because the results of Figure 4 are skewed by extremely low survival probabilities of a few system-level domains (at the one-year mark) such as the *perl* (0.67%), *ruby* (0.0%) and *javascript* (2.3%) domains. Additionally, another reason for skewness could be extremely high survival probabilities of a few application-level packages domains (at the one-year mark) such as *education* (50.0%), *electronics* (53.33%), and *sound* (44.04%). More work is needed to analyze the reason for these outliers.

Once fixed, reproducible packages from the application-level package domains have a 5.67% greater survival probability of remaining reproducible when compared to system-level package domains. System-level packages tend to become unreproducible again more quickly, implying that these packages require more careful attention since they are more likely to be affected by vulnerabilities for a prolonged (and undesired) time.

Data of Figure 5 suggests that, within a year, the packages in application-level package domains like *fonts* (56.41%), *games* (25.0%), and *maths* (35.0%) have a 2 to 50% greater probability to remain reproducible when compared with the packages in the system-level package domains like *perl* (6.58%), *ruby* (4.0%), *java* (12.54%), *javascript* (15.87%), *admin* (23.73%) and *utils* (15.78%). One would expect the reproducibility status of system-level packages to be more stable over time in comparison with the status of application-level packages. This is because packages from the system-level package domain are more likely to be used by different applications, e.g., by serving as dependencies for other packages that also constitute the same distribution.

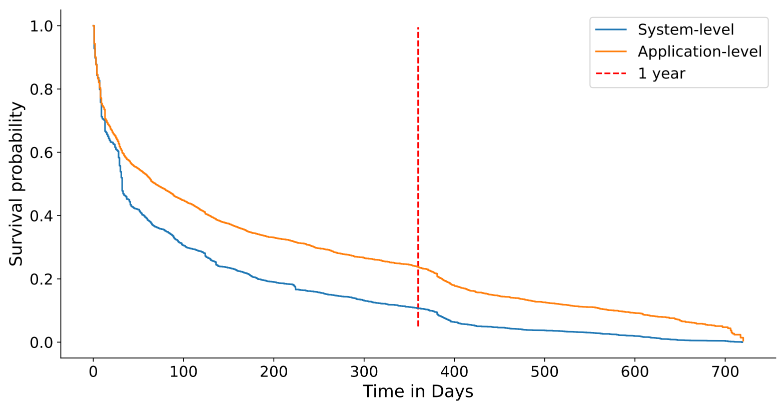

Fig. 4: Survival analysis for Unreproducible Packages of Domain Clusters

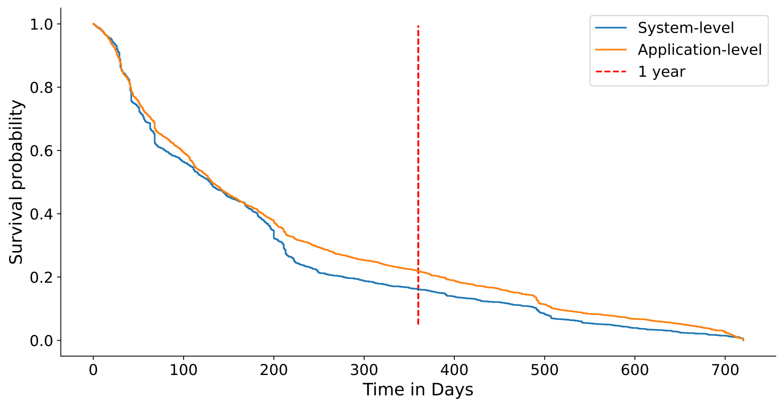

Fig. 5: Survival analysis for Reproducible Packages of Domain Clusters

Contradicting this expectation, we found that, within one year, packages belonging to the system-level domains are more likely to become unreproducible compared to the application-level package domains. We suspect that this could be because of the heavy changes that packages in the system-level package domains undergo, which might re-introduce reproducibility issues. For instance, the `perl`[38] package from the *perl* domain that encapsulates many `perl` libraries performs a new minor release every month. The `perl` package currently has 2.1k open issues and more than 500 core contributors. With the help of contributors, `perl` releases minor releases every month with more than 50 commits. Since code churn correlates with higher risks of introducing bugs [29], these frequent changes, which may

[38] `https://github.com/Perl/perl5/tree/v5.35.8`

range from minor code refactoring to fixing potential security issues, could lead to increased reproducibility concerns. Another example would be the `ruby`[39] package, which currently has 451 core contributors and 339 open issues. A minor release for the `ruby` package is published every four months with more than 100 commits per minor release. This might again correlate with a higher number of bugs due to frequent updates by many contributors, making the package less consistent in its reproducibility.

Summary of RQ1: Arch Linux packages become reproducible a median of 30 days quicker than Debian packages. However, after becoming reproducible, Debian packages remain reproducible for a median of 68 days longer compared to Arch Linux packages. Over the course of a year, initially unreproducible packages in application-level domains demonstrate a 12.9% increased likelihood of remaining unreproducible compared to those in system-level domains. After becoming reproducible, packages from application-level domains exhibit a 5.67% higher likelihood of maintaining their reproducibility in contrast to those from system-level domains.

5 Root Causes of Unreproducible Packages (RQ2)

Using card sorting of 397 build reproducibility issues (see Section 3.3), this RQ identifies a taxonomy of common root causes of build unreproducibility, then documents and discusses each cause, including its prevalence.

5.1 Taxonomy

Based on the manual labeling of 397 issues[40] related to Reproducible Builds, we derive a taxonomy of 16 root causes of unreproducible packages, grouped in 5 major groups, Figure 6 illustrates the derived taxonomy in which each box represents a different root cause. Each box is annotated with the number of categorized issues that belong to the root cause (top-right) and the number of packages that are affected by the root cause (bottom-right).

There are five major groups, each representing a different root cause of packages becoming unreproducible: **Build**, **Filesystem**, **Locale**, **Memory**, and **System**. The taxonomy has 16 leaves, each representing a specific root cause for the unreproducibility of a package. In Figure 6 and the definitions described below, we underline the six leading root causes identified by previous work [17]. We define and exemplify below each major group.

- **Build**
  1. **Build ID**
     Definition: Unique identification hash code generated during the build that is computed using parts of the software binary content. The purpose of

[39] `https://github.com/ruby/ruby`
[40] `https://tests.reproducible-builds.org/debian/index_issues.html`

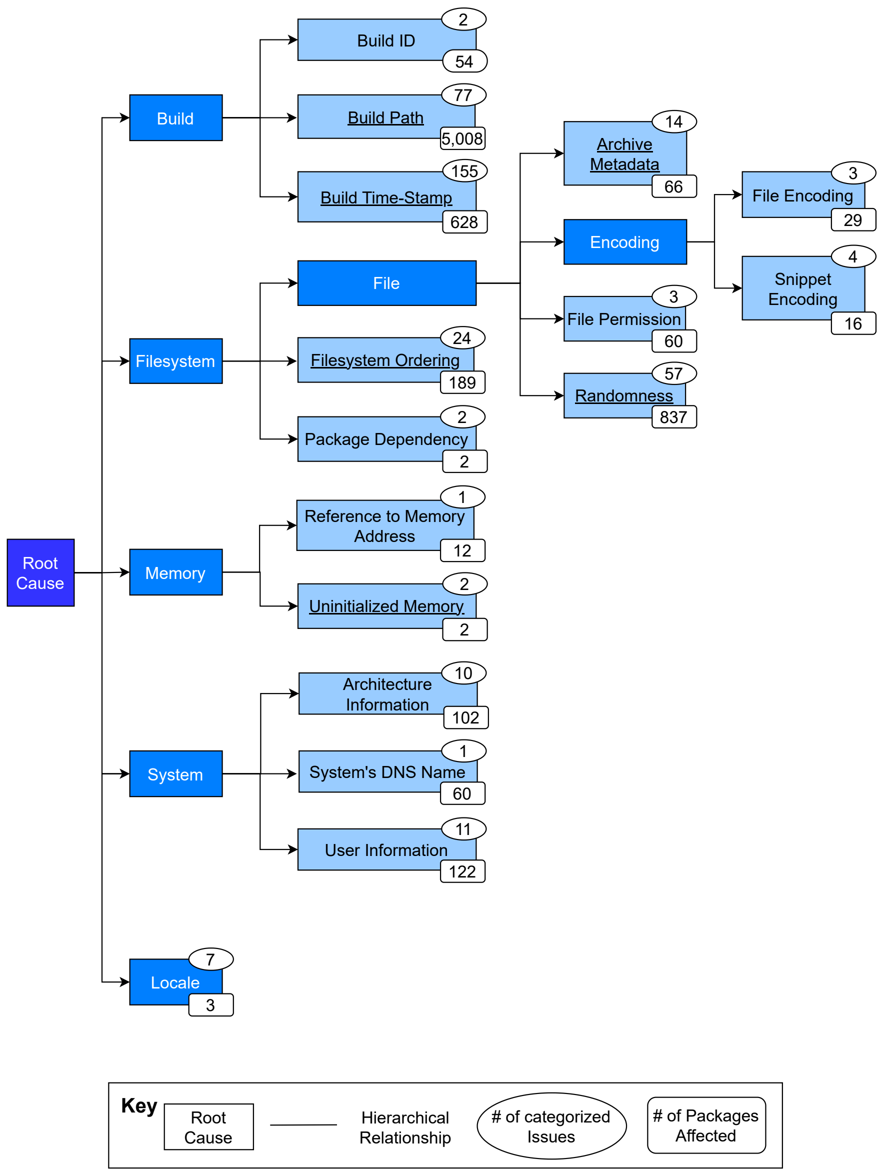

Fig. 6: Taxonomy of Root Causes of Unreproducible Packages

Build ID is to provide identical hash codes for identical binaries, allowing one to identify the latter uniquely based on their identity, not their

contents. When distinct builds on the same code artifacts output distinct Build IDs in the generated binaries, this indicates an unreproducible build process. Typically, how the Build Id is generated might be impacted by noise, preventing reproducible Build IDs.
Notable Instances: In one bug report,[41] a 128bit UUID comprising randomized bits including the time and host-based data, is used to generate the Build ID. Builds performed on different build systems would lead to different Build IDs due to inconsistent UUIDs, thereby causing unreproducible builds.[42]

2. **Build Path**
Definition: To perform a build, the compiler requires a build path to build configurations such as required build dependencies. Different values of the build path in distinct builds can lead to different dependencies being used during compilation or to different path names recorded in binaries and hence to differences in the resulting binaries, i.e., unreproducible builds.
Notable Instances: During the execution of builds for the `autoconf` package, one build was performed using a relative build path while the other used an absolute path.[43] Differences between the build paths recorded in the resulting binaries would cause unreproducible builds. To overcome such issues, the developers associated with the unreproducible build and the GCC community collaborated to introduce flags [17] that support the use of relative paths such that distinct builds are reproducible.

3. **Build Timestamp**
Definition: Information that corresponds to the date and time at which the build was performed. During the execution of a build, any newly generated, modified, or accessed files embed compile-time timestamps in the form of logs into the binaries [16]. Due to changes in build time, both builds produce binaries with different content when distinct builds are run. In principle, timestamps tell us very little about the software build, as builds can be performed on an older version of the software with a new timestamp. Instead, the reproducible builds developers introduced the environment variable SOURCE_DATE_EPOCH,[44] with the goal of replacing the current timestamp. This variable contains the timestamp of the last modification to the source code for that release, which is obtained from the source changelog file [17].
Notable Instances: C pre-defined macros such as `_Date_` and `_Time_` output the current time.[45] When these macros are invoked by distinct build systems, different timestamps become a part of the compiled code, which leads to different binaries.

– **File System**

[41] `https://tests.reproducible-builds.org/debian/index_issues.html`
[42] `https://fedoraproject.org/wiki/Releases/FeatureBuildId#Unique_build_ID`
[43] `https://tests.reproducible-builds.org/debian/issues/unstable/captures_build_path_issue.html`
[44] `https://reproducible-builds.org/specs/source-date-epoch/`
[45] `https://tests.reproducible-builds.org/debian/issues/unstable/timestamps_from_cpp_macros_issue.html`

1. **File**
   (a) **Archive Metadata**
   Definition: Compressed file formats such as zip and tar store file metadata such as file owners, permissions, and timestamps. This metadata is called Archive Metadata. Extracting these compressed files during a build can cause the unreproducibility of packages because files can be extracted at a location with a different file owner and permissions than the original source. This leads to assigning different file owners and permissions to the extracted files. Furthermore, when uncompressed, these files generate a timestamp that is inconsistent with the originally generated file timestamp.
   Notable Instances: In case of `libguestfs`,[46] the compressed files store a modified timestamp (mtimes). This mtimes gets modified during the execution of the build. When two distinct builds are run, the inconsistent timestamps from the compressed files are written in the resulting executables causing unreproducible builds. Additionally, using the `cmake` command with the *tar* option inherits the User ID (UID) and Group ID (GID) on the build system during the execution of the build.[47] Inconsistent UID and GID in resulting binaries lead to unreproducible builds.
   (b) **Encoding**
      i. **File Encoding**
      Definition: File Encoding refers to the encoding of files. When builds are performed on distinct build systems, using different encoding schemes (e.g., UTF-8 vs. non-UTF) can lead to different binary patterns. This can cause packages to become unreproducible.
      Notable Instances: In the case of the `r-base` package, files on distinct machines were built using a different encoding, i.e., one of the builds was using a non-UTF encoder while the other was using a UTF-8 encoder. Since both represent different encoding strategies, the resulting binaries containing the files have different content leading to unreproducible builds.[48]
      ii. **Snippet Encoding**
      Definition: Encoding strings or parts of a file with random numbers is called Snippet Encoding. These random numbers serve as security keys that encode data to prevent unauthorized use. During the execution of a build, these randomized digits are included in the resulting binary. Resulting binaries with different content are produced as a result of distinct build systems producing different randomized digits.
      Notable Instances: The `libtext-markdown-perl` package makes use of the `srandom`[49] utility, which is used to provide a seed value to the randomization function. During the execution of distinct

[46] https://tests.reproducible-builds.org/debian/issues/unstable/varying_mtimes_in_data_tar_gz_or_control_tar_gz_issue.html
[47] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=973845
[48] https://tests.reproducible-builds.org/debian/issues/unstable/bundle_name_in_java_manifest_mf_issue.html
[49] https://linux.die.net/man/3/srandom

builds, the seed value to the randomization function was stored in the resulting binary, but was different for distinct builds, therefore build becomes unreproducible.[50]

(c) **File Permission**
Definition: Enables specific users and groups to either read, write or execute a file. During a build, new files created may inherit predefined file permissions from the containing folder. These permissions can differ for distinct build systems. Such inconsistent file permissions become a part of the compiled binaries during the build process, causing unreproducible builds.
Notable Instances: Packages such as `pcp` and `live-manual` have been found to have files with different permissions across the distinct builds, which have led to builds becoming unreproducible.[51]

(d) **Randomness**
Definition: Randomness leads to unpredictable outcomes for data stored in data structures or tasks performed in parallel. While performing builds on distinct build systems, the order in which parallel jobs execute might not be the same. The resulting logs of such parallel build execution could then be recorded in the resulting binaries causing the builds to be unreproducible.
Notable Instances: Python 2.7 data structures like tuples and dictionaries render unordered output. For instance, the Python package `python-ply` introduces a variable to store and retrieve data in the tuple format. When distinct builds are performed, the values are to be output into the binary; these outcomes are unordered. This randomness, when recorded in the build logs, causes unreproducibility. To overcome such issues, the developers need to sort data while retrieving them from the data structures.[52]

2. **Filesystem Ordering** *Definition:* Order in which files are created and displayed. Different file orders could result in a different ordering of segments inside generated binaries, causing unreproducible builds.
Notable Instances: `ruby2.3` contains *mkmf.rb*, which is used to auto-generate the Makefiles for multiple ruby applications. These generated Makefiles do not sort the list of object files.[53] Build logs of such unordered compilation when recorded in the resulting binaries of distinct builds cause unreproducibility.
3. **Package Dependency**
Definition: Package dependencies of a package P are essential software packages that are necessary for P to function effectively. They help eliminate code duplication within P's source package, and typically are developed by experts. However, when packages fail to specify the exact versions of their dependencies, this can result in a range of problems related to build

[50] `https://tests.reproducible-builds.org/debian/issues/unstable/markdown_random_email_address_html_entities_issue.html`
[51] `https://tests.reproducible-builds.org/debian/issues/unstable/different_due_to_umask_issue.html`
[52] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=890620`
[53] `https://tests.reproducible-builds.org/debian/issues/unstable/ruby_mkmf_makefile_unsorted_objects_issue.html`

dependencies. These issues include missing dependencies, conflicting dependencies, and the utilization of incompatible or outdated dependencies. Consequently, these challenges become the root cause of unreproducible builds. Moreover, even the behavior or execution of a build dependency can introduce inconsistencies in the build process.
Notable Instances: In the case of "ftbfs_due_to_virtual_dependencies",[54] certain packages experience build failures because they are unable to find or fulfill the virtual dependencies they require. Virtual dependencies are abstract dependencies that do not point to a particular package, but rather describe a feature that can be provided by several packages. This failure may stem from improperly defined virtual dependencies or a lack of available packages in the build environment that provide the necessary capabilities or features expected by these virtual dependencies.

– **Memory**
  1. **Reference to Memory Address**
     Definition: Memory addresses consist of digits that represent specific memory locations in build environments. Data structures of multiple programming languages, such as C and Python, can access these addresses. During the execution of distinct builds, having different memory addresses allocated for the same object causes dissimilar content stored in the resulting binaries. This causes the build to become unreproducible.
     Notable Instances: The `ldaptor` package makes use of a python module called `weakref`.[55] The `weakref` python modules uses the `_repr_`[56] function. This function outputs the memory address of the instance passed to it. Instances passed to this function yield distinct memory addresses that become a part of the compiled binaries, causing unreproducible builds.[57]
  2. **Uninitialized Memory**
     Definition: Uninitialized memory is the unused memory assigned to a resource such as a data structure or file system. For instance, data structures of multiple programming languages can be assigned a larger memory allocation than required. For optimization purposes, this memory is filled with randomized padding. During the execution of distinct builds, resources using the initialized memory are stored in files that could become linked into the resulting binary, causing the builds to become unreproducible.
     Notable Instances: During the execution of a build, the `ipadic` package generates a `.dat` file that contains uninitialized memory. To fill this uninitialized memory, randomized padding is done. When distinct build systems perform builds for the `ipadic` package, the generated `.dat` files in the resulting binaries with randomized padding cause the package to become unreproducible on Debian and openSUSE.[58]

– **System**

[54] `https://tests.reproducible-builds.org/debian/issues/unstable/ftbfs_uninvestigated_unsatisfiable_dependencies_issue.html`
[55] `https://docs.python.org/3/library/weakref.html`
[56] `https://docs.python.org/3/library/functions.html#repr`
[57] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=827416`
[58] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=881231`

1. **Architecture Information**
   Definition: This information reveals details related to architectural-level information like the Linux kernel version and hardware architecture name gathered from the `uname`[59] utility. Having performed builds on distinct machines, calls to the `uname` utility may output different hardware architecture and kernel versions that are compiled into different resulting binaries, causing unreproducible builds.
   Notable Instances: During the execution of a build, the `systemd`[60] package initiates a call to the `uname` utility (for debugging purposes), which outputs the build system's hardware architecture. The builds were run on distinct build systems, one using the `i686` architecture while the other was `x86_64`. When identified in the resulting binaries, the difference in architectural information causes unreproducible builds.
2. **System DNS Name**
   Definition: This name identifies a host computer in a specified network and is called the hostname of the system. Different build systems obtain distinct DNS names, similar to the Build Path root cause, which could cause unreproducibility for the resulting binaries.
   Notable Instances: Packages like `python-qtconsole` and `vlc` output unique hostnames of distinct build servers that end up in the resulting binaries. During the execution of builds on distinct build servers, the resulting binaries would not be bit-to-bit identical, causing unreproducible builds.
3. **User Information**
   Definition: Information that reveals a user's identity, such as username, and that could end up inside the build logs. When captured by the resulting binaries, divergent user information causes builds to become unreproducible.
   Notable Instances: During a build, the `gnustep-base` package executes the string *"generated by $USER"*, where the *$USER* represents the name of the system's user executing the build.[61] Since builds are performed on distinct build systems, the *$USER* outputs different usernames for each build system, causing unreproducible builds.

– **Locale**
  Definition: Locale provides users the ability to make use of user-specific language settings that are then translated to the corresponding binary code. Words in different locales correspond to different binary codes. Therefore, if there is a difference in the locale used between the two build systems while performing distinct builds, then the resulting binary will differ in content. This will cause the builds to become unreproducible.
  Notable Instances: In the case of the `python-babel` package, some of the default parameter values for functions were set to the user's locale rather than the build system's locale during the execution of a build. When different users performed builds on distinct machines, the difference in their locale led to the builds becoming unreproducible.[62]

[59] `https://man7.org/linux/man-pages/man1/uname.1.html`
[60] `https://tests.reproducible-builds.org/debian/issues/unstable/captures_build_arch_issue.html`
[61] `https://tests.reproducible-builds.org/debian/issues/unstable/user_in_documentation_generated_by_gsdoc_issue.html`
[62] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=795997`

Finally, we categorized 186 unreproducible packages (2.4%) as **Unknown**, which means that it was not possible to assign a known category to the issues reporting on the unreproducibility of these packages (Section 3.3). In certain instances, developers did engage in discussions regarding potential root causes, but provided only limited information, which hampers their definitive categorization. Similarly, there are cases where no explicit descriptions are available for the purpose of classifying the root cause. Since, at the time of analysis, these specific issues were yet to be thoroughly explored by Debian's reproducible build experts.

The `openssl` package, which provides secure communication over a computer network, is an example of the latter case. The developers argue[63] the following on the root causes potentially affecting the package, thereby reflecting their uncertainty on what should be done in order to fix a package.

Could be:
1) jobs that try to bind to localhost ports which are in use by something in the jenkins machines. [...]
2) the build tries to access a debian archive over http, which is something allowed but pbuilder doesn't support this yet [...]
3) something else, the machine is heavily loaded all the time...

5.2 Prevalence of taxonomy categories

Build Path (5,008 affected packages) has the most significant influence on unreproducible builds, with a five-fold increase in impact over the second most influential root cause, Randomness (837 affected packages). Table 8 lists the root causes (column 1) with corresponding frequency (column 2) found in the issues and the number of packages (column 3) that are affected by the root cause in consideration. We highlight the root causes from the previous literature [17] with bold font.

Based on data from Table 8, we see that four of the six categories introduced by the previous work are the most impactful in terms of how many unique packages they affect. Hence, our empirical study confirms the influence of those four categories proposed by Chris Lamb et al. [17]. In contrast, Lamb et al.'s other two root causes, i.e., Archive Metadata and Uninitialized Memory, are less common in the studied distributions. Indeed, the former regards creating and managing compressed files such as ZIP and RAR files, while the latter regards access to specific memory regions with uninitialized data.

During our analysis, it was observed that Build Timestamp has a significant impact on a relatively smaller number of packages (628 packages) when compared to Build Path and Randomness. This does not indicate that Build Timestamp is a less important root cause, but rather that the specification added to fix Timestamp-related issues, i.e., SOURCE_DATE_EPOCH, performed very well to resolve the issue. However, our findings have also revealed other valuable insights, highlighting the need for additional fixes to achieve the desired level of reproducibility in software projects, in contrast to existing literature [17, 23], which suggests that rectifying timestamps could potentially resolve the majority of unreproducibility

[63] `https://tests.reproducible-builds.org/debian/issues/unstable/ftbfs_in_jenkins_setup_issue.html`

Table 8: Frequency of Issues and Affected Packages

Root Cause	**Frequency of Issues**	**Affected Packages**
Build Path	77	5,008
Randomness	57	837
Build Timestamp	155	628
Filesystem Ordering	24	189
User Information	11	122
Architecture Information	10	102
Archive Metadata	14	66
File Permission	3	60
System's DNS Name	1	60
Build ID	2	54
File Encoding	3	29
Snippet Encoding	4	16
Reference to Memory Address	1	12
Locale	7	3
Uninitialized Memory	2	2
Package Dependency	2	2

cases. Nonetheless, our results indicate that additional essential fixes are required to accomplish the desired degree of reproducibility in software projects.

When considering frequency in terms of the number of affected issues, data of Table 8 shows that five out of the six root causes elicited by Lamb et al. again are amongst the most frequent ones, totaling 327 occurrences in the data set. This result is reasonably expected if we consider the nature of the previous work, which an expert proposed based on industry experience with reproducible builds. On the other hand, Uninitialized Memory rarely appeared in our manual analysis (two occurrences only), which may indicate that this root cause is specific to a particular domain. The remaining 12 root causes are less frequent in our data set, but they have their importance when it comes to unreproducible builds. These root causes are related to problems that may not be trivial to cope with in practice, such as Package Dependency and File Permission.

5.3 Package Domain-related Analysis

Because of the diverse nature of root causes impacting the packages, our findings show that unreproducible packages from system-level package domains are likely difficult to be fixed (made reproducible). Table 9 presents the number of packages of a domain impacted by each of the root causes. Out of a total of 59 package domains, we decided to consider only the top five domains whose packages are affected by most of the different root causes – namely, *devel*, *science*, *virtual*, *utils*, and *admin*. We split the rows into two groups: the first group (rows 3 to 8) corresponds to the root causes that appeared in the literature [17]; the second group (rows 9 to 17) corresponds to root causes that emerged from our manual labeling (see Section 5.1). For each group, we sorted the rows considering the second column (i.e., the *devel* package domain) in decreasing order because *devel* is affected by the majority of the root causes (13 in total).

We highlighted in bold font the root cause affecting the highest number of packages for a given domain and root cause. When considering frequency in terms

Table 9: Domain-wise packages affected by Root Causes

Root Cause	**Number of Packages Affected by Domain**				
	Devel	**Science**	**Virtual**	**Utils**	**Admin**
Build Path	**190**	**190**	**64**	**260**	**102**
Build Timestamp	58	45	15	27	10
Randomness	15	29	10	6	7
Archive Metadata	9	4	**64**	3	0
File System Ordering	5	8	**64**	9	8
Uninitialized Memory	0	0	1	0	0
Architecture Information	**11**	**12**	2	**9**	**3**
User Information	**11**	4	**5**	1	2
File Permission	**11**	2	2	2	0
Build ID	4	3	1	3	0
File Encoding	2	2	0	0	1
Encoding Information	1	1	0	0	**3**
System's DNS	1	2	1	1	2
Locale	1	2	0	0	0
Package Dependency	0	0	0	0	1

of the number of affected packages across the domains, data of Table 9 shows that the *virtual* package domain has the least packages affected by Build Path. In contrast, it has the most packages affected by Archive Metadata and File System Ordering. Furthermore, among the newly discovered root causes, Architecture Information affects the most packages across domains.

Except for *science*, all package domains are system-level package domains, which suggests that unreproducible packages from the system-level domains tend to suffer more from unreproducible build issues due to the varied nature of root causes affecting them. This confirms, to some extent, our past assumptions drawn while discussing RQ1 results. Those results suggested that packages belonging to the system-level domains remain unreproducible for the longest time and might be affected by vulnerabilities for a considerable time.

It is important to stress that each root cause is different: some are simple to fix, e.g., Randomness, while others are more complex and require more effort to be fixed, e.g., Package Dependency and Build ID. Thus, it is even more challenging for developers of the system-level domain packages first to identify the varied root cause, then provide an appropriate solution to fix the issue.

We noticed that a considerable number of packages (18.46%) are affected by multiple root causes, which makes it even harder to prevent them from becoming reproducible. Figure 7 demonstrates that while most (81.54%) of the packages are affected by one root cause, there are 18.46% that are affected by two or more root causes. For instance, packages such as *odc* are affected by nine root causes, while `php7.4`, `r-base`, and `gdb` are affected by eight different root causes. Fixing packages affected by different root causes can be tedious due to the varied nature of root causes.

Because it is unfeasible to present and discuss all of them in the paper, we cherry-picked five examples of packages to be discussed out of the three system-level package domains (*devel*, *admin*, and *utils*) listed in Table 9. For a detailed discussion, Table 10 presents all the root causes affecting the selected packages – namely, `gdb`, `autoconf`, `apt`, `pcp`, and `openssl`. The legend provides identifiers for

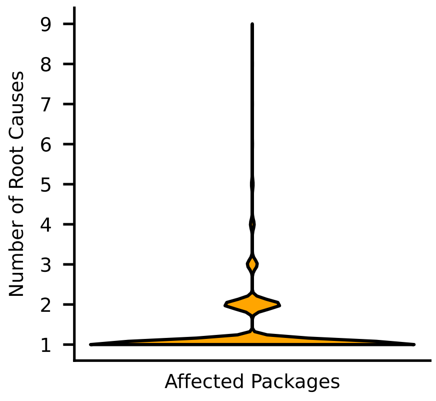

Fig. 7: Number of Root Causes Affecting Packages

the issues related to reproducible builds[64] from which we derived the root causes as discussed in Section 5.1.

Table 10: Packages Affected by Multiple Root Causes

Root cause	**Devel**		**Admin**	**Utils**	
	gdb	**autoconf**	**apt**	**pcp**	**openssl**
Architecture Information	11				
Archive Metadata				14	
Build ID		09			
Build Path	02	03	04, 07	02	01
Build Timestamp	11, 13	09		12	
File Permission	05			05	
Randomness		10	08		
User Information	11				
Unknown Issue					06

Legend:

01 - build path captured in assembly objects	08 - nondeterministic ordering in documentation generated by doxygen
02 - captures build path	09 - pdf created by ghostscript
03 - captures shell variable in autofoo script	10 - random id in pdf generated by dblatex
04 - cmake rpath contains build path	11 - test suite logs
05 - different due to umask	12 - timestamps in gzip headers
06 - ftbfs in jenkins setup	13 - timestamps in jar
07 - gcc captures build path	14 - users and groups in tarball

Proposing fixes to a package suffering from multiple reproducible build issues can be challenging yet essential. This was the case for the `gdb` package (from the *devel* package domain), which had problems associated with Architecture Information, Build Timestamp, and User Information (cf. Issue 11). It is worth mentioning that `gdb` is the official GNU project debugger and is an essential feature for the debugging of C and C++ programs. Furthermore, `gdb` is a dependency for crucial packages such as `python` and `rust`.[65]

[64] `https://tests.reproducible-builds.org/debian/index_issues.html`

[65] `https://archlinux.org/packages/extra/x86_64/gdb/`

Multiple issues of the exact root cause might be reported for a given package. This was the case for the `apt` package, which is reported to be affected by the root cause of Build Path in multiple issues (cf. Issues 04 and 07). We note that `apt` stands for Advanced Package Tool, and this is probably the most important package of Debian because it is responsible for managing the installation and upgrade process of packages for Debian. Furthermore, the fact that Build Path-related issues appear from issues related to `cmake` (Issue 04), which is a build-dependency of the `apt` package[66] and `gcc` (Issue 07) which is a runtime dependency of the `cmake` package, suggests that the unreproducibility is emerging from the build environment that is being used to build the `apt` package. In other words, external factors such as build dependencies and the runtime dependencies of their build dependencies may be affecting the reproducibility status of the package – which will be further investigated in RQ3 (see Section 6).

Summary of RQ2: Our qualitative analysis enabled us to considerably expand the current literature knowledge on the root causes of unreproducible packages. We derived a taxonomy consisting of 16 root causes grouped into five categories: **Build**, **Filesystem**, **Memory**, **Locale**, and **System**. While Build Timestamp is the most frequent root cause found in the issues analyzed, Build Path has the most significant impact on packages.

6 Correlation between External Factors and Reproducibility Status (RQ3)

This final RQ uses statistical χ^2 tests, as discussed in Section 3.4, to study the impact of three ecosystem factors, external to a given package on that package's own build reproducibility.

6.1 Overview of Correlation Results

As discussed in Section 3.1, RQ3 aims to understand software ecosystem factors outside the control of a given project that might correlate with its reproducibility status. For this, we determine the correlation between the external factors defined in Section 3.4 and the reproducibility status of packages across Arch Linux and Debian. Table 11 provides the cross-tabulation for the reproducibility status of packages in Arch Linux and Debian for each external factor. It also shows the Chi-Square test score (χ^2), which determines whether the impact of the external factor is statistically significant ($p - value < 0.01$) or not. Moreover, the table also provides insights into the effect size of the correlation by using Cramer's V estimator.

We also perform the analysis per domain for each factor, with results summarized in Table 12. We could find at least one domain having significant findings for each factor, except for runtime and build dependencies. In the latter case, no effect size could be calculated. We discuss the results of the domain-oriented analysis in detail for each external factor in the following sub-sections.

[66] `https://sources.debian.org/data/main/a/apt/2.5.4/debian/control`

Table 11: Cross-tabulation of reproducibility status of packages w.r.t their ecosystem

Distribution			
		Arch Linux Packages	
		Unreproducible	Reproducible
Debian Packages	Unreproducible	189	480
	Reproducible	688	2,751
$\chi^2 = 22.1875$; $p < 0.01$; $V = 0.0743$			
Build Dependencies			
		Build Dependencies for Debian Packages	
		Unreproducible	Reproducible
Debian Packages	Unreproducible	11,030	2,632
	Reproducible	15,929	8,206
$\chi^2 = 925.4351$; $p < 0.01$; $V = 0.1565$			
Architectures			
		Debian Packages Built on ARM64 Architecture	
		Unreproducible	Reproducible
Debian Packages	Unreproducible	6,013	822
	Reproducible	329	29,139
$\chi^2 = 29024.896$; $p < 0.01$; $V = 0.8942$			

Table 12: Cramer's V strength of statistically significant associations between Domains and External Factors.

Correlation Strength	**External Factors**		
	Distribution	**Build Dependencies**	**Architecture**
Negligible			
Weak		*admin, misc, web, java, python, science, utils.*	
Moderate	*devel, gnome.*	*graphics, devel, net, mail, interpreters, otherosfs, text.*	
Relatively Strong	*graphics*	*comm, fonts.*	
Strong		*lisp*	*java, gnustep, ruby, shells, metapackages.*
Very Strong			*All other domains except the ones mentioned in Strong correlation.*

6.2 Distribution

Test results: The difference between the reproducibility status of packages across the distributions these packages are part of is significant ($p - value < 0.01$), but with negligible effect size ($0.0 < V < 0.1$).

Discussion: Despite a negligible effect size, determining the reproducibility status of a package in a distribution does not guarantee similar reproducibility status for that package in other distributions. This result can be justified by different design decisions by stakeholders of various distributions. These design decisions might include factors like the supported architectures (Section 2.2), file systems, and release cycle management (Section 2.1). For instance, Arch Linux does not have a default file system defined, allowing users to choose the file system at their convenience, whereas Debian uses the ext4 file system by default. Using different file systems leads to different file permissions (Section 5.1) and sizes, which might lead to unreproducibility.

To further understand this finding, our domain-oriented analysis revealed that the *gnome* package domain and the *graphics* package domain are two of three package domains that exhibit a significant difference in the reproducible build ratio across distributions ($p < 0.01$). The effect size is moderate ($V = 0.3987$) for the former and relatively strong ($V = 0.5095$) for the latter. Both the *gnome* and *graphics* package domains are considered application-level package domains.

Packages within the *graphics* package domain are responsible for tasks like image manipulation, processing of image and video content, and rendering graphical content on various hardware to ensure optimal performance and compatibility. These packages frequently interact with binary data formats, such as compiled shaders, image files, or proprietary formats like the PSD file format for Photoshop documents and Autodesk's DWG format. Such formats might embed timestamps, version information, or metadata related to data manipulations. When builds are executed on different distributions, these factors can more easily influence the change in reproducibility status of the *graphics* package across those distributions.

Packages in the *gnome* package domain pertain to applications, libraries, and tools that support desktop environments on Linux-based systems. Desktop environments offer users a variety of desktop themes, the ability to use different locales and other graphical elements. Graphical elements, such as icons, might embed timestamps or vary based on the locale when compiled on different distributions. This can also influence the change in reproducibility status of the *gnome* packages across distributions.

Furthermore, the *devel* package domain demonstrates a significant difference in the reproducible build ratio across distributions ($p < 0.01$), with a moderate effect size ($V = 0.2626$). The *devel* package domain is regarded as a system-level package domain, encompassing build systems, compilers, and header files from these shared libraries. The paths to these header files might vary when compiled on different distributions, which could also contribute to the change in reproducibility status of the *devel* packages across distributions.

Implications: We saw in RQ1 that fixing unreproducible packages takes a long time for developers; therefore, this does not appear to be a simple task. On the other hand, the RQ3 results for distribution imply that a given package, especially in the *devel* domain, should be tested separately in terms of build reproducibility for all distributions since testing it for one distribution says little about what to expect in terms of reproducibility status in other distributions. Package builds, release cycles, file systems, and supported hardware architectures all differ across distributions, which makes the parameters in the build process of packages considerably different depending on the distribution. Considering the mentioned factors, we suspect that developers might discover totally different root causes to deal with while testing across distributions.

Considering the root causes elicited in RQ2 (Figure 6), we assume that specific root causes such as Architecture Information, Build Path, Filesystem Ordering, File Permissions, Locale, Reference to Memory Address, and Uninitialized Memory are distribution sensitive – i.e., each of these attributes has varied characteristics in different distributions. Since many packages are affected by either of the abovementioned root causes, it makes sense that the reproducibility status of a given package might be different between different distributions. However, we cannot provide evidence for such scenarios since build logs of packages are not maintained by the distributions; only the latest build results are stored on the aforementioned Reproducible Builds website. Having logs of previous builds performed can be a potential future work.

This also has another implication for distributions like Arch Linux, which do not categorize reported reproducibility bugs based on their root causes (as Debian is doing). Usually, when an issue is reported in the bug tracker for a distribution like Debian, members of the Reproducible Builds project examine it, categorize it according to pre-defined categorized issues,[67] and try to replicate the issue on their local instance. Subsequently, a patch is suggested by the members to fix the issue. Lastly, the patch is submitted to the upstream project. Although Arch Linux has a specific bug tracker[68] to cope with bugs related to reproducible builds, this bug tracker does not categorize the bug reports based on their root causes as done in Debian. Instead, Arch Linux members may have to check the Debian bug tracker to categorize the issues, helping them create patches for their distribution. However, due to the impact of distributions on the reproducibility of a given project, particularly for *devel* packages, causes of unreproducible builds might be highly different between distributions. Hence, we recommend that the Arch Linux community systematically categorizes the root causes independently.

6.3 Build Dependencies

Test results: The distribution of the reproducibility status of packages significantly differs based on the reproducibility status of the packages' build dependencies ($p - value < 0.01$), with weak effect size ($0.1 < V < 0.2$).

Discussion: The results indicate that even if a given project has a reproducible build, its own build dependency not being reproducible might render it unreproducible as well. In other words, this project's developers rely on the developers of the unreproducible build dependency to render their project build-reproducible.

Out of the top ten most influential build dependencies (i.e., packages that serve as build dependencies to most other packages), three packages belong to the *devel* package domain listed with their frequency of occurrences respectively – `debhelper` (15,887), `cmake` (2,965), and `cdbs` (2,595). Figure 8 shows the reproducibility status of different releases of the `debhelper` package, which has been the top most influential build dependency for the past seven years.

We observe that the majority of the builds in the past seven years have had a reproducible build. This result is reasonable since the `debhelper` package automates frequent tasks like assigning appropriate file permissions, installation, and compression of required files during the build process of Debian packages. Furthermore, tasks performed by this package are crucial since they directly relate to the root causes we found in RQ2, such as File Permission, Package Dependency, and Achieve Metadata. Additionally, we found that 67.4% of the packages that depend on `debhelper` as a build dependency are reproducible.

On the other hand, for at least 13 builds tested spread across six years (excluding 2020), the build was not reproducible, potentially impacting the build

[67] `https://salsa.debian.org/reproducible-builds/reproducible-notes`
[68] `https://bugs.archlinux.org/`

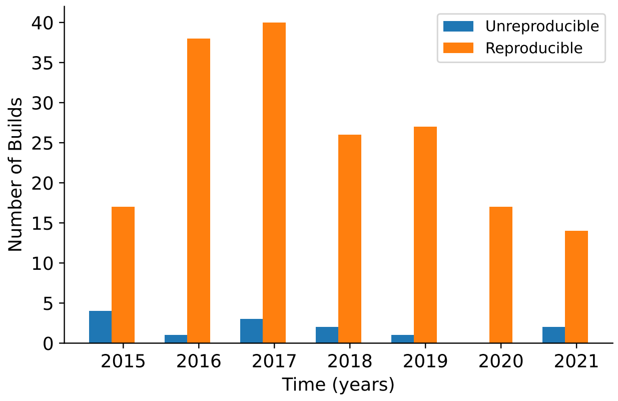

Fig. 8: Reproducibility Status of Debhelper Package

reproducibility of hundreds or thousands of packages depending on `debhelper`. Similar patterns as in Figure 8 have been seen for the two other most influential build dependencies that belong to the *devel* domain (`cdbs` and `cmake`). However, Figure 9 shows that `pkgconf` has been built reproducibly only once for the past three years. Furthermore, the latest build data performed in the year 2021 reveals that 1,558 packages (38.31%) that have `pkgconf` as a build dependency are built unreproducibly. This result illustrates our correlation results for how a package's reproducibility status depends on its build dependencies' reproducibility status.

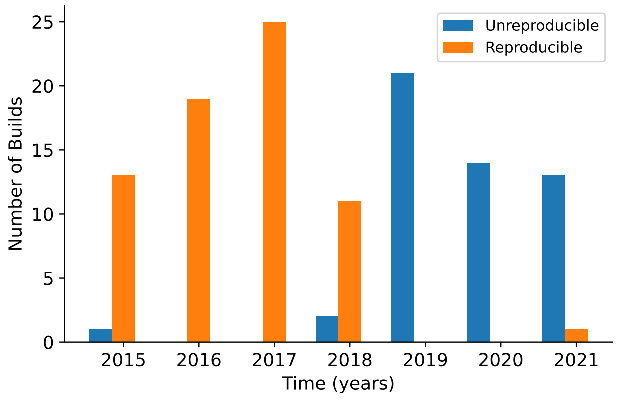

Fig. 9: Reproducibility Status of Pkgconf Package

Additionally, Table 12 shows that packages from seventeen package domains show a significant difference in the likelihood of having an unreproducible build depending on the build reproducibility of the packages' build dependencies ($p < 0.01$). We found that at least one package in each of the seventeen domains relies on GNU Compiler Collection (GCC)[69] for its compilation. Since GCC packages are utilized as build dependencies by a wide range of packages, we found several examples[70] in which developers seem to be concerned about issues related to Build Path, as illustrated by the quote below:[71]

> *We think it's not appropriate to patch all (3k+) of these packages to strip out -fdebug-prefix-map flags. This would involve adding quite complex logic to everyone's build scripts, and we have to adapt this logic every single time to that particular package.*
>
> *Also, in general CFLAGS is *supposed* to affect the compiler output, and saving it unconditionally is quite a reasonable thing for packages to do. If we tried to patch all of these packages, we would be turning "reproducible builds" in to a costly opt-in feature, rather than on-by-default that everyone can easily benefit from. So, we believe it is better to patch GCC centrally.*

Because GCC is an independent software project, proposals from the Reproducible Builds maintainers for fixing Build Path related issues may or may not be accepted by the GCC project maintainers; even if such patches are accepted, it may take a long time for these patches to be propagated to the end-users since GCC adopts feature-based releases that may take from a few months to years for such patches to be merged into GCC-based projects and distributed for use. In summary, the interdependence of various development teams, in this example, GCC and Reproducible Builds, can also have a significant impact on package reproducibility.

Implications We found in RQ1 that packages in the *devel* domain have a significantly higher survival probability for unreproducible packages after 30 days (74.45%) and 360 days (24.39%), implying that fixing packages from the *devel* package domain takes a long time. On the other hand, once fixed, these packages require the shortest time to become unreproducible again. A possible explanation for these earlier findings is hinted at by our findings about the impact of build reproducibility of a package's build dependencies. At the same time, this also provides an opportunity since the same build dependency might cause build (un)reproducibility for multiple packages, implying that fixing the issue once could solve many reproducibility problems.

In RQ2, we found that Package Dependency, probably the only root cause closely related to build dependencies, is one of the least frequent root causes and has the lowest impact in the entire distribution (Table 8). We observe that the reproducible status for the most influential build dependencies, such as `pkg-config`, has a significant impact on the reproducibility status of the package for which it is a dependency.

[69] https://gcc.gnu.org/

[70] https://tests.reproducible-builds.org/debian/issues/unstable/captures_build_path_via_assert_issue.html

[71] https://gcc.gnu.org/legacy-ml/gcc-patches/2016-11/msg00182.html

6.4 Architecture

Test results: The reproducibility of a package statistically significantly differs depending on the architecture for which the package is compiled ($p-value < 0.01$), with a very strong effect size ($0.8 < V < 0.9$).

Discussion: Architecture defines the hardware environment in which the system is compiled (Section 2). Different packages must be compiled differently to run on a specific hardware architecture. As seen in Table 11, a considerable number of the packages (7,167; 19.74%) are unreproducible on both the `armhf` and `arm64` architectures. After computing the Cramer's V estimator on the data, we found an effect size of 0.8942. Because $0.8 < V < 0.9$, this means a very strong correlation, which suggests that package reproducibility strongly depends upon the underlying hardware architecture.

Table 12 illustrates Cramer's V estimator for all packages in each package domain to understand the magnitude of the correlation between package reproducibility and packages built on specific architectures. We found that packages from 48 package domains show a significant difference ($p < 0.01$), with at least a strong effect size for all package domains. This result suggests that the overall observation of the role of architecture is also observed for all package domains separately.

Implications: RQ3 results strongly suggest that if a build is performed for a package in a given hardware architecture, it is not sure that the same package will build reproducibly on another hardware architecture. As a result, we must build each package across different hardware architectures to check for reproducibility, similar to our findings about the distribution factor.

Based on the results obtained from RQ3, it is understood that achieving complete build reproducibility fully within the control of a software package's own developers is often challenging. To comprehensively address this issue, the investigation of external factors that impact build reproducibility should be directed by researchers. Simultaneously, the refinement of tools and methodologies by practitioners engaged in the reproducible builds initiative should be diligently pursued to effectively differentiate between reproducibility issues that can be controlled by developers of a project and those that arise from external factors.

Summary of RQ3: We found strong effect sizes between the reproducibility status of packages and one external factor, i.e., hardware architecture. For the other three external factors, we found either weak correlation (build and their runtime dependencies) or negligible correlation (distribution), apart from some domains, which raises awareness of the need to perform builds more intensively.

7 Threats to Validity

Construct Validity: We opted for a time window of 720 days for performing the survival analysis of RQ1 (Section 4). Due to the unequal build data size for Arch Linux (5 years of build data) and Debian (10 years of build data), we decided to compare the build data of the first 720 days. We discarded packages built multiple times on the same day, as mentioned in Section 3.2 since their reproducibility status is likely to change multiple times in a short time. For instance, we initially had 37,858 packages for Event E1 analysis but discarded 343 out of 37,858 (0.9%) unreproducible Debian packages and 37 out of 8,529 (0.4%) unreproducible Arch Linux packages. Nevertheless, the discarded packages represent less than 0.8% of all packages, which suggests that discarding packages had little impact on our study.

The RQ3 analysis (Section 6) required identifying packages appearing in both Arch Linux and Debian distributions. We found 4,108 packages with the same name in both distributions. However, not all matching names imply having the exact package implementation. Remarkably, the package size, version, and functionality sometimes vary across distributions. Two examples of packages are `python-keyutils` (version 0.6-2 in Debian's Unstable suite vs. 0.6-6 in Arch Linux's Community suite) and `libvrt` (version 7.6.0-1 in Debian's Unstable suite vs. 7.8.0-1 in Arch Linux's Community suite). Variations are probably due to each distribution's different release mechanisms, though further investigation is required. Unfortunately, computing all variation occurrences is unfeasible because it requires skills to understand and identify whether each package performs similar functionality in both distributions.

For RQ1, RQ2, and RQ3, we use the Debian Unstable suite. We decided to use the Unstable suite for all our analysis since it consists of 37,858 (99.45%) of 38,064 packages tested for reproducibility. Furthermore, packages belonging to the Unstable suite undergo constant development by the Debian community. Changes in the source code may lead to unreproducibility; therefore, we studied the Unstable suite in all three experiments.

Internal Validity: We were inspired by the approach of previous studies [41] to perform descriptive analysis on the RQ2 data. The first three paper authors carefully revised the manual labeling procedures of RQ2 (Section 5) in at least three meetings. The first two authors performed the manual labeling independently based on procedures documented in recent work [41]. Additionally, the first two paper authors built together the taxonomy of root causes of unreproducible packages (details in Section 3.3). The third author has validated the taxonomy, and his feedback helped us address questions regarding the most appropriate way to group root causes into major categories. The taxonomy identified in our research was not only discussed with the reproducible builds community, but was also adopted by the official reproducible builds website,[72] complete with our definitions and prominent examples.[73]

The first two authors double-checked the spreadsheet created to support manual labeling and fix missing information problems. We computed agreement on the independent labeling outputs to mitigate human biases regarding the root causes identified by each author. We then had two meetings to discuss cases of disagreement, i.e., whenever the authors disagreed about the root cause behind a specific known issue related to reproducible builds. We expected to reach a consensus on the final labeling and improve our study's reliability. Following these discussions, we also sought feedback from experts to further enhance the accuracy and robustness of our findings.

[72] `https://reproducible-builds.org/docs/plans/`

[73] `https://reproducible-builds.org/docs/env-variations/`

We extracted the issues for analysis from the Reproducible Builds database[74] on October 11, 2021. Since issues change over time, our study targets a particular data snapshot. Since only four issues were added three months after (data of January 11, 2022), we assume that changes do not represent a significant threat to validity.

External Validity: We are aware that our quantitative results regarding the root causes may not generalize to all software projects. We stress that the Reproducible Builds project has more than ten years of build data. Furthermore, we see an increase in the projects participating in this initiative, indicating that unprecedented issues regarding reproducible builds (and root causes) may appear in the future. Still, it is worth mentioning that we manually labeled all 397 issues reported until late 2021, thereby covering 100% of the data available at that time, as described in Section 3.3.

Moreover, our quantitative results may not generalize to all of OSS. We discussed in Section 2.1 that the Reproducible Builds project comprises 33 OSSs of different natures. Five of the 33 OSSs are Linux distributions: Alpine, Arch Linux, Debian, OpenSUSE, and OpenWrt. However, RQ1 and RQ3 analyses target Arch Linux and Debian only because these are the only ones whose communities performed multiple build cycles for longer than one year. In addition, Debian packages can be tested on multiple architectures such as `amd64` and `i386`, while Arch Linux only supports the `x86_64` architecture. Finally, the two distributions analyzed vary significantly in the number of packages: 8,529 for Arch Linux against 37,858 for Debian. Hence, one could argue that our study results may not generalize even for the specific context of Linux distributions and architectures.

Conclusion Validity: This threat is concerned with the degree to which other research teams would reach a similar conclusion based on the given data and results. Our study is unlikely to be affected by the threat mentioned above, for two reasons: (1) All quantitative analyses performed in this study are supported with statistical tests that have been used previously with similar purposes. For instance, the survival analysis is used to determine the survival probabilities for open source projects [37], database framework usage in Java [13], and Debian package incompatibilities [8]. Additionally, our conclusion on the correlation between the reproducibility status of packages and the external factors is based on statistical significance and not incidental relationships. (2) Our qualitative analysis, performed by the first two authors, involves categorizing unreproducible build issues into root causes under the supervision of the third author. Furthermore, in case of complete uncertainty during the categorization of issues, we asked relevant questions to the industry experts about reproducible build issues via the mailing list. Moreover, we also performed a quantitative impact analysis of our categorized root causes focused on the #packages affected by these root causes. To ensure the rigor and accuracy of our findings, we also incorporated feedback from experts in the field after our initial analyses.

[74] `https://tests.reproducible-builds.org/debian/index_issues.html`

8 Related Work

Previous work highlights the problems related to trusting software binaries [39,42]. On the one hand, there is a lack of verifiability of software binaries from their source code. Previous studies [6,17,36,38,40] confirm that developers were not cognizant of verifiability of the software binaries while developing their build systems. On the other hand, there have been instances where malicious code has been injected during software builds [31,32,43]. We discuss below the papers closely related to our study, either related to the research questions, or related empirical studies based on reproducible builds.

Butler et al. [5] performed qualitative analyses of the benefits and challenges of the Reproducible Builds project in industry settings using interviews conducted with employees of 12 companies, including managers and software practitioners. The study revealed that reproducible builds are a good practice from a security point of view. However, the interviews also reveal that reproducible builds have a minimal business impact and are not very attractive from a customer's point of view. Additionally, the interviews reveal that the cost involved in maintaining and performing reproducible builds is a challenge that might inhibit its adoption. In our study, we not only provide empirical evidence on the maintenance cost (in terms of #days) for developers to fix an unreproducible package, but we also provide developers with 16 root causes of unreproducibility.

Maes-Bermejo et al. [18] performed and verified builds of 139,389 commits from past snapshots of 79 Java software projects. The study reveals that 63.45% of the errors that lead to the build failure are caused by build dependencies and external artifacts that are required to perform builds, not being resolved. The study claims that build compilation errors (2.92%) have a minor contribution for the builds to fail compared to the non-resolution of the build dependencies. Congruently, we emphasize in our study that modern software is part of a larger software ecosystem where each project depends on another to function efficiently. Furthermore, our study stresses that the reproducibility status of a project, is not wholly under the control of the project members. Rather it depends on external factors like the *distribution* in which a package is released, the *build dependencies* of a package, the *runtime dependencies* of the build dependencies, and the hardware *architecture* (RQ3, Section 6).

de Carné et al. [6] verify builds for a security-critical open source project – TrueCrypt. They compare 16 versions of the official software binary, including the major and minor versions, with the software binaries generated by performing builds on the source code. The authors fix issues related to Architecture, Build Path, and Build Timestamp to make the builds reproducible. However, there is no information about the frequency of such issues concerning the different versions considered nor about the prevalence of said issues across larger open-source projects. Furthermore, the efforts to fix the abovementioned issues introduce a delay for builds to become reproducible. In comparison, our study on build data from Linux distribution packages identifies a median delay of 104 days for Debian and 12 days for Arch Linux to become reproducible. Furthermore, our analysis of hundreds of releases for packages in both distributions expands the taxonomy of 6 root causes of build reproducibility to 16 root causes.

Goswami et al. [14] analyzed the reproducibility of the build of 3,390 versions of 226 NPM packages. The study signifies that 38% of the NPM packages are considered unreproducible and that end-users must not trust binaries from the NPM repository. Furthermore, to examine the differences between builds for unreproducible packages, the authors found that out of 19 randomly selected unreproducible builds, eight were due to specifying deprecated package versions. This result is interesting as Package Information is one of the root causes we discovered in our manual study of RQ2 (Section 5.1). Furthermore, the paper's authors identify three root causes for the unreproducible builds – Package Dependency, Filesystem Ordering, and Randomness. However, there is no mention of the frequency of such issues; therefore, the impact of each of the aforementioned root causes cannot be determined. In our study, we expand the taxonomy to 16 root causes and calculate the frequency and impact of the specified root causes (RQ2, Section 5.2).

Lamb et al. [17] broadly explain the reproducible build process through the experience of making Debian project reproducible and identify six root causes for unreproducible builds – Build Path, Build Timestamp, Filesystem Ordering, Achieve Metadata, Randomness, and Uninitialized Memory. This study reveals the fixes related to these six root causes. However, this study is an experience report from an industry expert in the reproducible builds community without empirical data. Additionally, the study confirmed that Build Timestamp is the most impactful root cause. In contrast, our study confirmed that Build Timestamp is the most frequent root cause. According to our study, Build Path (RQ2, Section 5.2) affects the most packages, which makes it the most impactful root cause. Additionally, in our study, we found that two root causes – User Information and Architecture Information are more impactful in terms of the number of packages affected than the ones mentioned in the recent study (RQ2, Table 8).

To fix an unreproducible build performed for a package, developers need to inspect the differences between the two builds (Section 1). Once they identify the files that are the cause for such differences, they perform fixes to those files in order to achieve a reproducible build. Previous work [36] proposes a tool, RepLoc, that ranks the source files responsible for unreproducible builds in Debian packages. Using the RepLoc tool revealed that `Makefile`s typically cause unreproducible builds (rank 1). A `Makefile`[75] consists of build instructions specified to generate a software binary. These files can be auto-generated by using packages like `cmake` and `automake`, which, according to Table 3, belong to the *devel* package domain. This domain shows a moderate to strong statistically significant difference in terms of build reproducibility with two of the external factors (Architecture, Distribution) taken into consideration (RQ3, Section 6.2, Section 6.4). This implies that packages from the *devel* must be prioritized for fixing since they are significant for other packages to become reproducible.

9 Conclusion

This paper introduces an empirical study that combines qualitative and quantitative methods to systematically investigate unreproducible builds in ecosystem settings, particularly the Arch Linux and Debian distributions. Our qualitative analysis expanded the current knowledge on root causes of unreproducible builds

[75] `https://wiki.debian.org/Makefile`

to a taxonomy of 16 different causes, some impacting hundreds of packages simultaneously. The correlation analysis shows that the reproducibility of a package strongly differs across the hardware architecture that packages are compiled on. Survival analysis results suggest that reproducible packages are 25.75% likely to survive over one year across both distributions.

Our quantitative analysis partially addresses David Wheeler's [76] concerns about how developers prioritize packages for fixing them into reproducible packages. We found that within a year, the survival probabilities in application-level package domains to remain unreproducible are 12.9% higher than in system-level package domains. Once fixed, reproducible packages from the application-level package domains have a 5.67% greater survival probability of remaining reproducible when compared to system-level package domains. However, several things are still missing to fully address Wheeler's concern about prioritization of reproducibility checking and fixing of projects' builds. For example: (1) surveys involving qualitative and quantitative analysis to understand the characteristics of package developers use to prioritize them and (2) mining issues reported by upstream developers to understand the decisions they made to identify essential packages to fix.

10 Conflict of Interest

All authors declare that there is no conflict of interest.

11 Data Availability Statement

The datasets generated and analyzed during the study are available from the corresponding author in a GitHub repository.[77]

References

1. Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E.: Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), p. 385–395 (2017)
2. Adams, B., Kavanagh, R., Hassan, A.E., German, D.M.: An empirical study of integration activities in distributions of open source software. Empirical Software Engineering **21**(3), 960–1001 (2016)
3. Allison, P.D.: Survival Analysis Using SAS: A Practical Guide, 2 edn. SAS Institute (2010)
4. Brooks, F.P.: The mythical man-month. Datamation **20**(12), 44–52 (1974)
5. Butler, S., Gamalielsson, J., Lundell, B., Brax, C., Mattsson, A., Gustavsson, T., Feist, J., Kvarnström, B., Lönroth, E.: On business adoption and use of reproducible builds for open and closed source software. Software Quality Journal pp. 1–33 (2022)
6. de Carné de Carnavalet, X., Mannan, M.: Challenges and implications of verifiable builds for security-critical open-source software. In: Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC), p. 16–25 (2014)
7. Chowdhury, M.A.R., Abdalkareem, R., Shihab, E., Adams, B.: On the untriviality of trivial packages: An empirical study of npm javascript packages. IEEE Transactions on Software Engineering pp. 1–15 (2021)

[76] `https://linuxfoundation.org/blog/preventing-supply-chain-attacks-like-solarwinds/`

[77] `https://github.com/SAILResearch/replication-21-rahul_bajaj-reproducible_builds-code`

8. Claes, M., Mens, T., Di Cosmo, R., Vouillon, J.: A historical analysis of debian package incompatibilities. In: Proceedings of the 12th Working Conference on Mining Software Repositories (MSR), pp. 212–223 (2015)
9. Decan, A., Mens, T., Claes, M.: On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proceedings of the 10th European Conference on Software Architecture Workshops (ECSAW), pp. 21:1–21:4 (2016)
10. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th international conference on mining software repositories, pp. 181–191 (2018)
11. Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer (2008)
12. Fried, L.: Team size and productivity in systems development bigger does not always mean better. Journal of Information Systems Management **8**(3), 27–35 (1991)
13. Goeminne, M., Mens, T.: Towards a survival analysis of database framework usage in java projects. In: Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 551–555 (2015)
14. Goswami, P., Gupta, S., Li, Z., Meng, N., Yao, D.: Investigating the reproducibility of npm packages. In: Proceedings of the 2020 International Conference on Software Maintenance and Evolution (ICSME), pp. 677–681 (2020)
15. Kaplan, E., Meier, P.: Nonparametric estimation from incomplete observations. Journal of the American Statistical Association **53**(282), 457–481 (1958)
16. Koen, R., Olivier, M.S.: The use of file timestamps in digital forensics. In: ISSA, pp. 1–16. Citeseer (2008)
17. Lamb, C., Zacchiroli, S.: Reproducible builds: Increasing the integrity of software supply chains. IEEE Software **39**(2), 62–70 (2021)
18. Maes-Bermejo, M., Gallego, M., Gortázar, F., Robles, G., Gonzalez-Barahona, J.M.: Revisiting the building of past snapshots—a replication and reproduction study. Empirical Software Engineering (EMSE) **27**(3), 1–26 (2022)
19. Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: Proceedings of the 21st International Conference on Automated Software Engineering (ASE), pp. 199–208 (2006)
20. Mäntylä, M.V., Adams, B., Khomh, F., Engström, E., Petersen, K.: On rapid releases and software testing: A case study and a semi-systematic literature review. Empirical Software Engineering **20**(5), 1384–1425 (2015)
21. Mao, A., Mason, W., Suri, S., Watts, D.J.: An experimental study of team size and performance on a complex task. PloS one **11**(4), e0153048 (2016)
22. Massacci, F., Jaeger, T., Peisert, S.: Solarwinds and the challenges of patching: Can we ever stop dancing with the devil? IEEE Security & Privacy **19**, 14–19 (2021)
23. Maste, E.: Reproducible builds in freebsd. In: Proceedings of 11th Asian Conference on BSD Based Systems (AsiaBSDCon), pp. 1–8 (2017)
24. McHugh, M.: Interrater reliability: The Kappa statistic. Biochemia Medica **22**(3), 276–282 (2012)
25. McIntosh, S., Adams, B., Nagappan, M., Hassan, A.E.: Mining co-change information to understand when build changes are necessary. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 241–250 (2014)
26. Michlmayr, M., Hunt, F., Probert, D.: Release management in free software projects: Practices and problems. In: Proceedings of the 2007 International Federation for Information Processing International Conference on Open Source Systems (IFIPAICT), vol. 234, pp. 295–300 (2007)
27. Miller, P.: Recursive make considered harmful. AUUGN Journal of AUUG Inc **19**(1), 14–25 (1998)
28. Mirhosseini, S., Parnin, C.: Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE), pp. 84–94 (2017)
29. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering, pp. 284–292 (2005)
30. Nussbaum, L., Zacchiroli, S.: The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 52–61 (2010)

31. Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's knife collection: A review of open source software supply chain attacks. In: Proceedings of the 2020 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, vol. 12223, pp. 23–43 (2020)
32. Ohm, M., Sykosch, A., Meier, M.: Towards detection of software supply chain attacks by forensic artifacts. In: Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES), pp. 1–6 (2020)
33. Plackett, R.: Karl Pearson and the Chi-Squared test. International Statistical Review **51**(1), 59–72 (1983)
34. Raymond, E.: The cathedral and the bazaar. Knowledge, Technology & Policy **12**(3), 23–49 (1999)
35. Rea, L.M., Parker, R.A.: Designing and conducting survey research: A comprehensive guide, 1 edn. John Wiley & Sons (2014)
36. Ren, Z., Jiang, H., Xuan, J., Yang, Z.: Automated localization for unreproducible builds. In: Proceedings of the 40th International Conference on Software Engineering (ICSE), p. 71–81 (2016)
37. Samoladas, I., Angelis, L., Stamelos, I.: Survival analysis on the duration of open source projects. Information and Software Technology **52**(9), 902–922 (2010)
38. Shi, Y., Wen, M., Cogo, F.R., Chen, B., Jiang, Z.M.J.: An experience report on producing verifiable builds for large-scale commercial systems. IEEE Transactions on Software Engineering (2021)
39. Thompson, K.: Reflections on trusting trust. Communications of the ACM **27**(8), 761–763 (1984)
40. Vu, D.L., Pashchenko, I., Massacci, F., Plate, H., Sabetta, A.: Towards using source code repositories to identify software supply chain attacks, p. 2093–2095 (2020)
41. Wang, Z., Zhang, H., Chen, T.H., Wang, S.: Would you like a quick peek? Providing logging support to monitor data processing in big data applications. In: Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 516–526 (2021)
42. Wheeler, D.A.: Countering trusting trust through diverse double-compiling. In: Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC), pp. 1–13 (2005)
43. Yan, D., Niu, Y., Liu, K., Liu, Z., Liu, Z., Bissyandé, T.F.: Estimating the attack surface from residual vulnerabilities in open source software supply chain. In: Proceedings of the 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 493–502 (2021)
44. Zerouali, A., Constantinou, E., Mens, T., Robles, G., González-Barahona, J.: An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse, pp. 95–110 (2018)
45. Zerouali, A., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the diversity of software package popularity metrics: An empirical study of npm. In: Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 589–593 (2019)