Reproducibility is Not Enough: Artifact Verifiability in Decentralized-Build Package Ecosystems

Oreofe Solarin
Case Western Reserve University
Cleveland, United States
ons8@case.edu

Kelechi Kalu, James C. Davis, Paschal Amusuo
Purdue University
West Lafayette, United States
{kalu,davisjam,pamusuo}@purdue.edu

***Abstract*—Reproducible and verifiable builds increase trust in distributed software artifacts by enabling independent parties to detect artifacts produced by compromised build or release pipelines. However, artifact verification requires more than deterministic builds: a verifier must also recover the source state, build environment, dependencies, and build instructions that produced the artifact. Decentralized-build ecosystems make this difficult because artifacts are produced through heterogeneous tools, maintainer-controlled workflows, and fragmented metadata. As a result, it remains unclear how often artifacts in these ecosystems can be independently verified.**

This paper studies artifact verifiability across four popular decentralized-build package ecosystems. We define an independent verifier model that relies only on registry-derivable metadata and an artifact comparison model with tiered equivalence levels. We implement these models in an Artifact Verification Pipeline and use it to measure artifact verifiability across the target ecosystems. Our results show that, beyond build determinism, verifiability is limited by missing source and build metadata, implicit release transformations, and unconventional build practices. Provenance attestations and embedded VCS metadata improve verification, but they do not provide complete rebuild specifications. These findings identify concrete metadata gaps and ecosystem-level changes needed to make artifact verification practical at package-registry scale.

***Index Terms*—Reproducible builds, artifact verifiability**

I. Introduction

Reproducible builds provide an important foundation for trust in released software [1], [2], [3]: given the same source and build instructions, an independent party should produce an identical artifact, detecting build-time tampering and confirming that a released artifact corresponds to the reviewed source [4], [5]. Open-source communities, standards, and government agencies accordingly recommend them as a supply-chain defense [6], [7], [8]—especially for ecosystems such as npm and PyPI, which face a growing wave of attacks on compromised build and release pipelines [9], [10], [11], [12].

However, reproducibility alone is insufficient for verifying the integrity of released artifacts [13], [14]. An independent verifier must also know the source revision, build environment, and build instructions that produced the artifact in order to reproduce it. Centralized-build ecosystems such as Debian [1] and Nix [15] make this easier by preserving or publishing artifact-bound build specifications through ecosystem infrastructure. In contrast, *decentralized-build* ecosystems rely on third-party build infrastructure and diverse release practices, leading to fragmented or missing source and build metadata [16], [17]. Prior work has largely avoided this problem by assuming the relevant metadata is available [3], relying on human augmentation [18], or treating artifact verification as out of scope [19]. As a result, it remains unclear whether independent artifact verification is feasible in source-oriented package ecosystems.

This paper addresses this gap empirically. We focus on decentralized-build package ecosystems, such as npm and PyPI, because the growing prevalence of software supply-chain attacks highlights the urgency of reproducible builds and verifiability in these ecosystems [20], [21]. We define two models for evaluating verifiability: an independent verifier model using only registry-exposed and registry-derived metadata and an artifact comparison model that allows graded levels of artifact equivalence. We implement these models in AVP, an Artifact Verification Pipeline that recovers rebuild metadata, reconstructs build environments, rebuilds artifacts, and compares rebuilt outputs against registry artifacts using tiered equivalence levels. We use AVP to measure verification rates, failure causes, and the effects of provenance attestations, artifact characteristics, and development choices across the four ecosystems: Crates.io, npm, PyPI, and RubyGems.

Our results show that artifact verifiability is low and uneven: across ecosystems, roughly 53–87% of completed rebuilds verify, and recovering the source commit is the dominant obstacle. Provenance attestations improve recovery, but their benefit comes mostly from the development practices of the projects that adopt them rather than the recorded metadata itself; most remaining mismatches trace to unpinned build-tool and dependency versions, CI/CD-generated files, and source modifications, and many are small or metadata-only—byte identity alone is too coarse for source-oriented artifacts. How a package is *built*, its packaging determinism, predicts verifiability far better than what it *is*, with native compilation a structural limit. Finally, against the state-of-practice OSS-Rebuild, AVP verifies about twice as many artifacts and, unlike curated rebuild platforms, scales to the popular long tail.

In summary, this paper makes three contributions:

- We define new models for independent artifact verifiability and implement them in AVP for recovering rebuild metadata, rebuilding and verifying published software artifacts.

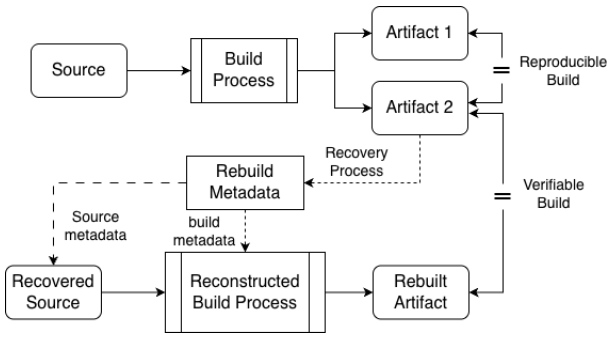

Fig. 1. **Reproducible** builds ensure the same source code produces identical artifacts. **Verifiable** builds require an independent party to recover the source revision, build environment, and build instructions needed to reproduce that artifact. We study the artifact verifiability in decentralized-build ecosystems.

- We empirically study the verifiability of 34,005 packages across four source-oriented package ecosystems.
- We identify the metadata gaps, release practices, and ecosystem-level changes that affect verifiability.

Significance: This paper provides the first cross-ecosystem measurement of independent artifact verifiability. Verifiability in decentralized-build ecosystems remains limited by insufficient rebuild metadata. By identifying the metadata gaps and packaging practices that prevent verification, the study provides guidance for registries, standards, and tooling to support verifiability as a practical software supply-chain defense.

II. Background and Related Concepts

Here we review packages and publishing models, associated security risks, mitigation by reproducible and verifiable builds, and the challenges of verifying artifacts in open-source package ecosystems.

A. Package Ecosystems and Artifacts

Software *packages* are reusable software units that provide functionality to downstream applications [22]. A *package artifact* is the concrete file published for a package version and installed by its users. It contains the package's code, either source directly, or code transformed through compilation, bundling, transpilation, or minification, along with metadata describing the artifact. A *package ecosystem* comprises the packages for a given language or system, the maintainers who publish them, and the tools used to install them [22].

Ecosystems differ in where artifacts are built. In *centralized-build* ecosystems, typically operating-system distributions, ecosystem-managed infrastructure builds artifacts from submitted source packages—Debian, for example, builds `.deb` artifacts from signed source packages on its own daemons [23]. In *decentralized-build* ecosystems such as npm [24] and PyPI [25], maintainers build artifacts themselves—on their own machines, CI/CD services, or third-party infrastructure—and upload the finished artifact. Because the build runs outside ecosystem-controlled infrastructure, the registry retains no authoritative record of how each artifact was produced [3].

B. Package Build and Publishing Models and Security Risks

Package ecosystems define the metadata and tooling maintainers use to build and publish artifacts. Most require a package manifest in the source directory specifying the name, version, dependencies, and build configuration (`Cargo.toml` for Crates.io, `package.json` for npm, `pyproject.toml` for PyPI), and a `.gemspec` for RubyGems; in monorepos, the subdirectory holding the manifest identifies the package. Ecosystems also provide canonical build and packaging interfaces while permitting third-party build systems and project scripts, such as `npm pack` for npm or `python -m build` for PyPI. To publish, maintainers can either upload pre-built artifacts to registries or publish from CI/CD workflows via trusted publishing [26], [27].

These models expose two security risks [28], [29], [30]. In a *build compromise*, an attacker subverts the maintainer's build tools, dependencies, scripts, or CI/CD to inject malicious code during artifact creation; in a *publishing compromise*, an attacker obtains credentials or tokens and uploads a malicious artifact directly. Recent npm and PyPI incidents exhibit both [11], [12], [31], [32], [33], [34] and in each the distributed artifact diverges from the reviewed source.

C. Reproducible Builds and Artifact Verification

Reproducible and verifiable builds provide independent evidence that a distributed artifact corresponds to its source code. A build is *reproducible* if, given the same source code, build environment, and build instructions, an independent party can produce a bit-for-bit identical copy of the artifact [1], [35]. *Verifiability* adds a recovery requirement: an artifact is verifiable if an independent party, starting only from the published artifact, can recover enough information about its source and build environment to reproduce an equivalent copy. The distinction is consequential. Reproducibility *assumes* the source revision, environment, and instructions are known; verifiability requires *recovering* them from the artifact and its metadata. Verifying an artifact therefore rests on two requirements: a build deterministic enough to reproduce, and rebuild information complete enough to drive that reproduction. Because it can detect artifacts altered during build or publishing, independent verification is recommended as a supply-chain defense by standards bodies [36], government agencies [7], researchers [1], and industry [35].

The determinism requirement is comparatively well understood: prior work has catalogued the common sources of build non-determinism—timestamps, build paths, filesystem ordering, archive metadata, and embedded randomness [37], [1], [2], [15]—and the reproducible-builds community has built tooling to remove them, such as `diffoscope` and `SOURCE_DATE_EPOCH`, yielding the high reproducibility rates reported in centrally built ecosystems such as Debian [1], [38] and Nix [15]. A complementary line of work relaxes byte-for-byte equivalence, accepting artifacts that preserve relevant structure or behavior [39]. The recovery requirement, by contrast, is far less settled, and it is where decentralized package ecosystems diverge from centrally built ones.

D. The Metadata Recovery Gap in Package Ecosystems

Even if build non-determinism is controlled, an independent verifier still needs the *rebuild metadata*—the source revision, build environment, and build instructions—that produced the published artifact. Centrally managed ecosystems can provide this metadata by construction. For example, Debian [1] and Nix [15] derive rebuild metadata from their managed build infrastructure and preserve it as part of the artifact or package specification. In decentralized-build ecosystems, however, artifacts are often produced outside the registry by maintainer-controlled CI/CD workflows, leaving registries without a complete record of the build inputs and process.

Several efforts attempt to close this metadata recovery gap. Maven Reproducible Central rebuilds Maven Central artifacts using contributor-provided build specification files, while recent work explores automated recovery of build metadata and generation of such specifications [19], [40]. These approaches use heuristics and provenance attestations [41] to recover source commit and CI/CD workflows to recover build commands. However, they are tailored to Maven and do not establish whether the recovered metadata is sufficient for artifact-level verification across source-oriented ecosystems.

Google's OSS Rebuild [18] extends rebuild efforts to Crates.io, npm, and PyPI, using similar metadata-recovery heuristics. However, it focuses on supported popular packages and reports using human augmentation when automation fails [42]. As a result, it remains unclear how far artifacts in decentralized-build ecosystems can be reproduced using only metadata available to an independent verifier.

III. Knowledge Gaps and Research Questions

Verifiable builds promise to mitigate software supply-chain attacks that exploit compromised build and release pipelines. However, existing verifiable-build systems either operate in ecosystems with centralized build infrastructure, where rebuild metadata is recorded during build, or rely on human intervention to recover missing metadata in decentralized ecosystems. As a result, we do not yet know whether artifacts in decentralized-build ecosystems can be verified automatically.

This paper asks: *To what extent can published artifacts in decentralized-build package ecosystems be independently verified against their source without human intervention?* We structure this question into four research questions:

RQ1. *Verifiability results:* What proportion of registry artifacts can be verified against their source?

RQ2. *Root causes:* What are the root causes of artifact verifiability failures?

RQ3. *Factor analysis:* How is verifiability affected by artifact characteristics and development choices?

RQ4. *Comparison:* How does AVP compare with state-of-practice rebuild systems?

IV. Models for Independent Artifact Verifiability

We model two aspects of independent artifact verification: a *verifier model*, to describe the capabilities of an independent verifier, and an *artifact comparison model*, describing how to compare published vs. rebuilt artifacts.

A. Verifier Model

We model a **verifier** as an independent entity that validates a published artifact using only registry-exposed or registry-derived information, without access to the original build machine, private CI/CD state, or undisclosed secrets. Examples include registries performing publication-time checks and downstream users validating artifacts before installation.

The verifier makes two design choices: how to recover rebuild metadata and how to rebuild the artifact. Our model focuses on fully-automatable and generalizable metadata recovery. It recovers metadata from three complementary sources: the published artifact, the registry-linked source repository, and registry-linked provenance when available. These sources provide evidence such as embedded source information, build-tool lock files, tags, commit history, manifests, source commits, and CI/CD workflows. We restrict recovery to these sources because they are available to independent verifiers and can scale across ecosystems and packages.

For rebuilding, our model adopts an independent builder approach [1]. Instead of replaying the original CI/CD workflow, the verifier uses ecosystem-level canonical build and packaging commands, such as `npm pack` for npm and `python -m build` for PyPI. This reduces dependence on potentially compromised workflow steps and tests whether artifacts can be reproduced from durable source and build metadata. However, it can fail when packages rely on custom workflow steps or package-specific build logic.

Unlike prior systems tailored to specific ecosystems or package sets, such as BuildGen for Maven [19] and OSS-Rebuild [18], our model intentionally restricts the verifier to ecosystem-level, registry-derived evidence to measure how far independent artifact verification can scale across decentralized-build package ecosystems.

B. Artifact Comparison Model

Artifact verification requires a criterion for comparing the published artifact with the independently rebuilt artifact. Package artifacts in decentralized-build ecosystems are mostly structured archives with inspectable contents, rather than flat byte sequences as modeled by prior work [39]. As a result, we define an **artifact comparison model** that uses tiered equivalence levels over package contents and metadata.

- *L1: byte-identical.* The published and rebuilt artifacts are byte-for-byte identical. This is the strongest level and matches the standard definition of reproducibility.
- *L2: content-identical.* The artifacts contain the same files with identical contents after normalizing archive-level metadata, such as timestamps, file ordering, compression parameters, ownership fields, and permissions.
- *L3: code-identical.* The artifacts match after excluding or normalizing known non-executable content differences, such as documentation, generated package metadata,

runtime-irrelevant lock files, or non-executable fields in build definition files.
- *L4: behavioral equivalence.* The artifacts differ syntactically, but source-code, binary, or semantic analysis shows that the differences do not affect execution. We treat L4 as out of scope because such analyses may be unavailable, incomplete, or unsound for arbitrary package artifacts.

Unlike prior systems that report only L1 (bit-identical) [18] or rebuild success [19], this model makes artifact comparison explicit and reports outcomes for multiple equivalence levels.

V. Design and Implementation of Artifact Verification Pipeline (AVP)

We implement AVP from the models in section IV for four decentralized-build ecosystems: Crates.io, npm, PyPI, and RubyGems. As shown in fig. 2, it operates in three stages: recovering rebuild metadata, rebuilding the artifact, and comparing the rebuild with the published artifact.

A. Stage 1: Recover Rebuild Metadata

The first stage recovers the metadata needed to rebuild an artifact: *source metadata*, to identify the source code to rebuild; and *build metadata*, to identify the tools, dependencies, timestamps, and commands needed to reconstruct the artifact.

1) Recovering Source Metadata: Source metadata consists of the repository URL, source commit, and package subpath. AVP recovers this metadata from three sources, in order of preference: provenance attestations, embedded source metadata, and heuristics.

For artifacts with provenance metadata, such as attestations (npm [43], PyPI [44], RubyGems) or trusted publishing information (Crates.io [27]), AVP extracts the repository URL, commit hash, and GitHub Actions workflow URI from the provenance attestation or trusted publishing information present in the artifact's registry metadata. In addition, for Crates.io packages with artifact-embedded source metadata (*e.g.,* `.cargo_vcs_info.json` [45]), AVP extracts the commit and package subpath from the embedded file.

When these sources are unavailable, AVP combines heuristics from prior work [19], [18], [40]. It recovers repository URLs from registry fields such as project URL, homepage, and description [17], [46], then normalizes and validates candidate URLs. It recovers commits using three ordered strategies: *tag matching* against the artifact version; *version-string search* in commits within `[-7d, +2d]` of artifact upload time; and, as a final fallback, the *latest default-branch commit* before upload. Finally, it identifies the package subpath by locating the directory whose package definition matches the artifact name, using lexical matching when no exact match exists.

2) Recovering Build Metadata: Build metadata consists of the tools, versions, dependencies, commands, and timestamps needed to rebuild the artifact. AVP recovers it from ecosystem-specific files in the artifact and source tree—for example, the build backend, its version, and dependencies from PyPI's `pyproject.toml`, or the package manager from npm's `package.json` and lockfiles—and then constructs the build and packaging commands from the ecosystem and recovered build tool.

Finally, to reduce non-determinism from file modification times, AVP uses the published artifact's uniform modification time as the build timestamp when one exists, and otherwise the recovered source-commit timestamp, following reproducible-builds guidance [47].

B. Stage 2: Rebuild the Artifact

The second stage uses the recovered metadata to produce an independent rebuild. Following our independent verifier model (section IV-A), AVP uses ecosystem-default build and packaging commands to rebuild the artifact.

AVP first prepares up to three variants of the checked-out source tree: without Git metadata, with Git metadata, and with Git metadata plus build-tool `pretend-version` variables, to accommodate tools that derive versions or timestamps from version control. It then installs the recovered build tools and dependencies, sets `SOURCE_DATE_EPOCH` to the recovered timestamp, applies tool-specific setup (e.g., Corepack for pnpm), and runs the recovered build and packaging commands.

C. Stage 3: Compare Artifacts

Stage 3 compares each rebuilt artifact with the published one and reports the strongest level reached. L1 compares artifact hashes; L2 compares per-file content hashes while allowing archive-metadata differences; L3 checks that residual differences fall within a small, fixed catalog of non-executable metadata files and fields built from ecosystem documentation (e.g., PyPI's `RECORD`, `METADATA`, `WHEEL`, `PKG-INFO`, and generated `_version.py` files, npm's `package.json` metadata fields, and Cargo's `.cargo_vcs_info.json`, and `Cargo.lock`). Artifacts reaching none are not verified; we do not operationalize L4.

VI. Artifact Verifiability Methodology

This section describes our methodology for answering the research questions in section III.

A. Experimental Setup

1) Package Ecosystem Selection: We study four source-oriented ecosystems: Crates.io, npm, PyPI, and RubyGems. These are among the largest programming-language ecosystems and frequent targets of supply-chain attacks [29]; they also match the source-oriented ecosystems studied by recent reproducible-build work [3] and supported by OSS Rebuild [18].

2) Dataset Construction and Sampling: For each ecosystem, we enumerate packages from the registry and build a package-level dataset of metadata, source-repository information, and provenance-attestation status. We recover repository URLs from registry fields such as project URL, homepage, and description, following prior work [17], and query each ecosystem's provenance API for whether the latest artifact is attested.

Two properties of the dataset drive our sampling. Attestation adoption is low (2.0-6.6%), so a random ecosystem-level

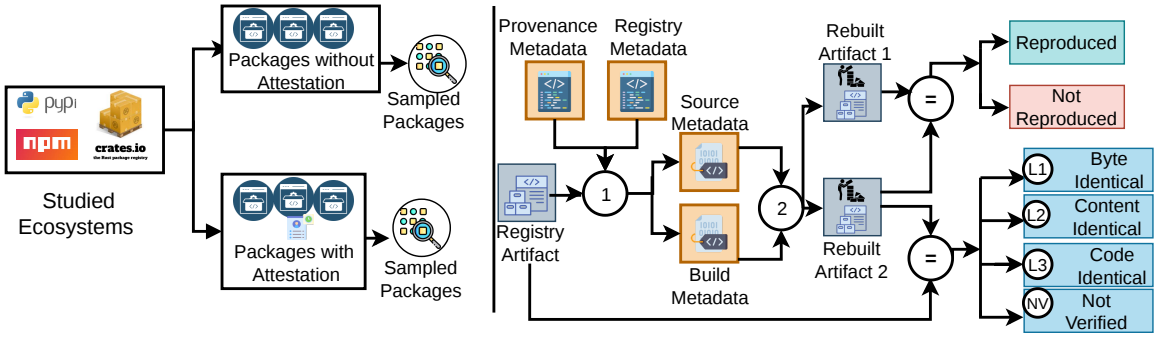

Fig. 2. Methodology Overview and Artifact Verification Pipeline Design and Implementation.

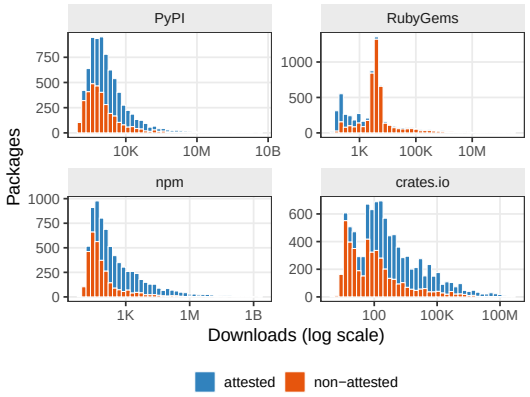

Fig. 3. Download distribution of the sampled packages per ecosystem.

sample would contain too few attested artifacts to evaluate attestations. Many packages lack valid repository URLs (14.7-42.4%), which we report as an ecosystem-level source-recovery limitation. We exclude packages with no registry-linked public repository candidate from the rebuild experiment; however, AVP may still fail source recovery for sampled packages when the linked repository is unavailable, invalid, lacks a matching package subpath, or no source commit can be recovered. We therefore sample within attested and non-attested strata separately, drawing up to 4,500 packages per stratum, above the sample needed for a 95% confidence level and 1.5% margin of error under the conservative infinite, population assumption of prior work [3]. For crates.io and RubyGems we take the full attested set—5,206 and 1,799 packages, respectively—since RubyGems has fewer than 4,500 attested packages in total, and for crates.io, we use all attested crates rather than subsampling. For each sampled package we select its most recent artifact as of May 2026, yielding 34,005 artifacts (table I); Figure 3 shows the download-rate distribution.

3) Experiment environment: We run all rebuilds and measurements on an IBM Cloud bare-metal server profile: 48 physical cores, 384 GB RAM, and a 960 GB local NVMe disk. The server runs Ubuntu Linux 24.04 LTS (x86-64).

B. RQ1: Proportion of Verified Software Packages

RQ1 measures the proportion of packages in each ecosystem that can be independently verified under the models in section IV. For each selected artifact, we run AVP to recover rebuild metadata, reconstruct the build environment, rebuild the package twice, and compare the rebuilt artifacts against the published one. We run three experiments: (i) the non-attested ecosystem sample with heuristic metadata recovery, (ii) the attested sample with heuristic source recovery, and (iii) the attested sample with attestation-based recovery. The latter two isolate the contribution of provenance attestations to verifiability.

We report two metrics per experiment: the *completion rate* (the fraction of artifacts for which AVP produces a comparison result, together with the reasons for pre-comparison failures), and the *equivalence rate* (the fraction reaching each level). We consider an artifact *verified* at L1, L2, or L3 (section IV-B).

C. RQ2: Root Causes of Non-L1 Verifiability Failures

For artifacts that did not reach L1, we qualitatively analyzed the observed differences and their causes. We restricted this to artifacts with provenance metadata, whose attestations or trusted-publishing records identify the source commit and CI/CD workflow, enabling inspection and partial reconstruction of the original build.

In the first stage, two authors independently open-coded 48 randomly sampled non-L1 artifacts: for each, they examined the `diffoscope` [48] report to identify observable differences, then inspected the source commit and GitHub Actions workflow to infer the underlying source-state, build-process, or environment cause (30-60 minutes per artifact). They reconciled their findings into a shared taxonomy and codebook defining each symptom, its possible causes, and ecosystem-specific manifestations.

In the second stage, we used a frontier coding agent to scale the analysis, requiring experimental validation for each proposed cause: the agent corrected the hypothesized cause in a Dockerfile matching our rebuild environment and checked whether the artifact's equivalence level improved, with a second agent reviewing each classification. Applied over three phases to saturation, this covered 134 further artifacts (35

TABLE I
DATASET AND PIPELINE COMPLETION PER ECOSYSTEM. COMPLETION RATE AND FAILURE REASONS ARE % OF SAMPLED ARTIFACTS.

	Crates.io	npm	PyPI	RubyGems
Sampled: attested	5,206	4,500	4,500	1,799
Sampled: non-attested	4,500	4,500	4,500	4,500
Total sampled	9,706	9,000	9,000	6,299
Completion rate	83.4%	61.6%	74.4%	74.9%
No source/repo	4.1%	15.5%	13.4%	15.8%
Clone/download	9.1%	2.5%	0.4%	2.1%
Build/install	3.4%	20.4%	11.2%	7.3%
Timeout/other	0.0%	0.0%	0.6%	0.0%

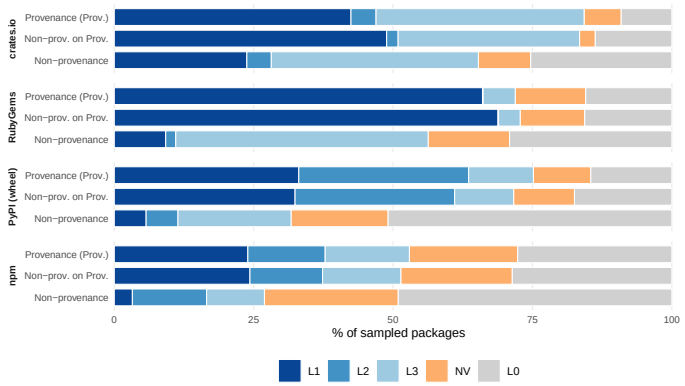

Fig. 4. Artifact equivalence by ecosystem and source-recovery strategy, over artifacts the pipeline could compare (completion rates in Table I).

from Crates/npm/PyPI and 29 from RubyGems). We report the resulting causes and symptoms, grouped by the rebuild metadata they affect, with frequencies across ecosystems.

D. RQ3: Impact of Artifact Characteristics and Development Choices

RQ3 tests whether six factors are associated with verifiability: artifact size, download rate, and age (artifact characteristics), and build tool, native code, and monorepo layout (development choices). The last three capture development choices that may complicate rebuilding: monorepos can obscure package subpaths, native extensions can depend on platform-specific environments, and build tools may introduce different sources of nondeterminism. We stratify continuous factors into quartiles and group categorical factors, comparing verifiability across bins.

E. RQ4: AVP vs State-of-Practice Rebuild Platforms

RQ4 compares AVP with OSS-Rebuild, Google's open-source platform that infers build definitions, rebuilds artifacts, and produces rebuild attestations for popular packages. We select it as open source, targeting our ecosystems, and operated at scale. We compare on Crates.io, npm, and PyPI, excluding RubyGems, which OSS-Rebuild does not support.

We draw 600 artifacts: 200 per ecosystem, split into two strata of 100 by OSS-Rebuild coverage. The supported stratum is sampled from packages OSS-Rebuild already tracks; the unsupported stratum from each ecosystem's most popular packages by download count, excluding any already supported. Both strata are thus widely-used software differing only in coverage, which separates how the platforms compare where OSS-Rebuild is tuned from how they compare on the popular long tail it does not support. This frame is independent of the attestation-stratified dataset used in our other RQs.

We rebuild each artifact with both platforms from the same published version: OSS-Rebuild via its tool, which infers a build definition, builds hermetically, and reports whether the result matches; AVP via our pipeline (fig. 2) on the same artifact and recovered source commit. Because OSS-Rebuild reports a binary match while AVP reports graded levels (section IV-B), we count an AVP artifact as verified at L1–L3, normalizing the same non-semantic differences OSS-Rebuild normalizes when deciding a match. We report, per ecosystem and stratum, each platform's verification rate and their agreement (verified by both, AVP only, OSS-Rebuild only, or neither); for artifacts verified by exactly one, we analyze the design choices: source recovery, build-environment inference, native-code handling, that explain the divergence.

VII. RESULTS

This section presents the results of our empirical study.

A. RQ1: Ecosystem Verifiability Results

Figure 4 reports artifact equivalence across ecosystems and source-recovery strategies, while Table I reports pipeline completion and failure causes. We separate these outcomes: *completion* measures whether AVP recovers the source and rebuilds the artifact, while *equivalence* measures how closely completed rebuilds match the published artifact.

Internal reproducibility: We first confirm that AVP's rebuilds are deterministic by building each package twice. These rebuilds were byte-identical for 93.0–100.0% of artifacts across ecosystems and content-identical for 94.8–100.0%, exceeding reported reproducibility rates for these ecosystems [3].

Heuristic recovery: For artifacts without provenance, AVP recovers the source commit heuristically (Section V-A1). Pipeline completion is 74.7%, 50.9%, 49.1%, and 70.9% for Crates.io, npm, PyPI, and RubyGems, respectively (Table I). The dominant failure is source-commit recovery, a limitation shared by prior rebuild systems and empirical studies that rely on similar heuristics [49], [18], [19], [3], [17], [46], [16]. Among artifacts that complete, 87.4% of Crates.io artifacts verify at L1–L3, compared with 52.9% for npm, 64.6% for PyPI, and 79.4% for RubyGems. Crates.io performs best on both completion and equivalence because crate artifacts embed source metadata that directly supports source recovery.

Provenance-based recovery: For provenance-bearing packages, AVP recovers the source commit directly from npm and PyPI attestations, Crates.io Trusted Publishing metadata, or RubyGems attestations. This improves both completion and equivalence. For example, npm completion rises from 50.9% to 72.4%, and equivalence among completed artifacts rises

from 52.9% to 73.2%. RubyGems similarly improves from 70.9% to 84.5% completion and from 79.4% to 85.1% equivalence, with comparable gains for Crates.io and PyPI (Figure 4). Once the source commit is known, remaining failures are dominated by build-execution errors and missing build dependencies rather than source recovery.

Isolating the effect of provenance metadata: Provenance-bearing packages may also be more popular, active, or mature, so their higher verifiability may not be caused by provenance metadata alone. To isolate the metadata effect, we hold the package set fixed and compare provenance-based recovery with heuristic recovery on the same provenance-bearing packages. Measured as the overall verified fraction of the sample, the direct gain from recorded provenance is small: 0.8, 1.5, 3.6, and -0.9 percentage points for Crates.io, npm, PyPI, and RubyGems, respectively. In contrast, the gap between heuristic recovery on provenance-bearing packages and heuristic recovery on the general sample is much larger: 18.2, 24.5, 39.9, and 16.5 points. Thus, most of the verifiability advantage of provenance-bearing packages reflects the build and release practices of adopting projects, not the provenance metadata itself.

RQ1 Takeaway

Verifiability is low and uneven across ecosystems: among completed rebuilds, 52.9–87.4% verify at L1–L3, and source-commit recovery is the dominant pipeline failure. Provenance improves source recovery and raises apparent verifiability, but holding the package set fixed shows that the recorded metadata contributes at most 3.6 percentage points. Provenance makes source recovery reliable; it does not, by itself, make artifacts verifiable.

B. RQ2: Root Causes of Non-L1 Verifiability Failures

Table II summarizes the observed root causes, their symptoms, and the reconstruction gaps they expose. For space, we discuss three representative causes.

Build tool or dependency version differences: The most common source of divergence was version drift in build tools or build dependencies. These differences produced either L3 metadata differences, such as changed lock files, or content differences in files generated by the build toolchain. For example, `tidesurf` 0.2.1 was originally built with Cython v3.2.3, while our rebuild resolved to Cython v3.2.6, producing different generated C files. In such cases, the relevant versions were not pinned in the source manifests, allowing dependency resolution to drift between the original build and the rebuild.

Generated files produced by CI/CD workflow: The second most common source of divergence was intermediate files generated by CI/CD workflow steps but not by the canonical build process used in our verifier model. For example, the npm `lingui-swc` 0.6.0 workflow ran `yarn build:ts` and `napi prepublish`, generating files under `dist/` that were included in the published artifact but absent from our rebuild. These cases show that artifact verification can fail when build-relevant workflow steps are not captured by registry-exposed metadata or ecosystem-standard build interfaces; prior work similarly finds published packages containing files absent from their source repositories [46].

One might expect that *replaying* the original workflow would close this gap. We evaluated this directly in a pilot: for 105 Crates.io and 105 npm artifacts sampled per outcome bucket, we re-ran each package's real release workflow using `act` [50] and compared the result to the published artifact.[1] Replay improved at most 4% of artifacts (Crates.io 1/105, npm 4/105) and closed essentially no existing divergences: every npm gain was coverage on a previously failed build rather than a match, and the sole genuine recovery (`ruckup` 0.9.1, which turned a mismatch into an exact L1 match) succeeded only because the workflow pinned a build-toolchain version our heuristic had guessed wrong. Replay is also fragile: even trivially reproducing artifacts fail to build under `act` in roughly 9 of 10 cases. Workflow replay therefore does not scale as a verification strategy; the one real gain points instead at build-environment pinning, which we return to in Section IX-A.

Source revision mismatch: Source revision mismatches occur when provenance metadata records a different commit from the one that produced the published artifact. For example, the Crates.io `aviutl2` 0.27.0 crate pointed to commit `7dcd140` and tag `0.27.0`, while trusted publishing metadata recorded commit `b2d6bc16`; rebuilding from the recorded commit produced an L1-equivalent artifact. These mismatches often arose from semantic-release-style workflows that modify source metadata, commit the changes, and then publish the artifact; provenance records the triggering commit, not necessarily the final source commit used to produce the artifact.

RQ2 Takeaway

61% of analyzed divergences were caused by build tool or dependency version drift, workflow-generated intermediate files, or source modifications introduced by CI/CD steps. Improving artifact verifiability requires infrastructure that records build tool versions, source-state changes, and build-relevant workflow steps so that verifiers can accurately reconstruct the source state, environment, and build process used to produce the artifact.

C. RQ3: Artifact Characteristics and Development Choices

We consider two classes of factors that may explain verifiability outcomes: artifact characteristics, such as size, age, and popularity, and development choices, such as build backend, native code, and repository layout. Figure 5 shows that artifact characteristics have limited explanatory power: verifiability declines only slightly with artifact size, shows little relationship with popularity, and the apparent effect of release age largely disappears once attestation is controlled for. In contrast, development choices have a much stronger effect.

[1] We excluded other ecosystems after poor results on Crates.io and npm.

TABLE II
ROOT CAUSES OF ARTIFACT VERIFIABILITY FAILURES AND THE SYMPTOMS THEY PRODUCE. WE ANALYZED 134 ARTIFACTS, RESULTING IN 185 ROOT CAUSE IDENTIFICATIONS.

ID	Root Cause	Failure Symptoms	Reconstruction Gap	Count
RC01	Build tool or dependency version differences	Dependency or file version differences in metadata files; version fields of build tools and/or dependencies in manifest or lock files differ; generated files differ because build tool or dependency versions differ.	Environment reconstruction	47
RC02	CI/CD source code modification	Original artifact contains files or metadata modified by CI/CD workflow steps; package version fields, version-dependent paths, or pre-processed source files differ from the source commit.	Source-state reconstruction	38
RC03	Generated files or artifacts produced by CI/CD workflow	Generated files or intermediate build outputs such as compiled files or downloaded assets exist in the original artifact but are missing in the rebuilt artifact; files in the original artifact do not exist in the source repository; published artifact represents generated package content.	Build-process reconstruction	28
RC04	Original timestamp not fixed	Timestamp fields in archives differ.	Build-process reconstruction	18
RC05	Packaging directory differences	Files in original and rebuilt artifacts were packaged from different temporary build directories.	Build-process reconstruction	14
RC06	Source revision mismatch	There is a drift between the source commit that produced the artifact and the commit used for rebuilding.	Source-state reconstruction	11
RC07	Environment-dependent metadata in generated files or file names	Generated files contain local file paths from the build runner environment, or generated file names depend on host paths or directory names.	Environment reconstruction	6
RC08	Native artifact differences	Native binaries, wheel tags, repair metadata, records, or bundled native files differ.	Environment reconstruction	5
RC09	Package subpath differences	Files in the rebuilt artifact correspond to a different subdirectory or package in the source repository.	Source-state reconstruction	4
RC10	Git submodule materialization differences	Files from a Git submodule exist in one artifact but are missing in the other.	Source-state reconstruction	4
RC11	External VCS source inputs in artifacts	Additional files in original artifacts are VCS-tracked external source payloads, such as Git LFS project files, that were not materialized in the rebuild.	Source-state reconstruction	4
RC12	Build output contains unstable or unordered inputs	Generated tables, manifests, caches, source files, file names, directories, source maps, or embedded IDs differ across rebuilds.	Build-process reconstruction	4
RC13	Archive container encoding not fixed	File contents, paths, modes, and mtimes match, but archive SHA differs.	Build-process reconstruction	1
RC14	VCS line-ending normalization differences	Shared text files differ only by CRLF/LF normalization.	Source-state reconstruction	1

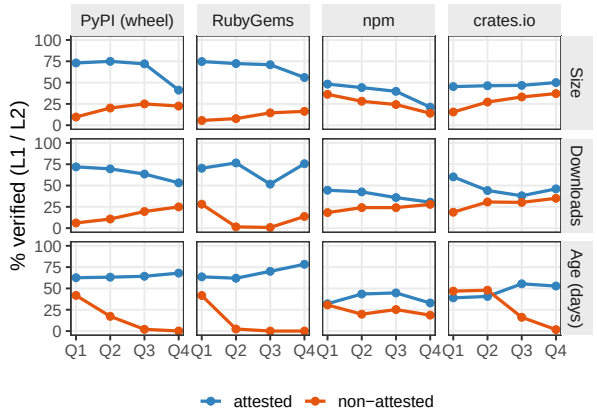

Fig. 5. Verification rate (% reproduced at L1/L2) by within-ecosystem quartile (Q1 lowest, Q4 highest) of artifact size, downloads, and age, for attested vs. non-attested packages across four ecosystems.

Across ecosystems, verifiability is primarily shaped by how deterministically the build and packaging process reconstructs the published artifact.

Packaging determinism matters more than closeness to source: Crates.io performs consistently well because crates are published as source tarballs through a standardized packaging process, with remaining mismatches mostly caused by embedded build timestamps (Section VII-B). However, being close to source is not sufficient. Package artifacts can diverge from their repositories even without compilation, because packaging can rewrite metadata, normalize files, or apply preprocessing [46]. npm illustrates this effect: `pack-only` packages verify less often than bundled or transpiled packages, even though the latter transform the source more heavily. The difference is packaging determinism: stable generated outputs can be easier to reproduce than source-like artifacts produced by nondeterministic packaging steps.

PyPI has high-verifiability backends & low-verifiability paths: PyPI shows the widest spread across build backends. At the high end, `hatchling` reproduces many wheels at L1–L2, requiring little normalization. Other backends, including `pdm-backend`, `poetry-core`, and `setuptools`, reach comparable total verification rates mainly through L3 normalization, indicating that their outputs often differ in metadata, timestamps, or ordering rather than executable content. At the low end, wheels built through legacy `setup.py bdist_wheel` paths or packages without a declared backend rarely verify. This spread helps explain why

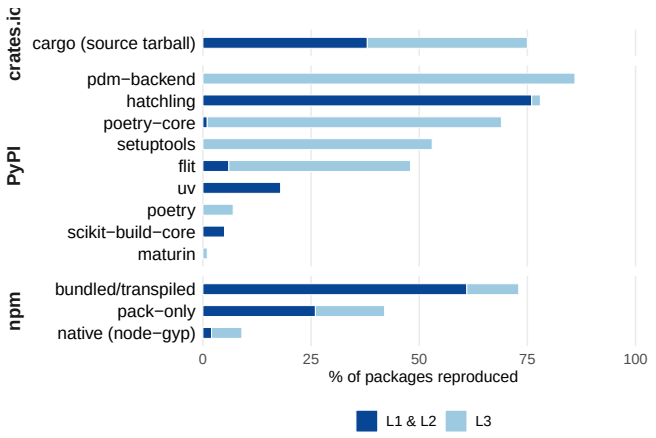

Fig. 6. Packages verified (%) by build backend or packaging strategy, grouped by ecosystem and split into L1–L2 and L3 matches.

PyPI contains some highly verifiable build configurations while remaining difficult to verify overall.

Native compilation reduces verifiability: Native compilation substantially lowers verification rates in ecosystems that ship built artifacts. On PyPI, wheels with compiled extensions, such as those built with `maturin` or `scikit-build-core`, verify far less often than pure-Python wheels; on npm, `node-gyp` modules similarly sit near the bottom of the ecosystem. These packages depend on platform-specific compilers, flags, and toolchain behavior, introducing variation that package-level normalization cannot remove [51], [52]. Native packaging therefore remains a major barrier to independent artifact verification.

Monorepos mainly affect source recovery: Repository layout has little effect once the correct package directory is recovered. PyPI packages built from monorepo subdirectories verify at the same rate as root-level packages, suggesting that monorepos primarily create a source-recovery challenge rather than a rebuild or comparison challenge. Their impact therefore appears in pipeline completion failures (Table I), not in equivalence outcomes among completed rebuilds.

RQ3 Takeaway

Verifiability is shaped more by how an artifact is built than by what the artifact is. Size, age, and popularity have limited effects, while deterministic packaging, declarative build backends, and avoidance of native compilation strongly improve verification outcomes. Monorepos mainly affect source recovery rather than rebuild equivalence, so the most important levers for improving verifiability are build- and packaging-side changes.

D. RQ4: AVP vs. OSS-Rebuild

Across the 600 artifacts, AVP verifies 331 (55%) versus 158 (26%) for OSS-Rebuild—more than twice as many. Table III provides the details. AVP attains the higher verification rate in four of the six strata, ties in one (PyPI unsupported), and is exceeded only on supported npm packages.

TABLE III
VERIFICATION BY AVP AND OSS-REBUILD ON 600 ARTIFACTS (100 PER ECOSYSTEM/STRATUM).

Eco.	Strat.	AVP-only	OSS-only	Both	None
crates	supp.	61%	2%	25%	12%
crates	un-supp.	74%	0%	0%	26%
npm	supp.	6%	13%	33%	48%
npm	un-supp.	22%	13%	21%	44%
PyPI	supp.	44%	4%	22%	30%
PyPI	un-supp.	4%	6%	19%	71%
Overall		35%	6%	20%	39%

AVP generalizes to the long tail; OSS-Rebuild does not: The starkest gap is the unsupported long tail: on unsupported crates.io, OSS-Rebuild verifies *none* (0%) against AVP's 74%, and 74%, 22%, and 6% of unsupported crates, npm, and PyPI artifacts respectively are verified by AVP alone. OSS-Rebuild depends on curated, per-package build definitions, so outside its supported set it often has no recipe to run—most visibly on crates.io, where it rebuilt no unsupported package. AVP instead recovers the published source commit and rebuilds from it, transferring to arbitrary packages without curation.

AVP is competitive even where OSS-Rebuild is tuned: Even on OSS-Rebuild's supported strata, AVP verifies more crates.io (86% vs. 27%) and PyPI (66% vs. 26%) artifacts; OSS-Rebuild leads only on supported npm (46% vs. 39%), where its hand-maintained definitions reproduce bundled and transpiled JavaScript that AVP's generic `npm pack` does not. The platforms are complementary, not redundant (Table III): on supported crates.io they agree on only 25% of artifacts while AVP uniquely verifies a further 61%, and OSS-Rebuild uniquely verifies only a small share anywhere (0–13%). A non-trivial 12–71% is verified by neither—native compilation, undeclared build backends, and unrecoverable source.

RQ4 Takeaway

AVP verifies about twice as many artifacts as OSS-Rebuild overall and is the only platform that scales beyond a curated package set. It trades some per-package fidelity on heavily transformed artifacts (e.g., supported npm) for broad coverage that needs no per-package build definitions, making generality the decisive advantage for assessing artifact authenticity at scale.

VIII. THREATS TO VALIDITY

Construct and internal validity: Our verifiability rates are lower bounds. We count an artifact as unverified when AVP cannot recover its rebuild metadata or reproduce it with canonical build commands. Thus, the rates reflect both ecosystem verifiability and AVP's verifier model, and environment reconstruction; a more complete verifier may verify more artifacts.

Our L3 results depend on a manually curated catalog of non-executable differences, built from ecosystem documentation and analysis of non-L1 cases. This catalog may be too restrictive or permissive, so we interpret L3 verification as evidence of source-to-artifact correspondence, not as proof that all normalized differences are security-neutral.

Our RQ2 root-cause analysis may also misattribute failures. We used an automated agent to scale manual coding and accepted hypothesized causes when corrective changes improved equivalence. Such changes may identify contributing causes rather than the only or true cause. Moreover, the taxonomy is derived from provenance-bearing artifacts, whose failures may differ from non-attested artifacts.

External validity and reliability: Our study evaluates only the most recent artifact of each package (section VI-A2), so it does not capture version-to-version variation or temporal drift in tools, dependencies, and attestation adoption.

We restrict rebuilds to packages with recoverable public source repositories, excluding 14.7-42.4% [53], [54]. Reported rates are therefore conditional on source recovery and may overestimate ecosystem-wide verifiability. Because we oversample attested packages, ecosystem rates are stratum-weighted rather than population-weighted, and we may miss packages with attestations outside registry-provided pathways.

Finally, our OSS-Rebuild comparison reflects one tool version and our operation of it, which may miss capabilities requiring additional configuration or expertise.

IX. Discussion

Open-source software supply-chain standards recommend reproducible builds as a means of improving trust in distributed software artifacts. We measured the gap between these recommendations and artifact verifiability in practice.

A. Verifiability Requires Artifact-Bound Rebuild Metadata

Section VII-A shows that end-to-end verification (over all sampled artifacts) ranges between 25% and 80% across ecosystems. Section VII-B further shows that many failures stem from unpinned build tools, CI/CD source modifications, and custom packaging. Independent verifiability therefore requires more than deterministic builds: it needs durable, structured, artifact-bound or recoverable metadata that lets independent parties rebuild and compare the published artifact.

Centralized build ecosystems such as Debian, Arch Linux, and Nix partly address this problem by recording rebuild metadata through ecosystem-controlled build infrastructure and providing verifiers that consume them. This model is difficult to apply directly to decentralized-build ecosystems where maintainers use diverse build tools, packaging conventions, dependency managers, and CI/CD workflows. A more practical direction is for other centralized infrastructure, such as CI/CD workflow runners, to record rebuild metadata while executing workflow steps and embed in artifacts before publishing. Our replay pilot (Section VII-B) reinforces this: the one artifact recovered by re-running its workflow matched only because the workflow pinned the build-toolchain version our heuristic missed—it is the build environment, not the workflow logic, that must be recorded. Additionally, developer practices that improve verifiability, such as pinning dependencies and including package preparation and build steps in package manifest or other metadata files, should be encouraged.

This also bears on supply-chain standards. Existing standards recommend reproducible builds [36], but our results show reproducibility becomes actionable only when ecosystems also make recovering rebuild metadata feasible; standards should therefore add concrete guidance on which metadata to preserve, where verifiers obtain it, and how results are represented, informed by our RQ2 causes (section VII-B).

B. Deploying Verifiability in Practice

This paper reframes reproducible builds as a necessary but incomplete condition for software supply-chain integrity in decentralized-build ecosystems. The practical barrier is whether an independent verifier can recover enough rebuild metadata to reproduce the published artifact. By identifying the metadata gaps and packaging practices that prevent verification, this study gives registries, standards bodies, and rebuild-tool developers concrete targets for making artifact verification practical at package-registry scale.

One challenge is how verifiability can be integrated into package lifecycles without immediate, ecosystem-wide enforcement. Registries are a natural deployment point—they already receive, validate, and distribute artifacts—and could adopt it incrementally: first publishing verification metadata and warnings, then prioritizing stronger equivalence levels for popular, critical, or security-sensitive packages. Downstream package managers and organizational policies can require a target equivalence level (section IV-B) before installation to protect users.

Provenance attestations offer another path: they improve metadata recovery but attest only to the build process, not the artifact's authenticity [55]. They could instead distinguish *workflow* provenance from *package-build* provenance providing sufficient evidence to reproduce the artifact [56].

X. Conclusion

We studied whether artifacts in decentralized-build package ecosystems can be independently verified against their source, formalizing a registry-only verifier model and tiered artifact-comparison levels and implementing both in AVP. Across 34,005 packages from Crates.io, npm, PyPI, and RubyGems, verifiability remains low and uneven: the barrier is not only build non-determinism but the difficulty of recovering the exact source revision, environment, and build process behind each artifact, and provenance attestations aid source recovery without yielding complete rebuild specifications. Compared with OSS-Rebuild, AVP verifies about twice as many artifacts and covers a broader set of popular packages. Overall, artifact verifiability at registry scale requires durable, artifact-bound rebuild metadata.

References

[1] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9403390/

[2] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar, "It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1527–1544. [Online]. Available: https://ieeexplore.ieee.org/document/10179320/

[3] G. Benedetti, O. Solarin, C. Miller, G. Tystahl, W. Enck, C. Kästner, A. Kapravelos, A. Merlo, and L. Verderame, "An empirical study on reproducible packaging in open-source ecosystems," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1052–1063. [Online]. Available: https://ieeexplore.ieee.org/document/11029905/

[4] Why reproducible builds? — reproducible-builds.org. [Online]. Available: https://reproducible-builds.org/docs/why/

[5] Open source software supply chain best practices at the eclipse foundation. [Online]. Available: https://github.com/eclipse-cbi/best-practices/blob/main/software-supply-chain/osssc-best-practices.md

[6] M. Moore, "Software supply chain security best practices v2," CNCF TAG Security, Nov. 2024, accessed: 2026-06-26. [Online]. Available: https://tag-security.cncf.io/blog/software-supply-chain-security-best-practices-v2/

[7] (2022) CISA, NSA, and ODNI release part one of guidance on securing the software supply chain | CISA. [Online]. Available: https://www.cisa.gov/news-events/alerts/2022/09/02/cisa-nsa-and-odni-release-part-one-guidance-securing-software-supply-chain

[8] O. S. B. SIG. Open source project security baseline. [Online]. Available: https://baseline.openssf.org/versions/2025-02-25.html

[9] "Malicious node-ipc versions published to npm," https://snyk.io/blog/malicious-node-ipc-versions-published-npm/, 2026, accessed 2026-05-20.

[10] "Mini shai-hulud hits antv: 300+ malicious npm packages published via compromised maintainer account," https://snyk.io/blog/mini-shai-hulud-antv-npm-supply-chain-attack/, 2026, accessed 2026-05-20.

[11] "Microsoft's durabletask pypi package compromised in supply chain attack," https://www.stepsecurity.io/blog/microsofts-durabletask-pypi-package-compromised-in-supply-chain-attack, 2026, accessed 2026-05-20.

[12] "Tanstack npm packages compromised," https://snyk.io/blog/tanstack-npm-packages-compromised/, 2026, accessed 2026-05-20.

[13] J. Xiong, Y. Shi, B. Chen, F. R. Cogo, and Z. M. J. Jiang, "Towards build verifiability for java-based systems," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. Association for Computing Machinery, 2022, pp. 297–306. [Online]. Available: https://dl.acm.org/doi/10.1145/3510457.3513050

[14] X. de Carné de Carnavalet and M. Mannan, "Challenges and implications of verifiable builds for security-critical open-source software," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. Association for Computing Machinery, 2014, pp. 16–25. [Online]. Available: https://dl.acm.org/doi/10.1145/2664243.2664288

[15] J. Malka, S. Zacchiroli, and T. Zimmermann, "Does functional package management enable reproducible builds at scale? yes." in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, 2025, pp. 775–787, ISSN: 2574-3864. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11025777

[16] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, "Investigating the reproducibility of npm packages," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 677–681. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00071

[17] K. Gao, W. Xu, W. Yang, and M. Zhou, "PyRadar: Towards automatically retrieving and validating source code repository information for PyPI packages," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 115:2608–115:2631, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3660822

[18] Google, "OSS Rebuild," https://github.com/google/oss-rebuild, 2025, accessed: 2026-06-08.

[19] B. Hassanshahi, T. N. Mai, B. S. Smith, and N. Allen, "Unlocking reproducibility: Automating re-build process for open-source software," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 3392–3402, ISSN: 2643-1572. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11334351

[20] C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "SoK: Analysis of software supply chain security by establishing secure design properties," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED '22. Association for Computing Machinery, 2022, pp. 15–24.

[21] Sonatype, "2026 state of the software supply chain," Sonatype, 2026, accessed: 2026-08-11. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[22] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[23] Debian – the universal operating system. [Online]. Available: https://www.debian.org/

[24] npm | home. [Online]. Available: https://www.npmjs.com/

[25] PyPI · the python package index. [Online]. Available: https://pypi.org/

[26] Trusted publishing for npm packages | npm docs. [Online]. Available: https://docs.npmjs.com/trusted-publishers

[27] crates.io Docs, "Trusted publishing," https://crates.io/docs/trusted-publishing, 2026, accessed: 2026-06-08.

[28] Supply chain threats. [Online]. Available: https://slsa.dev/spec/v1.0/threats-overview

[29] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Springer International Publishing, 2020, pp. 23–43. [Online]. Available: https://doi.org/10.1007/978-3-030-52683-2_2

[30] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[31] Snyk, "Poisoned security scanner: Backdooring litellm," https://snyk.io/articles/poisoned-security-scanner-backdooring-litellm/, 2026, accessed: 2026-03-25.

[32] Account takeover and malicious replacement of ctx project — python security 0.0 documentation. [Online]. Available: https://python-security.readthedocs.io/pypi-vuln/index-2022-05-24-ctx-domain-takeover.html

[33] "Post mortem: axios npm supply chain compromise," https://github.com/axios/axios/issues/10636, 2026, accessed 2026-05-20.

[34] K. Pandya, "Malicious dYdX packages published to npm and PyPI after maintainer compromise," Socket, Feb. 2026, accessed: 2026-06-26. [Online]. Available: https://socket.dev/blog/malicious-dydx-packages-published-to-npm-and-pypi

[35] Reproducible Builds, "Reproducible builds website." [Online]. Available: https://reproducible-builds.org/

[36] Securing build pipelines. Section: community. [Online]. Available: https://tag-security.cncf.io/community/publications/supply-chain-security-tools/securing-build-pipelines/

[37] Variations in the build environment — reproducible-builds.org. [Online]. Available: https://reproducible-builds.org/docs/env-variations

[38] Debian Reproducible Builds Team, "Debian variance testing dashboard," 2026, accessed: 2026-06-30. [Online]. Available: https://tests.reproducible-builds.org/debian/reproducible.html

[39] J. Dietrich, T. White, B. Hassanshahi, and P. Krishnan, "Levels of binary equivalence for the comparison of binaries from alternative builds," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 576–587, ISSN: 2576-3148. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11186060

[40] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch, "Aroma: Automatic reproduction of maven artifacts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 836–858, 2024. [Online]. Available: https://doi.org/10.1145/3643764

[41] Provenance. [Online]. Available: https://slsa.dev/spec/v1.0/provenance

[42] (2025) Introducing OSS rebuild: Open source, rebuilt to last. [Online]. Available: https://blog.google/security/introducing-oss-rebuild-open-source/
[43] npm Docs, "Generating provenance statements," https://docs.npmjs.com/generating-provenance-statements/, 2026, accessed: 2026-06-08.
[44] W. Woodruff, F. Tuesca, and D. Ingram, "Pep 740: Index support for digital attestations," https://peps.python.org/pep-0740/, 2024, accessed: 2026-03-25.
[45] The Cargo Book, "cargo-package(1)," https://doc.rust-lang.org/cargo/commands/cargo-package.html, 2026, accessed: 2026-06-08.
[46] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Lastpymile: Identifying the discrepancy between sources and packages," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '21. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468264.3468592
[47] Reproducible Builds Project, "SOURCE_DATE_EPOCH — reproducible builds," https://reproducible-builds.org/docs/source-date-epoch/, 2023, accessed: 2026-03-28.
[48] Reproducible Builds. diffoscope: in-depth comparison of files, archives, and directories. Software tool developed as part of the Reproducible Builds project. Accessed: 2026-06-20. [Online]. Available: https://diffoscope.org/
[49] N. Imtiaz and L. A. Williams, "Are your dependencies code reviewed?: Measuring code review coverage in dependency updates," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4932–4945, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3319509
[50] Nektos Authors, "act: Run your github actions locally," Jul. 2026. [Online]. Available: https://github.com/nektos/act
[51] J. Malka, S. Zacchiroli, and T. Zimmermann, "Reproducibility of build environments through space and time," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. Association for Computing Machinery, 2024, pp. 97–101. [Online]. Available: https://dl.acm.org/doi/10.1145/3639476.3639767
[52] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, "Options matter: Documenting and fixing non-reproducible builds in highly-configurable systems," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. Association for Computing Machinery, 2024, pp. 654–664. [Online]. Available: https://dl.acm.org/doi/10.1145/3643991.3644913
[53] A. Tsakpinis and A. Pretschner, "Analyzing the accessibility of github repositories for pypi and npm libraries," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 345–350. [Online]. Available: https://doi.org/10.1145/3661167.3661231
[54] A. Tsakpinis, N. Raube, and A. Pretschner, "Investigating notable metadata practices in pypi libraries: An empirical study about repository and donation platform urls," 2026. [Online]. Available: https://arxiv.org/abs/2601.15139
[55] OpenSSF. Introducing artifact attestations—now in public beta – open source security foundation. [Online]. Available: https://openssf.org/blog/2024/05/24/introducing-artifact-attestations-now-in-public-beta/
[56] Build: Verifying artifacts. [Online]. Available: https://slsa.dev/spec/v1.2/verifying-artifacts