Are Your Dependencies Code Reviewed?:
Measuring Code Review Coverage in
Dependency Updates

Nasif Imtiaz © and Laurie Williams ©, Fellow, IEEE

Abstract—As modern software extensively uses free open source packages as dependencies, developers have to regularly pull in new third-party code through frequent updates. However, without a proper review of every incoming change, vulnerable and malicious code can sneak into the codebase through these dependencies. The goal of this study is to aid developers in securely accepting dependency updates by measuring if the code changes in an update have passed through a code review process. We implement Depdive, an update audit tool for packages in Crates.io, npm, PyPI, and RubyGems registry. Depdive first (i) identifies the files and the code changes in an update that cannot be traced back to the package’s source repository, i.e., phantom artifacts; and then (ii) measures what portion of changes in the update, excluding the phantom artifacts, has passed through a code review process, i.e., code review coverage. Using Depdive, we present an empirical study across the latest ten updates of the most downloaded 1000 packages in each of the four registries. We further evaluated our results through a maintainer agreement survey. We find that phantom artifacts are not uncommon in the updates (20.1% of the analyzed updates had at least one phantom file). The phantoms can appear either due to legitimate reasons, such as in the case of programmatically generated files, or from accidental inclusion, such as in the case of files that are ignored in the repository. Regarding code review coverage (CRC), we find the updates are typically only partially code-reviewed (52.5% of the time). Further, only 9.0% of the packages had all their updates in our data set fully code-reviewed, indicating that even the most used packages can introduce non-reviewed code in the software supply chain. We also observe that updates either tend to have high CRC or low CRC, suggesting that packages at the opposite end of the spectrum may require a separate set of treatments.

Index Terms—Software supply chain security, open source security, dependency analysis.

I. INTRODUCTION

MODERN software extensively uses free open source packages as dependencies [1]. However, using open source has opened up new attack vectors, as vulnerable and even malicious code can sneak into software through these third-party dependencies [2]. Further, practitioners are recommended to keep dependencies up to date with the latest version [3], resulting in developers pulling in new code through frequent updates [4], often automatically and without a security review [5].

Recent times have seen popular open source packages be compromised and push malicious updates, such as the attack through ua-parser-js, an npm package used by major software corporations [6]. Attacks through dependencies, such as the preceding example, are categorized as “supply chain attacks” by security practitioners [7]. While the existence of malicious packages in different ecosystems is well-known [2], popular packages from reliable sources can also be compromised, such as through: 1) hijacking of a maintainer’s account [8]; 2) a maintainer going rogue [9]; 3) account handover through social engineering [10]; and 4) build system compromise [11].

Therefore, developers are now recommended to review dependency updates before merging them into the codebase [12], [13], as the responsibility of security lies on the consumer when using free open-source code [14]. However, manually reviewing each update’s code changes may not be a practical solution, as projects may have hundreds of direct and transitive dependencies [1], [15]. Further, actively maintained packages get frequent updates, over-burdening any project that would employ such strict measures. Therefore, we propose that dependency updates go through automated security and quality checks before being merged, which can act as the first line of defense and aid developers in securely accepting dependency updates.

Recently, the software industry has proposed frameworks to define the compliance standards for using open source packages, such as the “Supply chain Levels for Software Artifacts (SLSA)” [16] framework. Specifically, SLSA provides a checklist of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure. Among others, SLSA v0.1 requires a dependency package to employ a two-person code review [17]. However, while projects like the “Security Scorecards” [18] exist to aid in dependency selection, little research has been done yet on employing automated checks during each subsequent update [19]. The goal of this study is to aid developers in securely accepting dependency updates by measuring if the code changes in an update have passed through a code review process.
The check for code review ensures that each line of code change in the update can be traced back to at least two owners.

This check can guard against the threat scenarios where 1) only one maintainer’s account has been hijacked; 2) a single maintainer has gone rogue; or 3) attackers have compromised the publishing infrastructure to inject unwanted malicious code. Further, research has shown that code review helps improve code quality [20] and can prevent the introduction of new security vulnerabilities [21]. Ladisa et al. [22] have developed an attack taxonomy for open source supply chain attacks, where they have mentioned code review as a safeguard against the attack vector inject into sources of legitimate package.

However, the feasibility of employing an automated check for the code review requirement during dependency updates has not been studied yet. Packages get bundled and distributed in various ways and may use different code review tooling. Such differences in the maintenance and distribution of packages create challenges in reliably auditing the updates. Further, no empirical study exists on the code review practices among top packages to understand the practicality of this SLSA v0.1 requirement. This paper aims to address these gaps in the research of securely using open source dependencies.

We implement Depdive which measures the code review coverage (CRC) [23] in a dependency update. We define CRC as the proportion of the code changes in an update that has gone through a code review process. Depdive works for four package registries, namely Crates.io for Rust, npm for JavaScript, PyPI for Python, and RubyGems for Ruby. We choose these registries because they follow a similar package distribution model, where the maintainers upload the package code to the registry. We further scope our implementation to GitHub repositories as we leverage the platform to determine if certain code changes have been reviewed or not.

In summary, Depdive maps code changes between two versions of a package uploaded in the registry to the corresponding commits in the package’s source repository. It then identifies if there was a reviewer for the mapped commits through four GitHub-based checks. Along the process, Depdive also identifies the files and lines of code that cannot be mapped from the registry to the repository, which we refer to as phantom artifacts, following the definition in prior work [19]. While one approach can be to consider the phantom artifacts as non-reviewed in the denominator of the CRC measurement, these artifacts may exist in the form of binaries or programmatically-generated files, as will be shown in this paper. Therefore, they will require a provenance tracking mechanism to audit if changes in them were reviewed or not. With Depdive, we first filter out the phantom artifacts and output them separately from the CRC measurement.

For an empirical evaluation, we run Depdive over the latest ten releases of the most downloaded 1000 packages in Crates.io, npm, PyPI, and RubyGems. Based on Depdive’s output, we answer the following two research questions:

- **RQ1:** To what extent do phantom artifacts exist in the updates of the most downloaded packages?
- **RQ2:** Excluding phantom artifacts, what is the code review coverage (CRC) in the updates of the most downloaded packages?

Besides the answer to the above two research questions, the contributions of our work include a working tool, Depdive, that outputs details on the phantom artifacts and the code review data for a dependency update. We also present a survey of package maintainers’ agreement with Depdive’s outputs. The code, survey questionnaire, and data for this paper are anonymously available at https://tinyurl.com/depdive.

The rest of the paper is structured as follows: Section II explains key concepts to this study, Section III explains Depdive workflow. Section IV explains our dataset, while Section V presents our findings. Section VI presents the maintainer agreement survey to our analysis. Section VII lists the limitation of our study, and Section VIII discusses the implication of our findings. Section IX discusses related work before Section X concluding this paper.

## II. BACKGROUND

In this section, we explain the concepts key to our study:

- **Package:** A package, also referred to as a library or module, is a reusable software application unit that can be used by other software.

- **Dependency:** When a software uses a package, the package is referred to as a dependency. Moreover, the dependencies of the dependency package itself also become transitive dependencies of the client software. The client software depends on a specific dependency version and can update to a newer version when available.

- **Package Source Code Repository:** Repository is a cloud file hosting service with a versioning system to store the source code of a software project. In a git repository, the most granular unit to track a revision of the source code is called a commit that is identified through a unique commit hash.

- **Package Registry:** Package registries are centralized package hosting services to store and distribute packages. In this paper, we work with Crates.io, npm, PyPI, and RubyGems registry. These registries follow a similar package distribution model, where developers can upload their package source code to the registry alongside required artifacts, such as data files and pre-compiled binaries. The client software can then download and install packages from these registries. While it is expected that the same source code in the repository is distributed via the registry, the registry contains its own copy of the package code, which may not be identical to the one in the repository.

- **Code Review:** Code review is a manual review process of code changes by any developer(s) other than the author. While the history of code review is not tracked by git, reviews are generally performed using a tool, e.g., Gerrit [24]. GitHub offers a pull-based development model that integrates native code review tooling. A developer can open a pull request (PR) on GitHub to submit code changes and ask for reviews from other developers.

- **Emerging Industry Standards:** Recently, multiple industry standards have emerged that provide a checklist of controls and...
requirements for the safe use of open source packages. Notable examples are Supply Chain Levels for Software Artifacts (SLSA)\(^1\) and NIST Secure Software Development Framework\(^2\). SLSA identifies a number of threats in the software of supply chain and lists a set of requirements, such as build and provenance requirements, for which the client project developers can audit their dependencies to minimize risks.

III. DEPDIVE

In this section, we describe the implementation of the update audit tool, Depdive. Fig. 1 shows a high-level workflow of the tool.

Depdive takes four arguments as input: 1) registry name; 2) package name; 3) current version; and 4) update version. Depdive works with four package registries, namely Crates.io, npm, PyPI, and RubyGems. The input can be an update from any version to another version available in the registry. We scope Depdive’s implementation to GitHub repositories, as we rely on data available through the GitHub platform to determine if a commit was code-reviewed, as explained in Section III-E.

Depdive collects package code both from the registry and the repository and compares them to identify the phantom artifacts. Phantom artifacts in the registry do not map to any corresponding artifact in the repository, and therefore, cannot be audited for code review without a provenance tracking mechanism. Hence, Depdive outputs phantom artifacts separately from the code review analysis. Algorithm 1 lists pseudocode for determining phantom files and lines in an update.

A. Collect Package Code

In this subsection, we explain how Depdive collects package code from both the registry and the repository.

1) Download Package Code From the Registry: Depdive downloads package code for both the current and the update version from the respective registries. While Crates.io, npm, and RubyGems provide every package in a uniform format, PyPI packages can be available in multiple formats in the registry. When multiple formats are available, we prefer the \textit{wheel} distribution as it is the default distribution format.

2) Locate Package Repository: We determine a package’s repository from the metadata provided by the registry. Afterward, we locate the directory path of the package within the repository. A repository can contain source code for multiple packages. Therefore, we need to know the directory path of a package for an accurate one-to-one mapping of files between the registry and the repository. For example, the filepath \texttt{CHANGELOG.md} in the Rust package \texttt{tokio} maps to the filepath \texttt{tokio/CHANGELOG.md} in the repository.

For Crates.io, npm, and RubyGems, Depdive identifies the directory path by locating the manifest file (\texttt{Cargo.toml}, \texttt{package.json}, and \texttt{Gemspec}) of the package in the repository. While PyPI packages do not contain a uniform manifest file, we locate the directory by matching the filepaths in the registry with the filepaths in the repository. Through directory path locating, Depdive also validates the retrieved repository for a package. Further, a repository can contain submodules that point to different repositories. We obtain commit history recursively for all the submodules in the repository.

3) Map File Path From the Registry to the Repository: For each file in the registry, we obtain the repository filepath by locating the manifest file (\texttt{Cargo.toml}, \texttt{package.json}, and \texttt{Gemspec}) of the package in the repository. While PyPI packages do not contain a uniform manifest file, we locate the directory by matching the filepaths in the registry with the filepaths in the repository. Through directory path locating, Depdive also validates the retrieved repository for a package. Further, a repository can contain submodules that point to different repositories. We obtain commit history recursively for all the submodules in the repository.

1\(^{\text{https://slsa.dev/}}\)

2\(^{\text{https://csrc.nist.gov/Projects/ssdf}}\)
Step 1 Identify Phantom artifacts in an update

Require: registry name: $E$
Require: package name: $P$
Require: current version: $X$
Require: update version: $Y$

1: $P_X = \text{DownloadRegistryCode}(E, P, X)$
2: $P_Y = \text{DownloadRegistryCode}(E, P, Y)$
3: $F_{YP} = \text{GetPackageFilepaths}(P_Y)$
4: $R, D = \text{LocateRepositoryAndDirectory}(E, P)$
5: $C_X = \text{IdentifyReleaseCommit}(R, X)$
6: $C_Y = \text{IdentifyReleaseCommit}(R, Y)$
7: $F_{YR} = \text{GetPackageFilepathsAtCommit}(R, D, C_Y)$
8: Set of phantom files, $H_{PF} = \emptyset$
9: for all $f \in F_{YP}$ do
10: \hspace{1em} $f_r = \text{GetRepositoryFilePath}(f, D)$
11: \hspace{1em} if $f_r \notin F_{YR}$ then
12: \hspace{2em} $H_{PF} = H_{PF} \cup f$
13: end if
14: end for
15: Map of phantom lines to files: $M_{PL}$
16: for all $f \in F_{YP} - H_{PF}$ do
17: \hspace{1em} $f_r = \text{GetRepositoryFilePath}(f, D)$
18: \hspace{1em} Code changes in the registry,
19: \hspace{1em} $D_P = \text{Diff}(f, P_X, P_Y)$
20: \hspace{1em} Code changes in the repository,
21: \hspace{1em} $D_R = \text{Diff}(f, R, C_X, C_Y)$
22: \hspace{1em} phantom lines, $P_f = D_P - D_R$
23: if $P_f \neq \emptyset$ then
24: \hspace{2em} $M_{PL}.insert(f, P_f)$
25: end if
26: end for
27: return Phantom files and phantom lines, $H_{PF}, M_{PL}$

Further, we also resolve the cases where a filepath contains a symbolic reference to another file within the repository.

B. Identify Release Commit

To compare the package code in the registry and the repository, we need to identify the head commit from which a certain version was built and uploaded in the registry. To refer to this commit as the release commit.

Depdive identifies the release commit through the git tags in the corresponding repository. Tagging the release commit with the version number is a recommended developmental practice [25] and was followed in prior research work [26], [27]. Depdive identifies the release tag and the associated commit for a version through a regular expression match. If the repository does not contain a single tag to match the given version, Depdive fails to analyze the corresponding update.

While not every repository annotates the release commit via git tags or may do it inaccurately, the alternative is to compare the repository code with the registry code at all the commits in the history and take the commit with the smallest difference.

Step 2 Map code delta to corresponding commits in the repository

Require: Release commit of current version: $C_X$
Require: Release commit of update version: $C_Y$
Require: Files changed in update: $F_{XY}$

1: Commits between $C_X$ and $C_Y$
2: $L_{XY} = \text{git_log}(C_X .. C_Y)$
3: Common ancestor, $C_A = \text{ParentOf}(\text{Oldest}(L_{XY}))$
4: Map of added lines to commit: $A_{XY}$
5: Map of removed lines to commit: $R_{XY}$
6: for all $f \in F_{XY}$ do
7: \hspace{1em} Git blame: $B^X_f = \text{git_blame}(f, C_Y)$
8: for all $(l, c) \in B^X_f$ do
9: \hspace{2em} if $c \in L_{XY}$ then
10: \hspace{3em} $A_{XY}.insert(f, (l, c))$
11: \hspace{2em} end if
12: end for
13: Reverse git blame:
14: $RB^X_f = \text{git_blame}(f, C_A, reverse = True)$
15: for all $(l, c) \in RB^X_f$ do
16: \hspace{2em} if $c \neq C_Y$ then
17: \hspace{3em} $rc = \text{findRemovalCommit}(f, c, l)$
18: \hspace{3em} if $rc \in L_{XY}$ then
19: \hspace{4em} $R_{XY}.insert((f, (l, rc))$
20: \hspace{3em} end if
21: \hspace{2em} end if
22: end for
23: end for
24: return Code changes to commit map, $A_{XY}, R_{XY}$

Registry code may contain phantom artifacts which will not be present in the release commit, and therefore, may not return an exact match with any single commit. Vu et al. developed LastPyMile [19] to identify phantom artifacts in a PyPI package by comparing the registry code with every commit in the repository. But their approach cannot identify the specific release commit for a given version. Therefore, we take the heuristic to identify release commits via repository tags.

C. Identify Phantom Artifacts

Depdive compares the diff between the package code of the current and update version in the registry, and the diff between the two versions’ corresponding release commits in the repository4. However, code changes can be present in the registry, but not in the repository. We refer to such changes as phantom artifacts [19]. Depdive outputs phantom artifacts in two categories, phantom files, and phantom lines:

1) Phantom File: We define phantom files in an update as files that are present in the update version in the registry, but not present in the corresponding repository filepath at the release commit (line 8–14 in Algorithm 1).

2) Phantom Line: We define phantom lines in an update as the lines of code changes that are present in the diff

3https://github.com/nasifmitiazhohi/version_differ/blob/84187f77aceb01149b05ce2f0b668d12b0b7ed6c7/version_differversion_differ.py#L87

4We calculate diff through git-diff (https://git-scm.com/docs/git-diff)
between the current and the update version, but not present in the diff between the corresponding release commits (line 15–24 in Algorithm 2).

D. Map Code Delta to Commits

Besides phantom files and lines, the rest of the diff between two versions in the registry maps to the diff between the corresponding release commits in the repository. However, not every line of change in the diff can be mapped to a specific commit within the two release commits. To explain why, consider the two commit history graphs shown in Fig. 2, where $C_X$ and $C_Y$ refer to the release commit for version X and Y, respectively. In Fig. 2(a), both $C_X$ and $C_Y$ lie on the same development branch, where $C_X$ is a direct ancestor of $C_Y$. In this case, each line of change in the diff between $C_X$ and $C_Y$ can be mapped to a single commit between those two commits. Contrarily, in Fig. 2(b), $C_X$ and $C_Y$ lie on two different branches with a common ancestor at commit $C_A$. In this case, a line of code that was added in $C_X$, but was not present in $C_A$ and subsequently in $C_Y$, will be shown as a deleted line in the diff between $C_X$ and $C_Y$ (and between the two versions fetched from the registry). However, that line of code was not deleted in any commit, rather just did not appear between $C_A$ and $C_Y$.

Therefore, we define code delta in an update as code changes that can be mapped to a commit that is present in the update version, but not present in the current version. Following this definition, we only consider the commits between $C_A$ and $C_Y$ in Fig. 2(b) as the commits between $C_X$ and $C_Y$ (double-dot commit range as per git syntax). The rationale behind this approach is that when analyzing an update, Depdive assumes that the current version is already trusted by the user, and therefore, focuses analysis only on the code changes in the new update. Consequently, in Fig. 2(b), we assume that if $C_X$ is trusted by a user, any ancestor commit of $C_X$ is also trusted by the user. Therefore, when updating from $C_X$ to $C_Y$, we only focus on the code changes between their common ancestor $C_A$ and the update version $C_Y$. In the simpler case, where both $C_X$ and $C_Y$ lie on the same branch as shown in Fig. 2(a), $C_X$ itself is the common ancestor of the two versions.

Algorithm 2 shows Depdive’s implementation of identifying the code delta between two versions and mapping each line of code in the delta to a corresponding commit. To identify the newly-added lines in the update version, Depdive runs git blame on each package file at commit $C_Y$ to obtain the commits that added each line in the file. If a line was added in a commit that lies between $C_A$ (same as $C_X$ in Fig. 2(a)) and $C_Y$, we take the line as a newly-added line in the update (line 6–10 in Algorithm 2). To identify the lines that have been removed in the update, we run reverse git blame on each package file at commit $C_A$ to obtain the last commit where each line in the file was present. If a line was still present in $C_A$, the line was not removed in the update version. For the rest of the removed lines, we identify the removal commit by traversing forward from the blamed commit up to $C_Y$. If the removal commit lies between $C_A$ (same as $C_X$ in Fig. 2(a)) and $C_Y$, we take the line as a removed line in the update (line 12–20 in Algorithm 2).

E. Determine If a Commit Was Code-Reviewed

After obtaining the commits involved in an update, we need to determine if the commit was code-reviewed. Git does not store data on code-review for a commit. Therefore, we focus on the projects that use GitHub as their development platform and look for evidence on GitHub to determine if a commit had been code-reviewed. Specifically, we apply four checks. Note that, we adopt these checks from “Security Scorecards” [18], a tool that analyzes the last 30 commits of a GitHub project and generates a score for a project’s adherence to code review. Below, we explain the four checks:

1) GitHub review: We check if a commit belongs to a pull request and if the pull request was reviewed through GitHub’s native code review tooling.

2) Different merger: We check if a commit belongs to a pull request and if the pull request was opened and merged by two different GitHub accounts.

3) Different committer: We check if a commit was authored and committed by two different accounts.

4) Third-party tools: Besides GitHub’s native code review tooling, a project can use external code review tools, e.g., Gerrit [24], or custom bots to handle code-review, e.g., Prow [28]. When a commit is reviewed on Gerrit, the commit message contains metadata for the Gerrit review. When a pull request on GitHub is handled by Prow, Prow

\[^{5}\]In these cases, the client project is updating to a version in a different release branch, and, therefore, is deciding to revert some changes in the current branch that may not be present in the update branch. The decision to revert such changes is on the client, and code review coverage is not relevant in these cases.
adds a label to the pull request to indicate if it was code-reviewed. Following Security Scorecards’ implementation, we also check for evidence for Gerrit or Prow review in Depdive.

We determine a commit as code-reviewed if any one of the four checks is met. When checking for different mergers or committers, we do not consider a commit to be code-reviewed if both the author and reviewer are bot accounts on GitHub. We further exclude cases where GitHub’s own bots are involved (e.g., GitHub actions). However, in cases where a pull request has been opened by a bot, e.g., Dependabot [29], and merged by a non-bot account, we consider the commit as code-reviewed.

### IV. Dataset

In this section, we explain the packages and their updates we selected for this study, and the updates that Depdive could successfully analyze based on which we complete our empirical analysis.

#### A. Package Selection

We select the latest ten updates of the most downloaded 1000 packages in each of the four package registries, namely crates.io, npm, PyPI, and RubyGems. For crates.io and RubyGems, we download the official data dumps that are updated daily and then select the top packages in order of the total download count across all versions. For PyPI, we use the dataset from [30] that is updated on a monthly basis. We collected the data for these three registries at the end of December 2021. For NPM, we use the dataset from [31] that was constructed in August 2021. While the download count can be inflated in different ways, including through automated CI/CD tooling, sampling the most downloaded packages to study a package ecosystem is an established approach in the literature [19], [32] and provides an estimation of the most used packages in a registry.

We then collect the list of the available version releases on the registry for each of the selected packages at the end of December 2021. Depdive can run on an update from any version to another (e.g., 1.8.3 → 1.8.4, 1.5.1 → 1.9.0, 1.0.0 → 3.0.3 etc.). However, for this empirical study, for each version, we consider an update only from its prior version according to SemVer [33] ordering. For example, for tokiol@1.8.4, we consider an update from tokiol@1.8.3 to tokiol@1.8.4 to pass as input to Depdive. We choose this approach to avoid any data duplication, that is, the same code changes and commits to appear in multiple updates of a package selected in this study. Further, we exclude any pre-release version from our data set.

We also restrict our data set to only the most recent ten updates for each of the packages. The rationale is that Depdive makes multiple GitHub API calls for each commit in an update for code-review checks. However, GitHub limits API calls to 5000/hr for each user, which puts a constraint on how much analysis we can do within an hour. However, choosing at least ten updates will provide us with the data to measure the consistency of a package in its code review coverage in each update. In the case where a package has less than ten updates, we consider all of them in our study.

Out of the 4000 packages, 26 packages had only one release listed on the registry. Further, 12 packages had only one regular release available, while the other available versions were pre-releases. We exclude these 38 packages as there were no updates available to analyze. Finally, we select 33,895 updates from 3,962 packages. Table I shows a breakdown of these initially selected updates across the four registries.

#### B. Depdive Analysis

Out of the selected 33,895 updates, Depdive successfully analyzed 24,944 (73.6%) updates. In Section V, we present our empirical analysis based on Depdive’s output for these 24,944 updates. Table I shows a breakdown of the dataset of this study. We collect the following metrics for each update to conduct our empirical analysis: 1) No. of phantom files, 2) No. of files with phantom lines, 3) No. of added phantom lines, and 4) Code Review Coverage (CRC).

Note that, we measure the CRC of an update as the proportion of the update’s code delta that has been code-reviewed. Here, the code delta, as explained in Section III-D, is the code changes in an update that can be mapped to a commit in the repository and therefore, can be classified if code-reviewed or not.

#### C. Why Depdive Failed

As shown in Table I, Depdive only successfully analyzed 73.6% of the initially-selected updates. The primary reason that Depdive could not analyze an update was the absence of git tags in the repository pointing to the release commit of a version, which was the case for 5,747 updates (17.0%). The next major reason is Depdive’s inability to either locate or validate the repository of a package, which was the case for 2,669 of all the updates (7.9%). For 127 updates (0.4%), the listed repository was not on GitHub. For the rest of 1.1% of the updates, Depdive failed for various reasons, including not being able to read a

| Registry   | No. of Selected Packages | No. of Selected Updates | No. of Packages analyzed by Depdive | No. of Updates analyzed by Depdive |
|------------|--------------------------|-------------------------|-------------------------------------|-----------------------------------|
| crates.io  | 990                      | 8,434                   | 833 (84.1%)                        | 6,326 (75.0%)                    |
| npm        | 989                      | 8,158                   | 919 (92.9%)                        | 7,178 (88.0%)                    |
| PyPI       | 992                      | 8,657                   | 788 (79.4%)                        | 6,089 (70.3%)                    |
| RubyGems   | 991                      | 8,646                   | 674 (68.0%)                        | 5,351 (61.9%)                    |
| Total      | 3,962                    | 33,895                  | 3,214 (81.1%)                      | 24,944 (73.6%)                   |

7While Depdive also outputs phantom removal, we observe that the removed lines are often also phantom lines from the old version.
file containing non-Unicode characters, a private submodule within the repository, and version code not being available on the registry.

If Depdive were to be deployed in a CI/CD pipeline, our empirical evaluation shows that the tool could fail in 26.4% of the cases. However, the failure rate may be reduced if the package repositories followed the best developmental practices, such as annotating a git tag for each released version. We recommend that package manager tools add a feature to include metadata on the repository, package directory, and release commit in the package bundle during the build process to aid in a third-party audit.

V. FINDINGS

Based on Depdive’s output, we answer the following two research questions for the four studied registries: **RQ1**: To what extent do phantom artifacts exist in the updates of the most downloaded packages? and **RQ2**: Excluding phantom artifacts, what is the code review coverage in the updates of the most downloaded packages? In the following two subsections, we present our findings:

A. **RQ1: Phantom Artifacts**

Table II shows the occurrence of phantom files, and Table III shows the occurrence of phantom lines in updates across the four studied registries. Overall, 24.9% of the updates contained either a phantom file or a phantom line (3.2%, 44.3%, 38.5%, and 9.1% in the case of Crates.io, npm, PyPI, and RubyGems updates, respectively). Below, we discuss our findings:

1) **Phantom Files**: We find that 20.1% of the analyzed updates and 27.4% of the analyzed packages had at least one phantom file. We find that npm and PyPI updates are more likely to contain a phantom file (34.7% and 34.3%, respectively) than Crates.io and Ruby Gems updates (1.9% and 6.0%, respectively). Overall, we identified 306,940 phantom files across all the updates. Below, we provide a crude characterization of the phantom files that we identified:

Across all the npm updates, we identified 123,000 phantom files. We observed that for some npm packages, the code in the registry could be a transpiled version of the source code in the repository. For example, the source code may be written in TypeScript, whereas the package contains transpiled JavaScript code. Similarly, npm packages can also contain minified JavaScript, or JavaScript transpiled through transcompilers such as babel. Overall, 89.4% of the phantom files in the npm updates are JavaScript files (.js, .ts, .cjs, .mjs, .min.js, .map, .flow files) which may be code transpiled during the package build process. Besides transpiled code, 9.5% of the npm phantom files are .json files which are either data or configuration files, and possibly were git-ignored in the repository through the .gitignore file.

We identified 146,564 phantom files across all the PyPI updates. The majority of these files (46.1%) are machine-generated header files for C/C++ (.h, .hpp, .inc files), while 7.4% of the phantom files are compiled binaries (.so, .jar, .dylib). The core engine of many Python packages, e.g., tensorflow, are written in languages like C/C++, while the Python files only provide an interface to communicate with the engine. While the repositories of these packages contain the source code for the engines, the Python packages, in their default wheel distribution, only contain the compiled binaries and the machine-generated header files alongside the Python source files. The header files and binaries come from 118 distinct packages in our dataset (15.0% of the analyzed PyPI packages). Moreover, we find 16.3% of the phantom files to be Python files, where one-third of them (32.1%) are __init__.py files which are presumably machine-generated. Finally, we have also observed non-Python code files (7.1% are .js, .ts files), data files (5.0% are .dat files), and git-ignored files (e.g., .pyc, .pyi, .py files) as phantom files.

Across all the RubyGems updates, we found 7,483 phantom files. However, 83.2% of these files came from 7 packages where the package either (i) bundled its own Ruby dependency packages, (ii) bundled non-Ruby dependency code, or (iii) included log files that were git-ignored in the repository.

**TABLE II**

| Registry   | Total packages | Total Updates | Packages with phantom files | Updates with phantom files | Median phantom file count |
|------------|----------------|---------------|-----------------------------|---------------------------|--------------------------|
| Crates.io  | 833            | 6,326         | 42 (5.0%)                   | 123 (1.9%)                | 3.25                     |
| npm        | 919            | 7,178         | 413 (44.9%)                 | 2,489 (34.7%)             | 2.00                     |
| PyPI       | 788            | 6,089         | 355 (45.1%)                 | 2,088 (34.3%)             | 1.50                     |
| RubyGems   | 674            | 5,351         | 71 (10.5%)                  | 323 (6.0%)                | 2.00                     |
| **Total**  | 3,214          | 24,944        | 881 (27.4%)                 | 5023 (20.1%)              | 2.00                     |

**TABLE III**

| Registry   | Total packages | Total Updates | Packages with phantom lines | Updates with phantom lines | Median file count with phantom lines | Median count of added phantom lines |
|------------|----------------|---------------|-----------------------------|---------------------------|-------------------------------------|-----------------------------------|
| Crates.io  | 833            | 6,326         | 63 (7.6%)                   | 87 (1.4%)                 | 1.0                                 | 2.0                               |
| npm        | 919            | 7,178         | 265 (28.8%)                 | 1,733 (24.1%)             | 1.0                                 | 1.0                               |
| PyPI       | 788            | 6,089         | 157 (19.9%)                 | 471 (7.7%)                | 1.0                                 | 3.0                               |
| RubyGems   | 674            | 5,351         | 105 (15.6%)                 | 185 (3.5%)                | 1.0                                 | 2.0                               |
| **Total**  | 3,214          | 24,944        | 590 (18.4%)                 | 2,476 (9.9%)              | 1.0                                 | 2.0                               |
Across Crates.io updates, we identified 29,893 phantom files. However, 84.4% of these files come from a single package that contained the source code for its project website hosted in a repository branch. Overall, across all four registries, we have observed files that are git-ignored in the repository to be included in the registry, e.g., `.DS_Store`, `.npmignore`, `.editorconfig`, `setup.cfg` files.

**Accidental vs legitimate phantom files:** Broadly, we can characterize the phantom files into two categories: (i) phantom files due to legitimate reasons such as compiled binaries in PyPl wheel distribution, transpiled JavaScript, auto-generated files, and third-party dependency code; and (ii) phantom files presumably put in mistakenly, such as the git-ignored files.

For the legitimate phantom files, we need to track their provenance, and then determine if changes in the origin files have been code-reviewed or not. We discuss current research efforts and our recommendations on handling legitimate phantom files in Section VIII. On the contrary, we recommend package maintainers issue a new clean release in case of accidental phantom files, as the last mile between the repository and the registry has been used as an attack vector in the past to sneak in malicious code [2], [19].

Overall, while there may be legitimate reasons behind phantom files, we cannot audit the changes in these files through our proposed approach. Package users may manually review these files before accepting an update, especially when the current version does not contain any phantom files, but the new update does. In our dataset, 461 packages (52.1% of all the packages with phantom files) had phantom files only in a subset of all their updates. We presume that these packages may have accidentally put in phantom files in some of their updates.

2) Phantom Lines: We find that 9.9% of the analyzed updates across 18.4% of the packages had code changes that could not be mapped back to a commit in the repository. We find that phantom lines are generally small changes, with 2 added lines in 1 file at the median. We also find that npm updates are more likely to contain phantom lines than updates in the other three studied registries.

We identified phantom lines in 1,733 (24.1%) npm updates, where 93.9% of these updates contained phantom lines in the manifest file, `package.json`, that was presumably dynamically generated with added data such as the release commit. Further, as explained when discussing phantom files, the npm package code in the registry can be transpiled from the code in the repository. Therefore, the same filepath may have different code in the registry and the repository, resulting in many updates with phantom lines.

Across PyPI packages, we identified 471 updates (7.7%) with phantom lines, where 51.6% of the updates had phantom lines in a file named `_version.py` that was presumably dynamically generated with added data such as the release commit, the build date, and the version number. Further, `_init__.py` files can also be generated dynamically during the build process and may differ in content from the repository copy, which resulted in phantom lines in 117 updates (24.8%). For both Crates.io and RubyGems, we do not find any common pattern in the identified phantom lines. However, false positives may appear in the case where a release commit was inaccurately tagged.

While some files can be dynamically generated during the build process, we find that phantom lines are less likely to occur due to legitimate reasons in Crates.io, PyPl, and RubyGems updates, and package users should manually review these lines before accepting an update.

We find that phantom artifacts are not uncommon in package updates, specifically in the case of npm and PyPI packages (20.1% of the updates having at least one phantom file). While further research is required to verify if the changes in phantom artifacts were code reviewed or not, we recommend practitioners remain careful of accidental phantom artifacts, such as publishing local files in the package that are not checked into the repository.

**B. RQ2: Code Review Coverage**

In this section, we present our findings on code review coverage (CRC) for the analyzed updates. Note that, we exclude phantom artifacts in CRC measurement, and only consider the changes in the package code that can be mapped to the repository. Table IV presents the median lines of code (LOC) changes and the median CRC across updates in the four registries. We excluded 334 updates from the 24,944 analyzed updates as they contained zero code delta as measured by Depdive. These updates either only made changes in the non-package files in the repository, or only made changes in the phantom files. Further, we identified 110,657 commits as code-reviewed, of which 60.1% were GitHub review, 31.5% were Different merger, and 8.4% were Different committer, as per the checks explained in Section III-E.

The table also shows the portion of the updates with 100% and 0% CRC across the four registries. We find that 11.0% of the updates were fully code-reviewed, while 36.5% of the

| Registry   | Total Package | Total Updates | Median LOC changes in updates | Median code review coverage (CRC) in updates | Updates with 100% CRC | Updates with 0% CRC |
|------------|---------------|---------------|-------------------------------|---------------------------------------------|----------------------|---------------------|
| Crates.io  | 830           | 6,293         | 75.75                         | 42.9%                                       | 915 (14.5%)          | 2,068 (32.9%)       |
| npm        | 918           | 7,147         | 25.00                         | 5.9%                                        | 87 (1.2%)            | 3,497 (48.9%)       |
| PyPI       | 780           | 5,861         | 76.75                         | 52.7%                                       | 1,425 (24.3%)        | 1,760 (30.0%)       |
| RubyGems   | 672           | 5,309         | 61.25                         | 31.0%                                       | 269 (5.1%)           | 1,663 (31.3%)       |
| **Total**  | **5,200**     | **24,610**    | **31.00**                     | **27.2%**                                   | **2,096 (11.9%)**    | **8,988 (36.5%)**   |

TABLE IV

**CODE REVIEW COVERAGE OF THE ANALYZED UPDATES**

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
updates were not code-reviewed at all. The rest of the 52.5% of the updates were only partially code-reviewed. Further, we find that the median CRC across all the analyzed updates stands at 27.2%.

We find the npm packages to have the lowest median CRC. In Section V-A, we explained that npm packages may contain transpiled JavaScript, resulting in many phantom artifacts. In these cases, Depdive’s CRC measurement would be limited to only a subset of the package files. For example, for the no-case package, Depdive could only audit the package manifest files for code review. To address this limitation, we looked at the CRC for the 3,947 npm updates that did not contain any phantom artifacts with a presumption that such updates are less likely to contain transpiled code. However, we still observe a low median CRC of 6.1% for these updates. While many npm packages are small and may only have a single maintainer, our analysis in this study may be under-approximation for npm. The packages that contain transpiled JavaScript, e.g., packages with source code in TypeScript, may also be more likely to follow developmental best practices such as code review. We have observed npm packages from reputed organizations like babel and facebook to develop their source code in TypeScript and adhere to code review in the majority of the commits. However, Depdive could only audit a subset of the files in these packages, and therefore, we may have got an under-approximated CRC measurement.

For crates.io, PyPI, and RubyGems packages, we find the median CRC to be 42.9%, 52.7%, and 31.0%, respectively. Fig. 3 shows a violin chart of the CRC across the updates. We observe an hourglass shape in the violin chart, suggesting that updates tend to have either very high CRC or very low CRC. Open source packages are often maintained by a small group of maintainers. While the contributions from an outsider get code-reviewed by the maintainers, the code changes from the maintainers themselves may remain non-reviewed, which can be a possible explanation behind low-to-medium CRC. On the contrary, we have found updates that can have very high CRC yet not fully code-reviewed. For example, we have found 2,442 updates (9.9%) that are larger than 51 LOC changes (overall median) and have greater than 90% but less than 100% CRC.

While most of the commits in these updates were code-reviewed, we found some commits did not go through the code review process, possibly due to the following two primary reasons as per our observation: (i) Non-critical changes: the commit only changed configuration or documentation files; (ii) Cherry-picked commits: the commit cherry-picked a commit from a different branch where it was code-reviewed either to backport a fix or to import changes to a release branch from the master branch. Note that, SLSA v0.1 requires context-specific approval during a code review, which means the cherry-picked commits require their own separate review. Nonetheless, this phenomenon shows that non-reviewed code changes may sneak in even in packages that attempt to adhere to code review. A few notable packages where we found high CRC but not 100% CRC include numpy, ansible-core in PyPI, nokogiri in Ruby, openssl in crates.io, etc.

Table V shows packages for whom all the analyzed updates in our data set were measured to have 100% CRC or all the updates were measured to have 0% CRC. We find that only 9.0% of the packages in our dataset had all their updates fully code-reviewed, most of them coming from PyPI and crates.io. We observed that packages from reputed organizations such as google, Azure, rust-lang, tokio-rs, and aws are likely to consistently have fully code-reviewed updates. On the contrary, 29.7% of the packages had none of their updates code-reviewed at all. These packages are typically maintained by a small group of maintainers who do not review each other’s code.

Table VI shows the median CRC across updates of different sizes and release types. The bottom and third quartiles of median LOC changes across all analyzed updates lie at 12 and 203. Based on this, we divide the updates into three categories, small (< 12 LOC changes), medium (12 to 203 LOC changes), and large (≥ 203 LOC changes). Further, we parse the update version according to SemVer [33] formatting as either major, minor, or patch updates. If any update version does not correspond to SemVer formatting, we ignore that in this analysis.

We find that small updates with less than 12 LOC have a very low median CRC (4.8%). A possible explanation is that minor changes that result in a new update may be made by maintainers themselves and do not go through a code review process. Further, 43.4% of these small updates come from npm packages, which as an ecosystem, has a low median CRC. Apart from small updates, the median CRC for medium and large-sized updates, and major, minor, and patch versions range from 20.0% to 48.2%.

Fig. 3. Violin plot of code review coverage across updates.

| Registry       | Total Packages | Packages with all updates having 100% CRC | Packages with all updates having 0% CRC |
|----------------|----------------|------------------------------------------|----------------------------------------|
| crates.io      | 830            | 99 (11.9%)                               | 217 (26.1%)                            |
| npm            | 918            | 5 (0.5%)                                 | 386 (42.0%)                            |
| PyPI           | 780            | 162 (20.8%)                              | 185 (23.7%)                            |
| RubyGems       | 672            | 23 (3.4%)                                | 161 (24.0%)                            |
| Total          | 3,200          | 289 (9.0%)                               | 949 (29.7%)                            |
We find that 52.5% of the analyzed updates were only partially code-reviewed, with an overall median code review coverage (CRC) of 27.2%. Further, only 9.3% of the analyzed packages were measured for 100% CRC across all their updates. We observe that updates tend to have either low CRC where maintainers only code-review outsider contributors’ pull requests, or have high CRC with only commits that modify configuration or documentation files not going through a code-review process.

| Update type                  | Total Package | Total Updates | Median LOC changes in updates | Median code review coverage (CRC) in updates |
|------------------------------|---------------|--------------|-------------------------------|---------------------------------------------|
| Major updates                | 1,016         | 1,808        | 153.0                         | 20.7%                                       |
| Minor updates                | 2,386         | 7,223        | 108.0                         | 46.8%                                       |
| Patch updates                | 2,979         | 15,005       | 28.0                          | 20.0%                                       |
| Updates with less than 12 LOC change | 2318         | 6,460        | 6.5                           | 4.8%                                        |
| Updates with 12-203 LOC changes | 2,968       | 12,195       | 50.0                          | 36.9%                                       |
| Updates with more than 203 LOC change | 2069         | 5,955        | 536.0                         | 48.2%                                       |

VI. MAINTAINER AGREEMENT SURVEY

We conducted a survey to triangulate the results from Depdive with the package maintainers. Specifically, we evaluated whether the studied packages’ maintainers agreed with the Depdive analysis. We emailed maintainers of the packages and provided an analysis report of a randomly-selected update (of that package) and asked two Likert scale questions based on the analysis: 1) Do you agree with our analysis? 2) How often do you require code review?

We also gave options for the respondents to explain their answers.

We sent out the survey through email to the maintainers of 945 packages. We did not survey packages where: 1) we could not collect the maintainers’ valid email addresses; 2) we already emailed the maintainers querying on a different package they also own; and 3) the package had phantom artifacts resulting in incomplete analysis on our part. We received 96 responses, with a response rate of 10.2%.

Fig. 4 shows a pie chart of maintainer agreement to our analysis. 47 (49.0%) respondents fully agreed to our analysis, while 33 (34.4%) respondents partially agreed. The remaining 16 (16.7%) respondents disagreed. In 48 cases (16 disagreeing and 32 somewhat agreeing), the maintainers provided the reasoning behind their disagreement. We classified the reasoning into the following categories:

1) **Non-functional changes (15):** Maintainers noted that the unreviewed commits did not change the source code of the package, rather only configuration files, e.g. package manifest files, CI/CD scripts, and documentation files, e.g. README.md, CHANGELOG.md. The maintainers disagreed with our analysis, noting that changes in these files should not be included in the CRC analysis.

2) **Single maintainer (8):** Maintainers disagreed on the rationale that CRC analysis is invalid for their packages, as they are the sole maintainer.

3) **Review outside GitHub (7):** In 7 cases, maintainers noted that the commits flagged as unreviewed were indeed reviewed, either through an internal code review tool, or through discussion on GitHub.

4) **Misunderstanding of the analysis (7):** There were 7 cases where maintainers disagreed for reasons that appear to be a miscommunication on our part regarding what the tool should do. For example, maintainers noted that Depdive missed some commits in the surveyed update. When investigating those updates, we found that the omitted commits were on non-package files, e.g., test files or CI/CD scripts, which were not present in the registry, and therefore, were not included in the CRC analysis.

5) **Trivial changes (5):** Maintainers noted that the unreviewed commits only consisted of trivial changes, e.g., code re-formatting, and therefore, does not require a code review. They viewed flagging such commits as unreviewed as false alarms.

6) **Others (6):** While we marked dependabot PRs merged by a developer as reviewed, two maintainers opined that they should be marked as unreviewed. In one case, the maintainer disagreed that the backported commits needed to be reviewed again. In another case, the unreviewed commit was only a merge commit. In one case, the package only included auto-generated content. In another case, the package is now unmaintained.

Finally, Fig. 5 shows maintainers’ responses to how often they require code review for new contributions in their package.
We observe that the responses are in alignment with our empirical findings in Table V. Only 12.5% of the packages always require code review, while 41.7% of the packages often require code review. On the other hand, 28.2% of the packages either never or rarely require a code review.

![Percentage of packages requiring code review](image)

**Fig. 5.** Maintainers’ responses to code review requirement in their packages.

We found that maintainers largely agree with our analysis. While there are cases where the code review took place outside GitHub (and therefore, flagged as unreviewed by Depdive), the maintainers’ disagreement primarily stemmed from their opinion that some changes (e.g., non-functional changes) should not be included in the CRC analysis.

**VII. LIMITATIONS**

In this section, we discuss the limitations of our study.

A. Reliability Threat

We adopt multiple heuristics in designing Depdive that may result in false-positive outputs. Firstly, we map package files in the registry to a file in the repository through filepath matching. However, maintainers may choose a build process where the filepaths will be altered in the bundled package without altering the content of the file, in which case Depdive will output them as phantom files. While an alternate approach could be a pairwise comparison of all the files, our simpler heuristic is based on a realistic assumption that bundled packages typically follow the same directory structure as that in the repository. Secondly, we leverage repository tags to determine the release commit for a package version, inaccuracy in which may incorrectly output phantom artifacts. We explained this design decision in Section III-B and recommend packages themselves contain the metadata for the release commit.

Further, Depdive scopes its code-review checking to the GitHub platform, and therefore, may output false negatives in the case where a project is hosted on the GitHub platform but perform its code review elsewhere. Indeed, seven survey respondents (7.9%) noted that the code marked as unreviewed was actually reviewed in a way not covered by our checks. Overall, we chose our heuristics to keep the design simple while minimizing the possibility of false negatives. In this regard, our empirical analysis of the phantom artifacts may represent an over-approximation, while the analysis of code review coverage may represent an under-approximation.

We make an assumption in this paper that all code reviews, as identified by Depdive, are reliable. In reality, the review quality may differ. Further, an attacker can open a fake account on GitHub to stamp review approvals on malicious code pushed by them. However, our focus in this study is to automatically measure code review coverage, while the quality/trustworthiness of the reviews is in the scope of future work.

Further, Depdive’s design relies on the assumption that the file structure of the code in the repository and the registry would match. We manually investigated 25 random updates from each of the four ecosystems to check if our assumptions hold and if the identified release commit by Depdive is accurate. For 99 cases, we found our assumptions to be valid. In 1 npm update, the code in the registry was a transpiled version of the code in the repository, and therefore, the directory structure did not match (code lived in the `./src` directory in the repository and in the `./dist` directory in the registry). The transpiled code, however, is different from the code in the repository and, therefore, was correctly identified as phantom files by Depdive.

B. Generalizability Threat

Our work studies the most downloaded packages in four package registries. However, as shown in Table IV, we failed to analyze 26.4% of the initially-selected updates. The failure may introduce unknown sampling biases in our dataset for the empirical analysis. Further, the threat to the CRC analysis for the npm ecosystem due to the presence of transpiled code has been explained in Section V-B. Moreover, we study only the most downloaded 1000 packages in each ecosystem. Therefore, our empirical findings may not generalize to all the packages in the ecosystem. The selection criteria help us avoid introducing biases from rarely used packages. Overall, we believe our empirical findings provide an evaluation of our proposed approach and offer many insights into the existence of phantom artifacts and CRC among the most downloaded packages.

C. Limitations in Auditing Phantom Artifacts

In its current design, Depdive cannot link the phantom artifacts to its origin source files and audit if changes in those source files have been reviewed or not. This is a limitation of the current design. We discuss the possible workflows developers can adopt to review the phantoms picked up by Depdive and suggest possible future work to solve this limitation in Section VIII.

D. What Should (Not) Be Included in CRC?

In our survey, maintainers have opined that not all files should be included in the CRC analysis. For example, multiple maintainers have opined that changes in the package manifest file, e.g., `package.json` for npm, need not be reviewed. However, in the past, supply chain attacks were carried out by introducing a malicious package in the dependency chain listed in the manifest file9.

9https://security.snyk.io/vuln/SNYK-JS-NODEIPC-2426370
Similarly, changes in configuration scripts and test files were identified as non-code, non-functional, bookkeeping changes by the survey respondents. Maintainers may follow the practice of not uploading these files in the registry if the files are not part of the package source code. Alternatively, one survey respondent suggested providing a feature for the maintainers to define an allowlist that should be excluded from the CRC analysis.

In our study, we have taken a conservative approach of including every change in the package code in the CRC analysis. While not all changes may pose a security risk, future research is required to formally verify which files/changes are risk-free in each ecosystem, and can be safely ignored in any update audit.

VIII. Discussion

In this section, we discuss various aspects of Depdive’s design and the implication of the findings from our empirical study:

A. Depdive Design Philosophy

One driving philosophy behind Depdive is that every line of code in the software supply chain should be traceable to at least two owners [16]. To achieve that, we need to map the package code downloaded from the registry to the commits in the repository, and then reliably determine the author and reviewers of the mapped commits. However, we face two challenges in the process:

Provenance of the legitimate phantom artifacts: We find that legitimate phantom artifacts, such as the programmatically generated files, can appear in a package. Especially, transpiled JavaScript code in the npm packages and compiled binaries in the PyPI packages make Depdive’s audit incomplete. The proposed design in this paper cannot audit the changes in the origin files from where these programmatically generated artifacts have been compiled.

Nevertheless, the following approaches may help in identifying legitimate phantom artifacts and safeguarding against potentially malicious phantoms: (1) Client project developers can manually review phantom artifacts which we find to be small in size in average cases, as shown in Tables II and III. (2) We can add rule-based checks in Depdive that are specific to each ecosystem to identify legitimate phantoms.

Depdive can build the package from the source and check if the same phantoms arise. However, this may not be easy in practice as builds may not be reproducible due to many factors such as timestamps, build tool versions, etc. [34]. (4) Client projects can retrieve the package code directly from the source repository and then build the package by themselves. [10] Clients then can run Depdive on the source distribution for a complete CRC measurement.

Another approach can be to divide the update audit into two parts: (i) have a framework, such as in-toto [35] and Reproducible Builds [34], to verify the package has indeed been build from a commit in the repository, and (ii) measure CRC directly from the repository (without the step in Algorithm 1).

Further, we find that only 20.1% of the updates contain phantom files, and these updates contain only two phantom files and two lines at a median. Therefore, it would not be impractical for the client project developers to manually review the phantom artifacts identified by Depdive before accepting an update.

In this section, we discuss various aspects of Depdive’s design and the implication of the findings from our empirical study:

Difference with other tools: There are existing tools that either (i) measure the code review adherence of a project; or (ii) identify phantom artifacts in a package. “Security Scorecard” [18] gives a score for a GitHub project’s code review adherence by looking at the branch protection rules and the review history of the last 30 commits. While a periodic check can help in selecting a dependency, we aim to measure the code review coverage during each update, which presents an added challenge of mapping the code changes from the registry to the repository commits. Further, Vu et al. [19] have developed LastPyMile to identify phantom artifacts in a PyPI package by comparing the package code with every commit in the repository.
While Depdive’s approach has methodological differences with LastPyMile, as explained in Section III-B, both the tools should return similar results (for phantom artifacts in PyPI updates). In a way, Depdive brings the above two tools’ objectives into a single workflow and provides an isolated audit only for the changes in an update. Such an audit will help package users focus their review effort on the incremental changes in each update.

B. The State of the Package Ecosystems

Through our empirical evaluation of Depdive, we also present our findings on the code review coverage in the recent updates of the most downloaded packages. Below, we discuss some implications of our findings:

The lack of code review among popular packages: We analyze the most downloaded packages in this paper and find that the median code review coverage stands at only 27.2% with only 9.0% of the packages having all their updates fully code-reviewed. Our work highlights the lack of code review in packages that are widely used parts of today’s software supply chain. These packages are weak links in the supply chain as one developer’s account holds immense power in posing a threat to the global software supply chain [2], [31]. In the following paragraphs, we discuss possible future work directions in addressing this issue.

The hourglass phenomenon: We have seen that packages either tend to have very high CRC or very low CRC, as depicted by an hourglass shape in Fig. 3. The packages with high CRC should enforce strict branch protection to reject any non-reviewed commits, or not include any non-production files in the package that have loose restrictions on code-review. On the other end of the spectrum, packages with low CRC may be in need of more manpower. Future research may look at recommending reviewers for packages that are highly used but maintained by a small group of maintainers.

Post-release code vetting: The trade-off that comes with code review is slowed-down development, while maintainers of some packages may not welcome reviewers in their projects. Further, multiple maintainers can collide in developing a package and wait until the package becomes sufficiently popular before pushing in a backdoor as part of a long-term cyberattack. An alternate approach to code review can be the post-release crowd-vetting of the code changes in a new update [36]. Package registries can provide a system where developers from all around the world can review and approve each new release of a package, while the client projects wait a certain period until an update has garnered enough approval before accepting it in their codebase.

IX. RELATED WORK

Supply chain security: Recent works have focused on the secure use of open source dependencies as part of the software supply chain [2], [15], [31], [36]. Duan et al. have proposed static and dynamic analysis approaches to detect malicious packages for the interpreted languages [37], while Sejfić et al. have proposed machine learning models to detect malicious npm packages [38]. Further, Ferreira et al. have proposed a permission-based protection mechanism for malicious npm updates [39]. Our work differs from these prior works in the way that we do not explicitly aim to detect malicious updates, but rather propose an automated check for each update to aid developers in reviewing the security and the quality of the incoming changes before accepting them into the codebase.

Ecosystem-wide analysis: Recent works have done ecosystem-wide analysis to understand the state of different software supply chain networks. Zimmermann et al. [36] and Liu et al. [40] have looked at the vulnerability propagation in the npm ecosystem, while Alfadel et al. [41] have investigated the PyPI ecosystem. Similarly, Zahan et al. [31] and Bommarito et al. [32] have looked at various quality issues in the npm and PyPI ecosystems, respectively. Further, Imtiaz et al. [27] have looked at how packages release security fixes with an ecosystem-wide analysis for seven package registries.

Code review: There is a rich body of literature establishing the benefits of code review in software development [23], [42], [43]. Research has shown that code review can help both software quality and security [20], [21], while also helping disseminate the knowledge of the codebase [44].

X. CONCLUSION

We implement Depdive, a dependency update audit tool for Crates.io, npm, PyPI, and RubyGems packages, that (i) identifies the files and the code changes in an update that cannot be traced back to the package’s source repository, i.e., phantom artifacts; and then (ii) measures what portion of changes in the update excluding the phantom artifacts has passed through a code review process, i.e., code review coverage. Depdive can help package users in focusing their review effort on the phantom artifacts and the non-reviewed code when pulling in a new update, while also providing a quality estimate of the incoming changes. We ran Depdive over the latest ten updates of the most downloaded 1000 packages in each of the four above-mentioned registries, of which Depdive could successfully analyze 73.6% of the updates.

Overall, from our empirical evaluation of 24,944 updates across 3,214 packages, we present interesting insights regarding the studied package ecosystems. We find that phantom artifacts are not uncommon in the updates, either due to legitimate reasons, such as in the case of programmatically-generated files, or from presumably accidental inclusion, such as in the case of git-ignored files. While phantom artifacts are rare in Crates.io and RubyGems updates, we find that npm and PyPI updates can commonly have phantom artifacts in the form of compiled JavaScript code, compiled binaries, and other machine-generated files.

Regarding code review coverage, we find that updates are typically only partially code-reviewed (52.5% of the time). Further, only 9.0% of the packages in our dataset had all their updates fully code-reviewed, highlighting the fact that even the most used packages ship unreviewed code. We also observe that updates tend to have either very high CRC or very low CRC, indicating that packages at the opposite end of the spectrum...
require different treatments. We further evaluated our empirical findings through a maintainers agreement survey. Overall, this paper provides an empirical evaluation of our proposed approach to auditing a dependency update and an ecosystem-level analysis of code review coverage among the latest updates of the most downloaded packages.

REFERENCES

[1] Synopsys, “2021 open source security and risk analysis report,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyzer-reports/open-source-security-risk-analysis.html

[2] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in Proc. Int. Conf. Detect. Intrusions Malware Vulnerability Assess. Cham, Switzerland: Springer, 2020, pp. 23–43.

[3] “9 reasons for keeping software dependencies up to date,” 2019. Accessed: Jul. 1, 2023. [Online]. Available: https://nullbeans.com/9-reasons-for-keeping-software-dependencies-up-to-date/

[4] “Push and pull: When and why to update your dependencies,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://pythonspeed.com/articles/when-update-dependencies/

[5] S. Mirhosseini and C. Parmin, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?,” in Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE). Piscataway, NJ, USA: IEEE, 2017, pp. 84–94.

[6] R. Flowers, “Supply chain attack: NPM library used by Facebook and others was compromised,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://hackaday.com/2021/10/22/supply-chain-attack-npm-library-used-by-facebook-and-others-was-compromised/

[7] W. E. Deming, “For good measure,” USENIX PATRONS, 2020, p. 83.

[8] “Security issue: Compromised NPM packages of UA-parser-JS,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://github.com/ossf/scorecard

[9] A. Sharma, “Dev corrupts NPM libs ‘colors’ and ‘faker’ breaking thousands of apps,” 2022. Accessed: Jul. 1, 2023. [Online]. Available: https://arstechnica.com/information-technology/2021/09/travis-ci-flaw-exposed-secrets-for-thousands-of-open-source-projects/

[10] “Compromised NPM package: Event-stream,” 2018. Accessed: Jul. 1, 2023. [Online]. Available: https://medium.com/intrinsic-blog/compromised-npm-package-event-stream-d47d806d502

[11] “Travis CI flaw exposed secrets of thousands of open source projects,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://arstechnica.com/information-technology/2021/09/travis-ci-flaw-exposed-secrets-for-thousands-of-open-source-projects/

[12] J. Yang, Y. Lee, and A. P. McDonald, “Solarwinds software supply chain security: Better protection with enforced policies and technologies,” in Proc. Int. Conf. Softw. Eng., Artif. Intell., Newv. Parallel/Distrib. Comput. Cham, Switzerland: Springer, 2021, pp. 43–58.

[13] “Shifting supply chain security left with dependency review,” 2020. Accessed: Jul. 1, 2023. [Online]. Available: https://blog.github.com/2020-12-17-shifting-supply-chain-security-left-with-dependency-review/

[14] J. Washburn, “What is FOSS, and why should I be worried about it?,” 2017. Accessed: Jul. 1, 2023. [Online]. Available: https://www.stoelprivacyblog.com/2017/10/articles/software/what-is-fooss-and-why-should-i-be-worried-about-it/

[15] N. Imtiaz, S. Thorn, and L. Williams, “A comparative study of vulnerability reporting by software composition analysis tools,” in Proc. 15th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM), 2021, pp. 1–11.

[16] “Safeguarding artifact integrity across any software supply chain,” 2021. [Online]. Available: https://slsa.dev/

[17] “SLSA requirements,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://slsa.dev/spec/v0.1/requirements

[18] “Security scorecards,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://github.com/ossf/scorecard

[19] D.-L. Yu, F. Massacci, J. Hui, I. Pashchenko, H. Plate, and A. Sabetta, “Last-PyMile: Identifying the discrepancy between sources and packages,” in Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2021, pp. 780–792.

[20] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “An empirical study of the impact of modern code review practices on software quality,” Empirical Softw. Eng., vol. 21, no. 5, pp. 2146–2189, 2016.

[21] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the characteristics of vulnerable code changes: An empirical study,” in Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng., 2014, pp. 257–268.

[22] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “Taxonomy of attacks on open-source software supply chains,” 2022, arXiv:2204.04008.

[23] N. Imtiaz, A. Khanom, and L. Williams, “Open or sneaky? Fast or slow? Light or heavy?: Investigating security releases of open source packages,” 2021, arXiv:2112.06804.

[24] “Prow,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://github.com/kubernetes/test-infra/tree/master/prow

[25] “Dependabot,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://github.com/dependabot

[26] H. van Kemenade and R. Si, “hugovk/top-pypi-packages: Release 2022.01.01,” Jan. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5812615

[27] N. Zahan, L. Williams, T. Zimmermann, P. Godefroid, B. Murphy, and C. Maddilla, “What are weak links in the npm supply chain?,” 2021, arXiv:2112.10165.

[28] E. Bommarito and M. Bommarito, “An empirical analysis of the Python Package Index (PyPI),” 2019, arXiv:1907.1073.

[29] “Semantic versioning 2.0.0,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://semver.org/

[30] “Reproducible builds,” 2021. Accessed: Jul. 1, 2023. [Online]. Available: https://reproducible-builds.org/

[31] S. Torres-Arias, H. Afzali, T. K. Kuppussyamy, R. Curtmola, and J. Cappos, “In-toto: Providing farm-to-table guarantees for bits and bytes,” in Proc. 28th USENIX Secur. Symp. (USENIX Security), 2019, pp. 1393–1410.

[32] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, “Small world with high risks: A study of security threats in the npm ecosystem,” in Proc. 28thUSENIX Secur. Symp. (USENIX Security), 2019, pp. 995–1010.

[33] R. Daan, O. Altrawi, R. P. Kasturi, R. Elder, B. Saltfaffagamaggio, and W. Lee, “Towards measuring supply chain attacks on package managers for interpreted languages,” 2020, arXiv:2002.01139.

[34] A. Sejila and M. Schäfer, “Practical automated detection of malicious NPM packages,” 2022, arXiv:2202.13953.

[35] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, “Containing malicious package updates in npm with a lightweight permission system,” in Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE). Piscataway, NJ, USA: IEEE, 2021, pp. 1334–1346.

[36] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, “Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem,” 2022, arXiv:2201.03981.

[37] M. Afzal, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in Python packages,” in Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER). Piscataway, NJ, USA: IEEE, 2021, pp. 446–457.

[38] W. J. Gawinski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: A case study at Google,” in Proc. 40th Int. Conf. Softw. Eng.: Softw. Eng. Pract., 2018, pp. 181–190.

[39] O. Kononenko, O. Baysal, L. Guerrero, Y. Cao, and M. W. Godfrey, “Investigating code review quality: Do people and participation matter?,” in Proc. IEEE Int. Conf. Software Maintenance Evol. (ICSME). Piscataway, NJ, USA: IEEE, 2015, pp. 111–120.

[40] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in Proc. 35th Int. Conf. Softw. Eng. (ICSE). Piscataway, NJ, USA: IEEE, 2013, pp. 712–721.