PyRadar: Towards Automatically Retrieving and Validating Source Code Repository Information for PyPI Packages

KAI GAO, Peking University, China
WEIWEI XU, Peking University, China
WENHAO YANG, Peking University, China
MINGHUI ZHOU*, Peking University, China

A package's source code repository records the package's development history, which is critical for the use and risk monitoring of the package. However, a package release often misses its source code repository due to the separation of the package's development platform from its distribution platform. To establish the link, existing tools retrieve the release's repository information from its metadata, which suffers from two limitations: the metadata may not contain or contain wrong information. Our analysis shows that existing tools can only retrieve repository information for up to 70.5% of PyPI releases. To address the limitations, this paper proposes PyRadar, a novel framework that utilizes the metadata and source distribution to retrieve and validate the repository information for PyPI releases. We start with an empirical study to compare four existing tools on 4,227,425 PyPI releases and analyze phantom files (files appearing in the release's distribution but not in the release's repository) in 14,375 correct and 2,064 incorrect package-repository links. Based on the findings, we design PyRadar with three components, i.e., Metadata-based Retriever, Source Code Repository Validator, and Source Code-based Retriever, that progressively retrieves correct source code repository information for PyPI releases. In particular, the Metadata-based Retriever combines best practices of existing tools and successfully retrieves repository information from the metadata for 72.1% of PyPI releases. The Source Code Repository Validator applies common machine learning algorithms on six crafted features and achieves an AUC of up to 0.995. The Source Code-based Retriever queries World of Code with the SHA-1 hashes of all Python files in the release's source distribution and retrieves repository information for 90.2% of packages in our dataset with an accuracy of 0.970. Both practitioners and researchers can employ the PyRadar to better use PyPI packages.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Maintaining software**; • **Human-centered computing → Open source software**.

Additional Key Words and Phrases: PyPI Ecosystem, Python Package, Source Code Repository, Software Provenance, Software Supply Chain

ACM Reference Format:
Kai Gao, Weiwei Xu, Wenhao Yang, and Minghui Zhou. 2024. PyRadar: Towards Automatically Retrieving and Validating Source Code Repository Information for PyPI Packages. *Proc. ACM Softw. Eng.* 1, FSE, Article 115 (July 2024), 24 pages. https://doi.org/10.1145/3660822

1 INTRODUCTION

In order to boost productivity and reduce cost, developers widely reuse "the wheels" [He et al. 2021; Larios Vargas et al. 2020; Nguyen et al. 2020], i.e., reusing existing third-party packages rather

*Corresponding Author

Authors' Contact Information: Kai Gao, Peking University, Beijing, China, gaokai19@pku.edu.cn; Weiwei Xu, Peking University, Beijing, China, xuww@stu.pku.edu.cn; Wenhao Yang, Peking University, Beijing, China, yangwh@stu.pku.edu.cn; Minghui Zhou, Peking University, Beijing, China, zhmh@pku.edu.cn.

ACM 2994-970X/2024/7-ART115
https://doi.org/10.1145/3660822

than implementing the functionality from scratch. Many programming language (PL) communities provide centralized package registries (such as PyPI for Python and NPM for JavaScript) to facilitate the sharing and reuse of third-party packages, which host a substantially growing number of packages. As of September 2023, over 480 thousand Python packages have been published in PyPI.

Despite the benefits, reusing third-party packages also poses unique challenges to software development. First, given so many packages available, selecting the right one is laborious for developers [Larios Vargas et al. 2020]. Second, even if the right package is selected, how to estimate, monitor, and mitigate risks of the package, e.g., stop-of-maintenance [Valiev et al. 2018] and vulnerabilities [Alfadel et al. 2021; Decan et al. 2018; Liu et al. 2022; Pan et al. 2022; Pashchenko et al. 2018; Zimmermann et al. 2019], is important yet challenging. Merely using source code in the package's distribution is often insufficient in mitigating the above challenges. Practitioners and researchers commonly turn to the package's source code repository. On the one hand, development activity data recorded in the repository can help identify and mitigate some risks in the package. For example, the number of stars is a critical factor when selecting third-party packages [Larios Vargas et al. 2020]; the number of commits, contributors, and issues are popular indicators of the package's maintenance state [Valiev et al. 2018]; researchers also mine undisclosed vulnerabilities from issues [Pan et al. 2022] and track patches for known vulnerabilities [Xu et al. 2022]. On the other hand, the package's source code repository provides package users a place that the package registry lacks, allowing them to report bugs [Panichella et al. 2021], make contributions [Tsay et al. 2014; Zhou and Mockus 2012; Zhu et al. 2016], and seek community help [Dabbish et al. 2012; Zhou and Mockus 2015]. Therefore, a package's source code repository not only complements the role of its distribution in estimating and mitigating risks of the package but also plays a crucial role in its usage. Locating the source code repository is vital for every package.

However, many major PL communities (Python, JavaScript, Java, etc.) follow the common practice of separating the code repository and distribution artifact of the package. The separation offers numerous benefits, e.g. reducing the package's installation size [Ladisa et al. 2023] and streamlining the build process [Vu et al. 2021; Wang et al. 2020], but disconnects the artifact from its source code repository. The good news is, that developers usually use build tools to manage and publish distribution artifacts, and most build tools provide mechanisms empowering developers to declare the source code repository in the package metadata. For example, `setuptools`, the de facto build tool in the Python community, provides several optional keyword arguments (e.g., `url`, `project_urls`) in the `setup` function for package developers to declare package-related URLs in the package specification file such as `setup.py` [PyPA 2023c]. Then based on the package specification file, `setuptools` automatically generates release metadata [PyPA 2023a] in the packaging process. However, repository information in the metadata is still not the final answer to the package-repository linking problem. First, since the package's source code repository information is not mandatory for build tools, *package developers may not declare such information in the package specification file.* As shown in this paper, about 30% of PyPI releases' metadata do not contain repository information. Consequently, it is impossible to mine insights from source code repositories for these packages. Second, *package developers may declare wrong repository information in the package specification file* intentionally or unintentionally. An extreme case is the prevalent typosquatting attack [Duan et al. 2021; Ladisa et al. 2023; Ohm et al. 2020]. Malicious package developers usually copy the metadata (including the repository information) of popular packages they masquerade as. Another case is that developers did not change the placeholder repository URL. gh:pypa/sampleproject is a sample project that guides developers on packaging and distributing Python projects, but we find that 3,212 PyPI packages declare it as their source code repository. Wrong source code repository information tends to mislead existing package monitoring tools such as Libraries.io [Tidelift 2015] and open source insights [Google 2021c], and consequently, bias users' decisions.

Practitioners and researchers have implemented various tools [Microsoft 2020; PyPI 2023; Tidelift 2015; Vu 2021] to locate the repository for packages. However, these tools typically use the package metadata, which inevitably traps them into the two limitations mentioned above, i.e. wrong or unavailable repository information (that indicates where the repository is). To address the limitations, it is necessary to validate the repository information retrieved from the metadata and to utilize source code in the package to locate its repository when such information is unavailable in metadata. To that end, we begin with a large-scale empirical study in PyPI and then leverage the discovered insights to design a tool to retrieve the correct repository location for PyPI releases. Specifically, the empirical study explores two research questions:

- **RQ1:** *To what extent can existing tools retrieve source code repository information from the metadata and what are their differences?* Despite the plethora of metadata-based tools, there remains a lack of understanding about their capabilities. Thus, we propose this RQ to understand the status quo of techniques used to retrieve repository information from the metadata.
- **RQ2:** *What are the phantom file differences between correct package-repository links and incorrect links?* Phantom files are files appearing in the release's distribution but not in the release's repository, possibly indicating whether a release is built from a repository. Assuming the package-repository link is correct, prior work [Vu et al. 2021] investigated phantom files and found that Python files were rarely phantom files. However, to what extent phantom files differ between correct and incorrect package-repository links has not been investigated. Such understanding is vital for designing the tool that uses source code to validate and locate a package's repository.

We collect ecosystem-scale metadata of 4,227,425 PyPI releases to answer the questions. We find that existing tools retrieve repository information from the metadata for up to 70.5% of releases. We also identify several best practices such as URL redirection and searching from multiple information sources to locate the repository. We propose a heuristic approach to collect 14,375 correct and 2,064 incorrect package-repository links, and a novel Git repository traversal algorithm to accurately identify phantom files. We find that the number of phantom files in incorrect links is significantly higher and incorrect links are more likely to contain phantom package specification files.

Inspired by the empirical findings, we propose PyRadar, a novel framework that utilizes the release's metadata and source distribution to automatically **R**etrieve **A**nd vali**DA**te the source code **R**epository information for PyPI releases. PyRadar consists of three components: a **Metadata-based Retriever** similar to existing tools and two novel components to address the two limitations of existing tools, i.e., *a **Source Code Repository Validator** to deal with the limitation of incorrect repository information in the metadata*, and *a **Source Code-based Retriever** to address the limitation of missing repository information in the metadata*. Specifically, the Metadata-based Retriever combines best practices of existing tools and retrieves repository information for 72.1% of PyPI releases. The Source Code Repository Validator validates the correctness of the repository information retrieved by the Metadata-based Retriever. It contributes six crafted features, two of which are derived from the findings of RQ2. Common machine learning algorithms can achieve an AUC of up to 0.995 on these features, demonstrating their effectiveness. The Source Code-based Retriever uses source code in a release's source distribution to retrieve its repository from World of Code (WoC) [Ma et al. 2019, 2021], an infrastructure that collects almost all public Git repositories. Inspired by the findings of RQ2, we propose an efficient file hash-based repository retrieval algorithm. It retrieves repository information for 90.2% of packages in our dataset with an accuracy of 0.970 and completes a retrieval in an average of 40 seconds, demonstrating its effectiveness and efficiency.

In summary, the major contributions of this paper are:

- We conduct the first large-scale empirical study to compare existing metadata-based tools and investigate phantom file differences between correct and incorrect package-repository links.

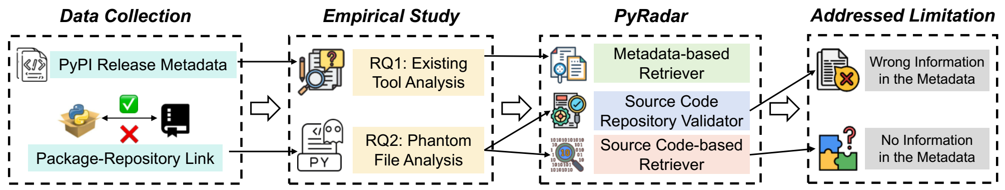
Data Collection
PyPI Release Metadata
Package-Repository Link
Empirical Study
RQ1: Existing Tool Analysis
RQ2: Phantom File Analysis
PY
PyRadar
Metadata-based Retriever
Source Code Repository Validator
Source Code-based Retriever
Addressed Limitation
Wrong Information in the Metadata
No Information in the Metadata

Fig. 1. Overview of this study

- We propose a heuristic approach to automatically and accurately collect correct and incorrect package-repository links to facilitate future research on this problem.
- We propose and evaluate PyRadar, a novel framework that utilizes the metadata and source distribution to automatically retrieve and validate repository information for PyPI releases. It works for all PyPI releases (about 88% of all PyPI releases) that provide source distributions.

Figure 1 demonstrates the overview of this study, including the data collection (Section 3), the empirical study (Section 4), and the PyRadar framework addressing the two limitations (Section 5).

2 BACKGROUND AND RELATED WORK

2.1 Terminology

Based on the official Python Packaging Glossary [PyPA 2023b] and prior work [Gao et al. 2024; Vu et al. 2021], we define the following terminologies for the convenience of discussion.

Package is a project registered on PyPI that '*is intended to be packaged into a distribution*'. A package can be considered as a folder consisting of a collection of files, with package specification files at the top-level folder. A package has one or more releases.

Release is a snapshot of a package at a certain time point and is identified by a version identifier. A release consists of one or more distributions.

Distribution is a versioned archive file that contains the Python package. Distribution is what the user will download from the package registries and install. There are two types of distributions: *source distribution* and *built distribution*.

Source distribution is a distribution format that contains package specification files and all essential files needed to generate built distributions.

Built distribution is another distribution format containing only metadata and files that will be copied to the correct location on the user's system at installation time. It contains pre-compiled files such as `.pyc` files and `.so`/`.dll` binary modules. However, Python files are not precompiled [PyPA 2023b]. Therefore, built distributions also contain Python source code.

Package specification file specifies build system information and release metadata. Currently, there exist multiple package specification files including `pyproject.toml`, `setup.py`, and `setup.cfg`. Multiple package specification files can coexist in a package.

Metadata contains descriptive information of a package such as name, version, and relevant URLs.

2.2 World of Code

World of Code (WoC) is an infrastructure for mining version control system data across the entire open source software ecosystem. It collects Git objects including commits, trees, and blobs [Chacon and Straub 2023] from nearly exhaustive public Git repositories on dozens of code hosting platforms such as GitHub, Bitbucket, and GitLab. Based on the collected Git objects, WoC provides several key-value databases for efficiently querying relationships between blobs, commits, repositories, and other relevant entities. For example, the blob-to-commit database maps a blob to all commits that introduced it; the commit-to-repository database maps a commit to all repositories that contain it. It is regularly updated and versioned. At the time of experimentation in this paper, the latest

Table 1. Existing tools that retrieve the PyPI package release's source code repository information

Tool	Provider	Language	Platform	Open Source
PyPI GitHub Statistics [PyPI 2023]	PyPI	Python	GitHub	✓
OSSGadget OSS Find Source [Microsoft 2023]	Microsoft	C#	GitHub	✓
Libraries.io [Tidelift 2015]	Tidelift	Ruby	Multi-platform	✓
py2src [Vu 2021]	Duc-Ly Vu	Python	GitHub	✓
Open Source Insights [Google 2021c]	Google	?	?	✗
Snyk Advisor [Snyk 2023]	Snyk	?	?	✗

version of WoC was labeled as U and the data was collected in October 2021, containing over 173 million Git repositories, 3.1 billion commits, 12.5 billion trees, and 12.4 billion blobs [swsc 2023].

2.3 Software Provenance

Software provenance refers to the origin and history of a software artifact such as code snippets and files [Godfrey 2015]. In this sense, finding the package's source code repository is a kind of software provenance task. Rousseau *et al.* [Rousseau et al. 2020] investigated the problem of file provenance in Software Heritage [Cosmo and Zacchiroli 2017], an infrastructure for preserving software source code. Hata *et al.* [Hata et al. 2021] studied how the same file evolves in different source code repositories. Reid *et al.* [Reid et al. 2022] developed a tool VDiOS based on WoC to detect vulnerabilities induced by file reuse. Wyss *et al.* [Wyss et al. 2022] proposed unwrapper to detect shrinkwrapped clones in NPM where a package duplicates or near-duplicates the code of another package. However, all these works are conducted at file-level granularity, different from the distribution-level granularity (i.e., a collection of files) required by finding the package's repository.

Sun *et al.* [Sun et al. 2023] proposed an identifier-based approach to map Debian source packages written in Python to PyPI packages. They indexed identifiers (classes and method/function names) in all PyPI packages and found that 76% of identifiers exist only in one package. Then they proposed an approach that randomly selects three non-frequent identifiers and queries the PyPI identifier corpus to locate the most probable PyPI packages. However, this work was conducted on the PyPI package corpus, which is different from and much smaller than the source code repository corpus.

There are also some tools that automatically retrieve the package's source code repository information as summarized in Table 1.

- PyPI GitHub Statistics [PyPI 2023] is provided by the PyPI website. Specifically, it detects GitHub repository URLs from the release's metadata and presents statistics for the retrieved GitHub repository such as the number of stars and forks in the sidebar of the package's PyPI page.
- OSSGadget [Microsoft 2020] is a collection of software supply chain tools released by Microsoft, one of which is OSS Find Source which attempts to locate a release's source code repository on GitHub. Similar to PyPI GitHub Statistics, OSS Find Source retrieves the release's source code repository information from the metadata.
- Libraries.io [Tidelift 2015] is an open source project maintained by Tidelift that collects package information from 32 package registries including PyPI. It also retrieves each package release's repository information from the metadata. Different from the above two tools, it detects repositories from multiple platforms.
- py2src [Vu 2021] is a tool proposed by a researcher from the University of Trento. It retrieves GitHub repository URLs from multiple information sources, including the package metadata and the websites referenced by the metadata (i.e., the package's homepage and Readthedocs page). Then it returns the GitHub URL with the most occurrences.

- Open Source Insights [Google 2021c] is a service developed and hosted by Google. It presents comprehensive information about the package such as vulnerabilities, dependencies, licenses, and the package's OpenSSF scorecard [OpenSSF 2023] information based on the GitHub repository it detects. Since this tool is not open source [Google 2021b], we have little information about its implementation. But it claimed that the detected repository information *is not guaranteed to be authoritative* as the package owner may list a link to any source code repository [Google 2021a].
- Snyk Advisor [Snyk 2023] is a service provided by Snyk. It presents the package's maintenance and community information based on the development activity data in the package's repository. It is also not open source. After manually inspecting several packages, we speculate it also retrieves repository information from the metadata and only extracts GitHub repository URLs.

To summarize, existing tools mainly retrieve the release's source code repository information from the metadata and do not validate the correctness of the retrieved repository information.

3 DATA COLLECTION

We build two datasets to conduct this study: 1) the Metadata dataset for comparing existing metadata-based tools (RQ1) and providing necessary information for the rest part of this study; 2) the Package-Repository Link dataset, which is used to investigate phantom file differences between correct package-repository links and incorrect links (RQ2) and lay a foundation on the design and evaluation of the Source Code Repository Validator and the Source Code-based Retriever.

Metadata dataset. We build this dataset with PyPI API [PyPI 2023] in March 2023. Specifically, we use the XML-RPC API to retrieve all packages registered on PyPI. We first get all its releases for each package and then get the metadata for each release using the JSON API. In total, we obtain metadata for 4,227,425 releases of 423,726 packages.

Package-Repository Link dataset. To validate the retrieved repository information of a release is correct or not, we need to build a dataset consisting of correct package-repository links and incorrect links. However, since the repository information in the metadata may be incorrect, *it is challenging to collect correct and incorrect package-repository links.* We, therefore, propose a heuristic approach to collect such data where we first collect correct package-repository links and then collect incorrect links based on the correct links.

We turn to the GitHub dependency graph [GitHub 2023] (GDG for short) to collect correct links. GDG identifies the packages associated with a GitHub repository, as shown in Figure 2. However, there are some problems when using GDG: 1) most GitHub repositories do not publish packages; 2) GDG detects packages across multiple packaging ecosystems, therefore packages published by different repositories may share the same name. For example, dmontagu/fastapi_client and kevinastone/django-api-rest-and-angular both publish the `example` package with the former as a Python package and the latter as a JavaScript package; 3) packages published by a repository may not be registered on PyPI and may even have the same name as PyPI packages. E.g., the `teras` package published by chantera/teras collides with the teras package on PyPI.

To address these problems, we first collect all source code repositories with at least 100 stars and written in Python using GitHub search API (to tackle the first and second problem), since popular repositories are more likely to publish packages [Borges et al. 2016; Wu et al. 2023]. We obtain 50,359 repositories in total. Then we collect packages published by these repositories by crawling each repository's dependency graph page, resulting in 14,471 packages. Next, we keep packages registered on PyPI by aligning GitHub package names with those in the Metadata dataset (to tackle the third problem), leaving 12,463 packages published by 11,803 GitHub repositories. We also include the top 4,000 most downloaded packages with their source code repository information retrieved by the Metadata-based Retriever. We consider these packages' repository information

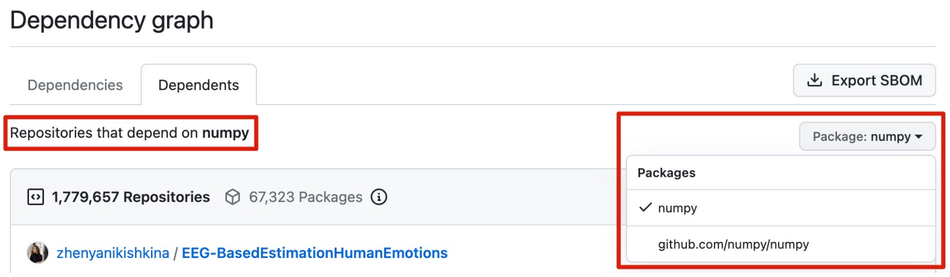

Fig. 2. GitHub dependency graph page of the repository gh:numpy/numpy

correct since they have a high impact on the Python community and have been widely studied in prior work [Vu 2021; Vu et al. 2021; Xu et al. 2023]. In total, we collect 14,375 correct links.

To evaluate the effectiveness of the approach to collecting correct links, we randomly sample 374 links (95% confidence level and 5% confidence interval). For each link, we check: 1) if the package name is declared in any package specification file in the repository; 2) if the package's PyPI maintainer [Foundation 2023a] (presented in the sidebar of the package's PyPI page) is a contributor to the repository (using common name abbreviation rules and user avatars); 3) if the repository or the documentation website referenced by the repository contains a link to the package's PyPI page or `pip install` commands with the package name. It is non-trivial to automate the three conditions accurately. Specifically, package developers can declare package names in the package specification file `setup.py` in various ways, posing challenges in automating the first condition. The second condition, linking PyPI accounts and GitHub accounts, involves a trade-off between precision and recall due to incomplete and inconsistent information on different platforms [Fang et al. 2020; Silvestri et al. 2015; Vasilescu et al. 2013]. The challenge of automating the third condition (searching for explicit mutual links between packages and repositories) is that relevant information is scattered and buried deep within repositories and related websites. Therefore, we choose manual checking over automated checking as it allows us to check the correct links with 100% accuracy, which is important for verifying the validity of collected correct links. We label a link as correct if it satisfies the first condition and one of the last two conditions. In total, 373, 369, and 349 links satisfy the three conditions respectively and 373 (99.7%) links are labeled as correct. The remaining link is not labeled as correct due to the complex packaging process of the `ansible` package [project contributors 2023]. It is noteworthy that the checking process is only a sufficient condition for a link to be correct. If a link does not meet the checking criteria, we can not assert it as incorrect. Overall, the manual inspection indicates the validity of the collected links.

Based on the correct links, we collect incorrect links as follows. We presume that the maintainers of PyPI packages are developers in the package's source code repository responsible for publishing releases. If two packages declare the same source code repository but have different maintainers, it is likely that one package's source code repository information is incorrect. Under this assumption, we select packages that have the same source code repository as packages in the correct links but have different PyPI maintainers. In this way, we obtain 1,721 links. As noted in Section 1, over 3,000 packages declare their source code repository as `gh:pypa/sampleproject` by mistake. Therefore, we expand the incorrect link data with these packages. Considering the large number of such packages, we randomly select 343 packages (95% confidence level and 5% confidence interval) to avoid them skewing the dataset. In total, we collect 2,064 incorrect package-repository links.

We verify the validity of incorrect links as follows. Since we are sure the incorrect correspondence between the 343 sampled packages and the `gh:pypa/sampleproject` repository, we sample 314 links (95% confidence level and 5% confidence interval) from the rest of 1,721 links. For each link, we manually check if the package name is specified in the source code repository. If not, we label it as

Table 2. The percentage of releases and packages for which the four reimplemented tools successfully retrieve repository information. Data in parentheses represents the adjusted percentage after URL redirection.

	PyPI GitHub Statistics	OSS Find Source	Libraries.io	PY2SRC
Releases	72.6% (67.2%)	72.7% (67.3%)	74.8% (68.4%)	75.5% (70.5%)
Packages	68.4% (60.9%)	68.5% (61.0%)	70.6% (62.2%)	70.2% (63.1%)

incorrect. In total, 312 (99.4%) links are labeled as incorrect. The rest two links are correct due to the move of the package's source code repository [Edward2 2019] and package renaming [coursera-dl 2016]. Overall, the inspection results suggest that the collected incorrect links are valid.

4 EMPIRICAL STUDY

4.1 RQ1: Existing Tool Analysis

Given the many metadata-based tools available, a clear understanding of their effectiveness and discrepancies can help better retrieve the release's repository information from the metadata. Specifically, we aim to understand the number of PyPI releases for which existing tools can retrieve repository information and the differences in the retrieved repository information.

4.1.1 Method. We select PyPI GitHub Statistics, OSS Find Source, Libraries.io, and PY2SRC for this RQ since they are open source, which enables us to make comparisons on the complete PyPI releases. However, since these tools are tightly tied to their contexts and used in different ways, it is difficult to deploy them on our Metadata dataset. We, therefore, choose to carefully reimplement them to facilitate the evaluation of their capabilities on the metadata of 4,227,425 PyPI releases. To ensure that the reimplemented tools are consistent with the original ones, we test them with test cases from the original tools. Then, we deploy them on our dataset and obtain the repository information retrieved by these tools for each release. To understand the discrepancies between these tools, we perform a stratified sampling of the differences in the retrieved repository information of the four tools and identify reasons for the differences based on the implementation of these tools. The number of sampled data for each pair of tools is shown in the parentheses of Table 3.

4.1.2 Results. Table 2 presents the percentage of releases and packages for which the four reimplemented tools retrieve repository information. It is worth noting that the retrieved repository information may be incorrect. We can observe that existing tools can retrieve repository information for about 3/4 of releases and 70% of packages, suggesting that substantial releases' metadata does not contain repository information. Libraries.io and PY2SRC retrieve repository information for more releases since Libraries.io takes more code hosting platforms into account and PY2SRC considers more information sources such as the homepage and Readthedocs page. The repository URLs retrieved by Libraries.io come from GitHub (3,032,984), GitLab (68,670), Bitbucket (53,668), SourceForge (5,918), ASF Subversion Server (2), and ASF GitBox Services (1). In the following, we only analyze the differences in the retrieved GitHub repository URLs.

The four tools retrieve different GitHub repository URLs for 511,480 (12.1%) releases. Table 3 presents the percentage of releases for which the four tools retrieve different GitHub repository URLs. The difference between PyPI GitHub Statistics and OSS Find Source is the least (0.16%) since they only differ in how the GitHub repository URL is extracted. PyPI GitHub Statistics uses the URL scheme while OSS Find Source uses the regular expression. PY2SRC differs from the remaining three tools a lot (10.47% ~ 11.86%).

Table 4 presents the seven identified reasons for the differences in the retrieved repository information of the four tools. URL redirection is the most common reason for the differences. Only PY2SRC deals with URL redirection when retrieving repository URLs from some information sources.

Table 3. The percentage of releases for which the four reimplemented tools retrieve different GitHub repository URLs. Data in parentheses represents the number of sampled differences for each pair of tools.

	PyPI GitHub Statistics	OSS Find Source	Libraries.io	py2src
PyPI GitHub Statistics	-	-	-	-
OSS Find Source	0.16% (2)	-	-	-
Libraries.io	2.06% (26)	2.02% (25)	-	-
py2src	10.47% (104)	10.43% (103)	11.86% (121)	-

Table 4. Reasons for the differences in the retrieved repository information of the four reimplemented tools.

Reason	Percentage
URL redirection	62.2% (237)
`project_urls` field searching	12.6% (48)
Badge URL searching	12.3% (47)
Readthedocs searching	8.9% (34)
URL Extraction method	5.2% (20)
Homepage searching	3.9% (15)
Other	1.6% (6)

We also present the adjusted percentage of releases and packages for which the four tools retrieve repository information after URL redirection in Table 2 (in parentheses). We can observe that existing tools can retrieve repository information for up to 70.5% of releases and 63.1% of packages. It suggests that URL redirection should be considered when retrieving repository information from the metadata due to the URL decay. The strategy of searching `project_urls` field accounts for the second most differences. This field is an arbitrary map of names to URLs. Libraries.io takes a rather conservative approach by searching only URLs whose names are in a predefined list, thus omitting repository URLs with other names. The badge URL, Readthedocs page, and Homepage searched by py2src account for about 1/4 of the differences. Different URL extraction methods also lead to different retrieved repository information. Specifically, PyPI GitHub Statistics and py2src extract repository URLs according to the URL scheme while OSS Find Source and Libraries.io rely on regular expressions. We find that regular expressions are more robust since the URL information provided by developers may not strictly observe the URL scheme. It is noteworthy that these reasons may contribute to inaccuracies when retrieving repository information from the metadata. For example, when the metadata contains multiple URLs linking to repositories on code hosting platforms, these tools may select different URLs due to different retrieval strategies, leading to incorrect repository information retrieved by certain tools.

Answers for RQ1: The four reimplemented tools can retrieve repository information from the metadata for up to 70.5% of releases and 63.1% of packages. The percentage of differences in the repository information retrieved by the four tools ranges from 0.16% to 11.86%. Seven reasons induce the differences such as URL redirection and the `project_urls` searching strategy.

Implications: When retrieving repository information from the metadata, it is necessary to take the following elements into account: URL redirection, multiple code hosting platforms, multiple information sources that are contained in the metadata, and badges, homepage, and Readthedocs page that are referred by the metadata.

4.2 RQ2: Phantom File Analysis

Prior work has revealed that most Python files in the release distribution also exist in the release's repository [Vu et al. 2021], which sheds light on how to validate the correctness of the release's repository information retrieved from the metadata and how to retrieve the release's repository using source code in the release's distribution. Thus, our second research question aims to understand the differences in phantom files (files appearing in the release's distribution but not in the release's repository) between correct package-repository links and incorrect links.

4.2.1 Method. The basic idea of obtaining phantom files is to traverse all files in the release's distribution and the release's source code repository, calculate their hashes, and find the files whose hashes appear in the distribution but not in the repository.

As noted in Section 2.1, a release consists of one or more distributions and there are two kinds of distributions in a release: source distribution and built distribution. We choose source distribution to obtain phantom files for two reasons:

- The number of releases providing source distributions is much higher than the number of releases providing built distributions (3,719,068 vs. 2,892,007). Therefore, the source distribution enables us to analyze more releases.
- For the 2,419,223 releases providing both source distributions and built distributions, we randomly select one release for each package, resulting in 243,518 releases. For each release, we compare files in the source distribution and built distribution. We find that files in the built distribution are all included in the source distribution for 216,741 (89.0%) releases. Among the remaining releases, the built distributions mostly contain pre-compiled files (such as `.pyc` files, `.so` binary modules), which are usually not included in the repository.

It is straightforward to get hashes for all files in the distribution. We download the distribution, open it with the Python standard library `tarfile` (for `.tar.gz` distribution files) or `zipfile` (for `.zip` distribution files) [Foundation 2023b], traverse files in it, and calculate each file's blob SHA-1 hash [Chacon and Straub 2023]. We choose the blob SHA-1 hash because Git uses it to index files so that we don't have to calculate hashes for files in the repository.

However, *it is challenging to traverse all files in the repository due to the complex Git-based development [Bird et al. 2009], especially the adoption of Git submodules [Scott and Ben 2023].* Submodules, which are configured in the `.gitmodules` file, allow developers to add a Git repository as a subfolder of another Git repository. We find submodules also popular in PyPI releases' repositories. Specifically, among the 3,047,112 releases for which the Metadata-based Retriever retrieves repository information, 159,360 (5.2%) releases' repository use submodules such as `numpy` and `scipy`. Files in submodules are also packaged into distributions. If not considering submodules when traversing files in the repository, many false positive phantom files will be identified. To make things more complicated, submodules are updated over time. To properly deal with Git submodules and accurately identify phantom files, we propose a novel Git repository traversal algorithm (Algorithm 1).

This algorithm traverses all commits to obtain a complete list of files in the repository. We use the Git command: `git cat-file --batch-check --batch-all-objects --unordered` (line 4) to ensure that all commits are traversed. For each commit, we obtain the tree object it points (line 6) and traverse the tree in a recursive way (traverse_tree) to obtain all files in the commit snapshot (line 7). Specifically, we first list all entries in the tree object, where each entry consists of the SHA-1 hash of a blob, tree, or commit object with its associated mode, type, and filename. We process each entry as follows:

- If the entry points to a blob object, we simply record its name and SHA-1 hash (line 14-15). If the blob's filename is `".gitmodules"`, we parse the path and URL of submodules configured in this file with the Python standard library `configparser` (line 16-17).

Algorithm 1: Traversing files in a Git repository

```
Input: A Git repository URL: $URL$
Output: A set of tuples with the file name and the file's SHA-1 hash: $\mathcal{F}$
1  function traverse($URL$)
2      $\mathcal{F} \leftarrow \emptyset$
3      $repo\_dir \leftarrow$ open_repository($URL$)
4      $C \leftarrow$ list_commits($repo\_dir, URL$)
5      for $commit\ c \in C$ do
6          $t \leftarrow$ get_root_tree($repo\_dir, c$)
7          $\mathcal{F} \leftarrow \mathcal{F} \cup$ traverse_tree($URL, t$)
8      return $\mathcal{F}$
9  function traverse_tree($URL, t$)
10     $f \leftarrow \emptyset$
11     $repo\_dir \leftarrow$ open_repository($URL$)
12     $submodules \leftarrow \emptyset$
13     for $entry\ e \in$ list_tree_entries($repo\_dir, t$) do
14         if $e$ is a blob object then
15             $f.add((e.filename, e.sha1))$
16             if $e.filename =$ ".gitmodules" then
17                 $submodules \leftarrow$ parse_gitmodules($e.content$)
18         else if $e$ is a tree object then
19             $f \leftarrow f \cup$ traverse_tree($URL, e$)
20         else if $e$ is a commit object then
21             $submodule\_url \leftarrow submodules.get(e.path)$
22             $submodule\_dir \leftarrow$ open_repository($submodule\_url$)
23             $submodule\_tree \leftarrow$ get_root_tree($submodule\_dir, e.sha1$)
24             $f \leftarrow f \cup$ traverse_tree($submodule\_url, submodule\_tree$)
25     return $f$
```

Table 5. Comparison of our algorithm with LastPyMile. ★ marks repositories using submodules.

Package	Repository	# of Distribution Files	# of Repository Files		# of Phantom Files	
			LastPyMile	Ours	LastPyMile	Ours
six	gh:benjaminp/six	11	743	743	1	1
certifi	gh:certifi/python-certifi	10	240	240	1	1
numpy	★ gh:numpy/numpy	2,231	64,506	63,719	296	4
scipy	★ gh:scipy/scipy	18,855	66,600	89,939	15,581	0

- If the entry points to a tree object, we traverse files in it and merge the result (line 18-19).
- If the entry points to a commit object, which indicates a submodule, we get the submodule's Git URL, obtain the tree object pointed by the commit object, and traverse files in the tree object of the submodule (line 20-24).

We implement this algorithm with as many operations provided by Git as possible to ensure a correct implementation. We make a simple comparison with LastPyMile proposed in prior work [Vu et al. 2021], which did not consider submodules when traversing the repository, on four popular packages, two of which do not use submodules in their repositories and the other two of which use submodules in their repositories. The comparison results (Table 5) indicate that our algorithm can properly deal with submodules and identify more accurate phantom files.

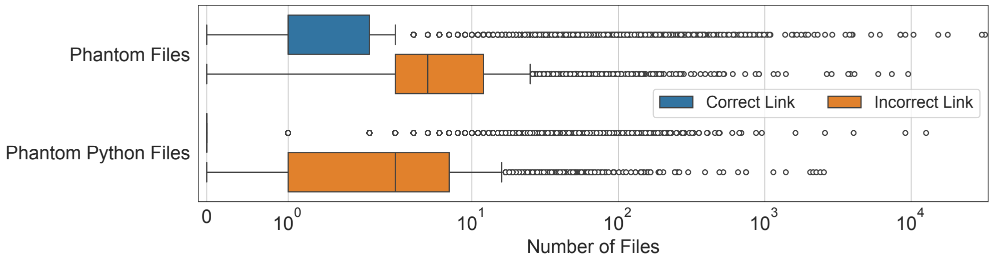

Fig. 3. Distribution of the number of phantom files and phantom Python files in correct and incorrect links.

Table 6. The 10 most common files and Python files in source distributions of in correct package-repository links. The two numbers in the cell correspond to the file's inclusion rate and phantom rate, respectively.

Files	Correct	Incorrect	Python Files	Correct	Incorrect
__init__.py	93.4%, 6.8%	93.7%, 48.5%	__init__.py	93.4%, 6.8%	93.7%, 48.5%
setup.cfg	87.8%, 98.1%	93.0%, 99.3%	setup.py	94.7%, 16.6%	93.0%, 96.4%
setup.py	94.7%, 16.6%	93.0%, 96.4%	utils.py	28.8%, 4.1%	25.9%, 31.1%
README.md	58.5%, 7.9%	55.5%, 58.0%	conf.py	12.8%, 4.4%	15.9%, 54.1%
MANIFEST.IN	54.7%, 6.1%	49.1%, 21.0%	exceptions.py	14.0%, 4.5%	15.3%, 24.1%
LICENSE	54.3%, 4.5%	43.5%, 31.0%	base.py	13.6%, 4.1%	12.8%, 37.9%
README.rst	33.5%, 6.6%	37.2%, 41.0%	__main__.py	13.7%, 8.2%	12.4%, 36.7%
utils.py	28.8%, 4.1%	25.9%, 31.1%	conftest.py	9.0%, 3.5%	12.1%, 57.0%
pyproject.toml	24.3%, 9.2%	25.2%, 64.6%	models.py	9.0%, 3.2%	10.6%, 37.6%
requirements.txt	16.3%, 5.2%	17.9%, 40.9%	version.py	11.7%, 18.6%	9.6%, 51.0%

We use the Package-Repository Link dataset to conduct this RQ. Specifically, for each (correct and incorrect) link in the dataset, we first download the source distribution of the package's latest release and obtain the blob SHA-1 hashes of files in the distribution. Then we clone the source code repository and obtain all blob SHA-1 hashes following Algorithm 1. Finally, we get phantom files by comparing SHA-1 hashes in the distribution and the repository.

4.2.2 Results. Figure 3 demonstrates the distribution of the number of phantom files and phantom Python files in the correct and incorrect links. It can be clearly observed that the number of phantom files and phantom Python files in the correct links is much lower than that in the incorrect links, with the significance confirmed by the Mann-Whitney U test [Mann and Whitney 1947]. Specifically, the median number of phantom files in the correct and incorrect links is 1 and 5 respectively, and the median number of phantom Python files is 0 and 3 respectively. It is also noteworthy that 75.3% of the correct links do not have phantom Python files whereas 96.2% of the incorrect links have at least one phantom Python file, suggesting that the number of phantom Python files may be useful to validate the correctness of a release's repository information and Python file in the release's source distribution may be used to retrieve the release's repository from WoC.

Table 6 presents the 10 most common files and Python files in source distributions of incorrect links. To derive effective features for identifying as much incorrect repository information for the release as possible, we calculate the inclusion rates and phantom rates of the files. The inclusion rate of a file is defined as the proportion of correct (incorrect) links that include the file in the source distribution compared to the total number of correct (incorrect) links. The phantom rate of a file is defined as the proportion of correct (incorrect) links in which the file is a phantom file compared to the number of correct (incorrect) links that include the file in the source distribution.

Almost every correct link and incorrect link include the package specification file `setup.py` or `pyproject.toml` in the source distribution, as revealed by the inclusion rates in correct links (99.5%) and incorrect links (99.2%). However, the phantom rate of `setup.py` or `pyproject.toml` differs greatly between correct (16.6%) and incorrect (97.0%) links. The results indicate that the presence of a phantom package specification file (`setup.py` or `pyproject.toml`) could serve as an effective feature in identifying incorrect repository information for the release: an inclusion rate close to 1 indicates that it is calculable for almost every package-repository link and the high phantom rate in incorrect links indicates that it achieves a high recall in identifying incorrect repository information. Despite the high phantom rate of README (`README.md` or `README.rst`) or LICENSE in incorrect links compared to correct links, the presence of a phantom README or LICENSE is not effective in identifying incorrect repository information for two reasons: 1) the relatively low inclusion rates (91.0% in correct links and 91.9% in incorrect links) suggest that it is incalculable for nearly 10% of links; 2) the low phantom rate (51.6%) in incorrect links indicates its low recall. Notably, the phantom rates of `setup.cfg` (another package specification file) in correct and incorrect links are both close to 1. We manually inspect 10 correct links with the phantom `setup.cfg` file and find that 7 of them do not contain the `setup.cfg` file in the repository, indicating that most of the phantom `setup.cfg` files are generated in the build process. Overall, the results indicate that whether the package specification file `setup.py` or `pyproject.toml` is a phantom file may be used to identify incorrect repository information for the release.

Answers for RQ2: The number of phantom files and phantom Python files in the incorrect links is significantly higher than in the correct links. Python files are generally not phantom files in the correct links. The percentage of phantom package specification files (`setup.py` or `pyproject.toml`) in the incorrect links is much higher than in the correct links.

Implications: The number of phantom Python files and whether the package specification file `setup.py` or `pyproject.toml` is a phantom file may be useful features to validate the correctness of a release's repository information. Retrieving a release's repository from WoC using only Python files in the release's source distribution may be sufficient and efficient.

5 THE PYRADAR APPROACH

Inspired by the empirical findings from Section 4, we propose PyRadar, a framework that utilizes the release's metadata and source code to automatically retrieve and validate the release's source code repository information. As shown in Figure 4, PyRadar consists of three components:

- Metadata-based Retriever. It retrieves repository information from the release's metadata.
- Source Code Repository Validator. It validates the correctness of the release's repository information retrieved by the Metadata-based Retriever. If the repository information is validated as correct, it will then be output.
- Source Code-based Retriever. If the Metadata-based Retriever fails to retrieve repository information from the release metadata or the Source Code Repository Validator concludes that the release's repository information retrieved from the metadata is incorrect, this component retrieves the release's repository from WoC using files in the release's source distribution. If this component fails to retrieve a repository, the output is empty.

In this section, we elaborate on the design and evaluation of each component.

5.1 Metadata-based Retriever

5.1.1 Design. We design this component following the best practices learned from existing metadata-based tools (Section 4.1). It first searches for repository URLs with regular expressions

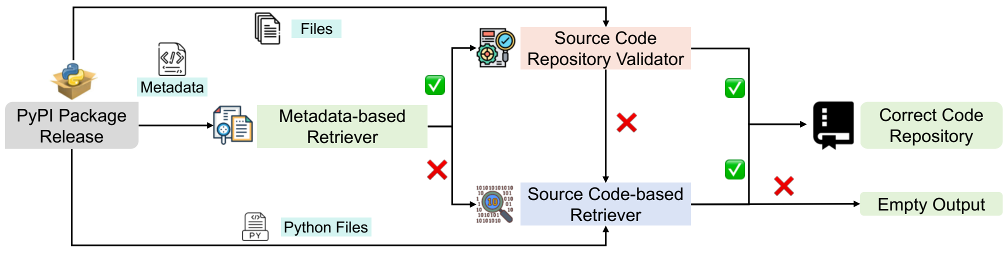

Fig. 4. Overview of the PyRadar framework

from the `url`, `download_url`, and `project_urls` fields in the release metadata sequentially. If any repository URL is found, it returns the redirected repository URL. Otherwise, it searches the `description` field in the release metadata for repository URLs and badge URLs. It only returns the redirected URL whose repository name is exactly the same as the package name after removing non-alphanumeric characters. If still no repository URL is found, it searches for the homepage URL and documentation page URL in the `project_urls` field, scrapes the homepage and documentation page, extracts repository URLs in them, and returns the redirected URL with the repository name exactly the same as the package name after removing non-alphanumeric characters. In the current implementation, it considers repositories on the three most popular code hosting platforms, i.e., GitHub, GitLab, and Bitbucket. It resolves URL redirections for GitHub repositories via the GitHub Repository API and for Bitbucket and GitLab repositories via HTTP requests.

5.1.2 Evaluation. We implement this component in Python and run it on the entire Metadata dataset. It successfully retrieves source code repository information for 72.1% of the releases (79.1% before URL redirection), 1.6% higher than the existing state-of-the-art tool py2src. The improvement of the Metadata-based Retriever over existing tools is minor, possibly because only about 70% of the releases' metadata contain repository information. Similar to other metadata-based tools, our Metadata-based Retriever will retrieve incorrect links when 1) the metadata does not contain a correct repository URL, or 2) the metadata contains a correct repository URL but the retriever selects another URL. Therefore, we design the Source Code Repository Validator to validate the correctness of the repository information retrieved by the Metadata-based Retriever.

5.2 Source Code Repository Validator

5.2.1 Design. The goal of this component is to validate the correctness of the repository information retrieved from the metadata. We design it as a classifier where the input is a pair of a release and a repository and the output is the probability of the input being an incorrect link. Inspired by the findings of RQ2 (Section 4.1) and prior work [Taylor et al. 2020; Vu 2021; Vu et al. 2020], we derive six crafted features (Table 7).

- According to Section 4.2, the correct link and incorrect link differ a lot on the number of phantom Python files and whether the package specification file `setup.py` or `pyproject.toml` changes. Therefore, we propose the `#phantom_pyfiles` and `pkg_spec_change` feature.
- Prior work [Vu 2021] manually checked the alignment of the repository tags and PyPI releases' version identifiers to determine a reliable repository for a release. After manually inspecting the release's version identifier and the repository's tag in some correct links, we find that the repository tags usually end with the release's version identifier (e.g., v1.2.1 and 1.2.1). Therefore, we propose the `tag_alignment` feature which is set to 1 if any tag in the repository ends with the release's version identifier otherwise 0.
- Name similarity is also used to determine a release's reliable repository [Vu 2021] and identify typosquatting packages [Taylor et al. 2020; Vu et al. 2020]. Therefore, we consider the

Table 7. Description of the six crafted features.

Feature	Description
#phantom_pyfiles	The number of phantom Python files
pkg_spec_change	Whether the package specification file `setup.py` or `pyproject.toml` is a phantom file or not
tag_alignment	Whether the release's version identifier aligns with any tag in the repository.
name_similarity	The normalized Levenshtein similarity between the package name and the repository name, ranging from $[0, 1]$.
#maintainers	The number of maintainers of the package
#maintainer_pkgs	The number of packages maintained by the package's maintainers

`name_similarity` feature useful for validating the correctness of the release's repository information as well. This feature is measured by the normalized Levenshtein similarity [Levenshtein et al. 1966] between the package name and the repository name, ranging from $[0, 1]$.

- We also take the release's maintainer information into account, i.e., the `#maintainers` and `#maintainer_pkgs` feature. Intuitively, if a package is maintained by more experienced developers, it is less likely to declare incorrect repository information.

We collect the six features for each link in the Package-Repository Link dataset,[1] including 14,375 correct links (labeled as 0) and 2,064 incorrect links (labeled as 1). Then we train machine learning models on the collected features. Specifically, we try seven commonly used models including Logistic Regression, SVM, Decision Tree, Random Forest, AdaBoost [Freund and Schapire 1997], Gradient Boosting Decision Tree [Friedman 2001], and XGBoost [Chen and Guestrin 2016]. Due to the imbalanced distribution of correct and incorrect links in the dataset, we employ resampling techniques on the training set for all models, which is a common practice to address the issue of unbalanced samples [Tian et al. 2022; Xiao et al. 2022]. We tune hyperparameters for each model with grid searching and select the best-performing model. Since the dataset is imbalanced and the relative ranking of incorrect links matters more, we choose AUC (Area under the ROC Curve) [Hanley et al. 1982] as the performance metric. AUC provides an evaluation of the model's performance across various classification thresholds and quantifies the probability that a random positive sample will have a higher ranking than a random negative sample [Melo 2013].

5.2.2 Evaluation. Table 8 presents the optimal performance of each model in identifying the incorrect link. We also present the accuracy, precision, and recall at a threshold of 0.5 as a reference. From Table 8, we can observe that either model archives an AUC higher than 0.95, possibly indicating the effectiveness of the six features. Random Forest outperforms the other models with an AUC of 0.995. It also achieves the highest recall score (0.989), suggesting that it can discover the most incorrect links in the test set. So we choose it as the classification model of this component. We also perform feature importance analysis to demonstrate how important each feature is for the fitted Random Forest model. Specifically, we use the permutation importance [Breiman 2001], a well-known technique to measure the contribution of each feature to the fitted model. It is calculated by randomly shuffling the values of a feature and observing the resulting degradation of the model's score. The results show that `name_similarity` is the most predictive feature (importance value: 0.117) followed by `pkg_spec_change` (importance value: 0.028) and `#phantom_pyfiles` (importance value: 0.026). The importance scores for the remaining three features are all below 0.010.

[1]We only select the latest release for each package.

Table 8. Performance of the seven machine learning models. Accuracy, precision, and recall are calculated at a threshold of 0.5.

Approaches	AUC	Accuracy	Precision	Recall
Logistic Regression	0.963	0.890	0.744	0.871
SVM	0.981	0.928	0.814	0.933
Decision Tree	0.982	0.966	0.918	0.954
Random Forest	**0.995**	0.973	0.913	**0.989**
AdaBoost	0.992	0.950	0.846	0.984
Gradient Boosting Decision Tree	0.992	**0.974**	0.936	0.963
XGBoost	0.991	0.972	**0.937**	0.956

We then use this component to validate the repository information retrieved by the Metadata-based Retriever. The Metadata-based Retriever successfully retrieves repository information for 3,047,112 releases (273,273 packages). We exclude packages in the Package-Repository Link dataset and select the latest release that provides the source distribution for each package, resulting in 228,448 releases. We then use the trained Random Forest model to output the probability of the link between the release and the retrieved repository being an incorrect link. We manually inspect the top 100 links with the highest probability and find that 85 (85%) of these links are indeed incorrect. For the remaining 15 correct links, we find the package name differs a lot from the repository name ($name\_similarity < 0.25$), which may lead to a high probability.

5.3 Source Code-based Retriever

5.3.1 Design. This component relies on the World of Code (WoC) [Ma et al. 2019, 2021] infrastructure due to its extensive collection of public repositories and convenient APIs. However, *the millions of repositories in WoC pose a great challenge in efficiently locating a release's correct repository based on source code.* To tackle this challenge, we propose a simple and efficient file hash-based algorithm based on the findings of RQ2. Specifically, this component retrieves a release's repository from WoC using Python files in its source distribution. We only use Python files since most of them are present in both the release's source distribution and the release's repository (Section 4.2), thus establishing a good link between the release and the repository. We use correct links in the Package-Repository Link dataset to design and evaluate this component. Among the 14,375 correct links, the repositories of 12,375 (86.1%) links are indexed by WoC.

This component first retrieves candidate repositories from WoC following the get_candidate function in Algorithm 2. For each Python file in the release's source distribution, it gets the first commit that introduces this file via the blob-to-commit database (line 4-5). Then it gets all repositories containing this commit (thus the file) via the commit-to-repository database (line 6). To speed up the retrieval and reduce the candidate set, we only consider files that do not exist in many repositories controlled by a threshold $blob\_uniqueness$ (line 7-8). Finally, we rank candidate repositories by the number of Python files in the source distribution that they contain.

To select the correct candidate, we manually inspect 373 links (95% confidence level and 5% confidence interval) from the 12,375 links and find that the top-ranked candidates are either the correct repository or forks of the correct repository in most (360, 96.5%) links. Therefore, we choose the $topn$ candidate repositories, find their upstream forked repository, and select the most common repository. To ensure the correctness of the retrieved repository, we only return the repository whose name similarity with the release's package name is above a threshold $name\_similarity$. Increasing the threshold improves the accuracy but reduces the percentage of releases for which this component

Algorithm 2: Retrieving repository from WoC

```
Input: A release's source distribution: $sdist$
Output: The most probable repository: $r$
function get_candidate($sdist, blob\_uniqueness$)
    $\mathcal{R} \leftarrow \emptyset$
    for Python file $f \in sdist$ do
        $blob\_sha \leftarrow$ calculate_sha($f$)
        $c \leftarrow$ get_first_commit($blob\_sha$)
        $repos \leftarrow$ query_c2p($c$)
        if $len(repos) \leq blob\_uniqueness$ then
            $\mathcal{R} \leftarrow \mathcal{R} \cup \langle repos, f \rangle$
    $R.rank()$
    return $\mathcal{R}$
function get_most_probable($sdist, blob\_uniqueness, topn, name\_similarity$)
    $repos \leftarrow \emptyset$
    $\mathcal{R} \leftarrow$ get_candidate($sdist, blob\_uniqueness$)
    for repo $r \in$ select_topn($\mathcal{R}, topn$) do
        $repos \leftarrow repos \cup$ defork($r$)
    $r \leftarrow$ most_common($repos$)
    if similarity($r, sdist.name$) $<$ $name\_similarity$ then
        $r \leftarrow null$
    return $r$
```

can retrieve repositories from WoC (i.e., the coverage), while decreasing it increases the coverage but reduces the accuracy. We heuristically set $blob\_uniqueness = 500$, $topn = 5$, $name\_similarity = 0.5$, which we find produce satisfactory results.

5.3.2 Evaluation. We evaluate the algorithm on the 12,375 correct links.[2] The algorithm can successfully retrieve repository information from WoC for 11,165 releases (i.e., coverage: 0.902) with an accuracy of 0.970. Table 9 shows the retrieval accuracy and coverage under different *name_similarity* setting. When setting *name_similarity* as 1, the accuracy increases to 0.986 but the coverage decreases to 0.757; when setting *name_similarity* as 0, the coverage increases to 0.986 but the accuracy decreases to 0.930. Therefore, we set *name_similarity* as 0.5 to strike a balance between retrieval coverage and accuracy. We run this component on the rest releases for which the Metadata-based Retriever can not retrieve repositories. We only select the latest release before October 2021 for each package, resulting in 81,751 releases. This component successfully retrieves repository information for 32,139 (39.3%) releases from WoC. The relatively low ratio may be attributed to the fact that many releases have repositories that are either not public or not indexed by WoC. We manually inspect 100 releases and find that this component correctly retrieves repository information for 90 releases, yielding an accuracy of 90%. On average, it takes 40 seconds to complete a repository retrieval, suggesting its high efficiency.

The Metadata-based Retriever and Source Code-based Retriever retrieve repository information using the release's metadata and source distribution, respectively. Therefore, we compare their retrieval results to further validate the two components. Specifically, the Metadata-based Retriever finds repository URLs for 3,047,112 releases of 273,273 packages. To conduct the comparison, we first select the most recent release before October 2021 (the time of WoC U version data collection)

[2]For each link, we use the package's latest release before October 2021, the time when the WoC version U data was collected.

Table 9. Retrieval coverage and accuracy under different *name_similarity* settings.

	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Coverage	0.986	0.985	0.976	0.954	0.927	0.902	0.876	0.843	0.803	0.768	0.757
Accuracy	0.930	0.932	0.939	0.954	0.965	0.970	0.975	0.979	0.982	0.985	0.986

for each package. Then, we keep releases that 1) provide a source distribution for the Source Code-based Retriever to use and 2) have their repository retrieved by the Metadata-based Retriever indexed by WoC. Finally, we obtain 173,809 releases. The Source Code-based Retriever successfully retrieves repository URLs for 143,035 (82.3%) of the selected releases, with 131,710 (92.1%) the same as the repository URLs retrieved by the Metadata-based Retriever, indicating a high consistency between the results of the two components.

5.4 Overall Evaluation

We evaluate PyRadar on the Package-Repository Link dataset with 14,375 correct and 2,064 incorrect links. For a release in the correct link, the expected output is the repository URL in the link. For a release in the incorrect link, the expected output is empty since there is no sufficient information to manually pinpoint its exact repository. As a result, PyRadar achieves an accuracy of 0.88.

6 LIMITATIONS

Several limitations pertain to the Package-Repository Link dataset. First, we rely on the black-box GitHub Dependency Graph to collect correct links. Therefore, we can not guarantee the accuracy of GDG data. To alleviate this threat, we carefully select popular Python repositories and conduct a manual evaluation to confirm the validity of the collected data. Second, the correct links are skewed towards popular packages and popular repositories, which may compromise the representativeness of the data. However, considering the number (14 375, 3.4% of all PyPI packages) and the monthly downloads (ranging from 28 to 1.02 million) of the collected packages, we believe the collected data are sufficiently representative. Third, we assume a link is incorrect where the package is linked to a repository that is the same as the repository in the correct link but with a different maintainer. The assumption has not been systematically tested and may threaten the validity of the collected data. We conduct a manual evaluation of the collected data to alleviate this threat.

In terms of the empirical study, its main threat is the reimplementation of the four tools. To alleviate this threat, we borrow test cases from the original tools to ensure the consistency of the reimplemented tools with the original ones. Also, due to the complexity of Git, the implementation of the repository traversal algorithm may threaten the internal validity. We use as many git native operations as possible to ensure the correctness of the implementation. Therefore, we believe the empirical study is conducted on a solid basis of correct implementation.

Despite the three most popular code hosting platforms considered by the Metadata-based Retriever, the package's source code repository may be hosted on other platforms such as SourceForge and other self-hosted platforms, e.g., GitLab at Inria. However, this component can be easily extended by adding new regular expressions to match repositories on these platforms. The major limitation of the Source Code-based Retriever is the dependence on the external infrastructure, WoC. Despite the relatively complete collection of open source Git repositories in WoC, it is impossible for WoC to contain all repositories. Besides, the several-month update [Gao et al. 2023] delay of WoC means that the Source Code-based Retriever can only be run periodically to retrieve repository information for releases uploaded to PyPI during the update interval. Despite these limitations, we believe WoC is still the most suitable infrastructure for our tool. Finally, PyRadar fails to retrieve a package's repository when 1) the package's repository is not public, e.g., hosted locally or privately. 2) the package's repository is public but is neither declared in the metadata nor indexed by WoC.

Due to the different usage scenarios and the challenge of automatically and accurately collecting ground truth, i.e., correct and incorrect package-repository links, the components are evaluated on different datasets, which may induce the overfitting issue. To evaluate the generalization of the second and third components, we run the second component against 228,448 package-repository links from the first component and run the third component against 81,751 releases for which the first component fails to retrieve repository information, which align with their practical usage scenarios. The manual inspection reveals that the two components achieve an accuracy of 85% and 90% respectively, suggesting the overfitting issue induced by different evaluation datasets is minor.

In terms of external validity, the dataset collection approach is dedicated to PyPI. We believe the correct link collection part can be generalized to other package registries since GDG supports multiple packaging tools. However, the incorrect link collection needs further research due to the differences in the package management practices adopted by different package registries [Duan et al. 2021]. The empirical findings can not be generalized to other package registries, too. However, the repository traversal algorithm can be applied to any repository, laying a foundation for future work on other package registries. Despite the specific design of PyRadar for PyPI packages, the framework is general and can be adapted to other package registries similar to PyPI (e.g., providing metadata and source distributions).

7 DISCUSSION

7.1 Comparison with Related Work

The most similar work to ours is [Sun et al. 2023]. There are two noteworthy differences between our approach and theirs. First, our approach and [Sun et al. 2023] serve different purposes. [Sun et al. 2023] targets the problem of mapping Debian source packages written in Python to PyPI packages, and our approach targets the problem of locating a PyPI package's code repository, which is more intricate due to the sophisticated Git-based software development [Bird et al. 2009] and a significantly larger retrieval corpus (244 thousand PyPI packages vs. 173 million repositories).

Second, due to different goals, [Sun et al. 2023] and the Source Code-based Retriever in our approach adopt different means but overlap in some particular steps in the whole process. Specifically, [Sun et al. 2023] involves three steps: 1) **index**: indexing global identifiers (i.e., class names and function names) defined in PyPI packages; 2) **retrieval**: retrieving candidate PyPI packages with three random identifiers from a random Python file in the Debian package; 3) **selection**: selecting the most popular candidate using the SourceRank metric. Our Source Code-based Retriever also has the retrieval and selection steps, but is superior in three aspects related to our problem:

- Our retriever does not require the time-consuming identifier indexing step. The phantom file analysis reveals that Python files are rarely phantom files in correct package-repository links. It suggests that querying candidate repositories with the hashes of Python files in the package is sufficient and does not require the time-consuming identifier indexing step that would *take months for the 12.4 billion blobs in WoC.*
- Our retriever utilizes more information to retrieve candidates. First, querying with a Python file's hash, as conducted in our retriever, is equivalent to querying with all identifiers in it, while [Sun et al. 2023] only queries with three identifiers. Second, our retriever uses all Python files, while [Sun et al. 2023] uses only a random Python file.
- Our approach selects the final candidate more suitably. First, the SourceRank metric used by [Sun et al. 2023] is unsuitable for ranking repositories due to its inclusion of many package-specific factors, such as the presence of a link to the source code. Therefore, the approach in [Sun et al.

2023] does not apply to our problem. Second, [Sun et al. 2023] selects the most popular candidate, whereas our retriever selects the most similar candidate to the package (measured by the number of matched Python files), which better suits our retrieval problem.

7.2 The Feasibility of Name-matching Approach

Although the package name and repository name are the same in 11,408 (79.36%) correct links, the name-matching approach is still insufficient for validating and locating a release's repository for three reasons. First, it is common that a package name differs from its repository name, as evidenced by the 20.64% of correct links, where the name-matching approach fails to retrieve the correct repository as a candidate. Second, even if the correct repository is retrieved as a candidate, the presence of many repositories with the same name necessitates further selection of the correct one. Specifically, the median number of repositories in WoC that have the same name as packages in the 14,375 correct links is 24. Third, when validating the correctness of the remaining 5,031 links, the AUC of using only the `name_similarity` feature is 0.563, while using all six features yields an AUC of 0.979, suggesting the effectiveness and necessity of the remaining five features.

7.3 Implication

Our research highlights the following future improvements in retrieving and validating the release's repository information.

Facilitate account linking between code hosting platforms and package registries. Cross-linking accounts can largely alleviate the problem of unavailable or incorrect repository information. On the one hand, package registries and code hosting platforms may collaboratively establish account binding or account authorization login mechanisms. On the other hand, to the best of our knowledge, account cross-linking tools between code hosting platforms and package registries are still lacking. Future work can bridge this gap.

Alert users to potentially incorrect repository information of the release. Package registries can integrate repository information validation mechanisms and display validation results on the package's PyPI page and in the metadata, which can be consumed by package managers and related package monitoring tools to alert users when searching for, assessing, or installing packages.

Conduct code analysis to identify package names built from a repository. As discussed in Section 5.2.2, our validator does not perform well in cases where the package name differs from the repository name greatly. To alleviate this issue, future work may explore code analysis techniques to precisely parse the package names built from the repository, which will benefit both the validation and retrieval of the release's repository information.

8 CONCLUSION

A package's source code repository is critical for the use and risk monitoring of the package. However, the package's metadata may not contain or contain wrong repository information. In this paper, we collect 4,227,425 PyPI releases' metadata, 14,375 correct package-repository links, and 2,064 incorrect links. Then we systematically compare four existing metadata-based tools and investigate phantom file differences between correct and incorrect links. Inspired by the empirical findings, we propose PyRadar, a novel framework that utilizes the PyPI release's metadata and source distribution to automatically retrieve and validate the release's repository information. We believe our work can help both practitioners and researchers better use PyPI packages. We provide a replication package at https://github.com/gaokai320/PyRadar.

ACKNOWLEDGMENTS

This work is sponsored by the National Natural Science Foundation of China 61825201 and 62332001.

REFERENCES

Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 446–457. https://doi.org/10.1109/SANER50967.2021.00048

Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 1–10. https://doi.org/10.1109/MSR.2009.5069475

Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344. https://doi.org/10.1109/ICSME.2016.31

Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

Scott Chacon and Ben Straub. 2023. Git - Git Objects. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects. (Accessed on 09/14/2023).

Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. https://doi.org/10.1145/2939672.2939785

Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017, Kyoto, Japan, September 25-29, 2017*, Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto (Eds.). https://hdl.handle.net/11353/10.931064

coursera–dl. 2016. Rename PyPI package name from "coursera" to "coursera-dl" · coursera-dl/coursera-dl@c2f318a. https://github.com/coursera-dl/coursera-dl/commit/c2f318a57183800a8fb9360761651690d7db3e5a. (Accessed on 04/25/2024).

Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1277–1286. https://doi.org/10.1145/2145204.2145396

Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/

Edward2. 2019. Move Edward2 from google-research/google-research to google/edward2. · google-research/google-research@f26db54. https://github.com/google-research/google-research/commit/f26db5490fa147a6052a78b2e479361833c3fd41. (Accessed on 04/25/2024).

Hongbo Fang, Daniel Klug, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. 2020. Need for Tweet: How Open Source Developers Talk About Their GitHub Work on Twitter. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 322–326. https://doi.org/10.1145/3379597.3387466

Python Software Foundation. 2023a. Help · PyPI. https://pypi.org/help/#collaborator-roles. (Accessed on 09/21/2023).

Python Software Foundation. 2023b. PEP 527 – Removing Un(der)used file types/extensions on PyPI | peps.python.org. https://peps.python.org/pep-0527/. (Accessed on 09/24/2023).

Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

Kai Gao, Runzhi He, Bing Xie, and Minghui Zhou. 2024. Characterizing Deep Learning Package Supply Chains in PyPI: Domains, Clusters, and Disengagement. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 97 (apr 2024), 27 pages. https://doi.org/10.1145/3640336

Kai Gao, Zhixing Wang, Audris Mockus, and Minghui Zhou. 2023. On the Variability of Software Engineering Needs for Deep Learning: Stages, Trends, and Application Types. *IEEE Transactions on Software Engineering* 49, 2 (2023), 760–776. https://doi.org/10.1109/TSE.2022.3163576

GitHub. 2023. About the dependency graph - GitHub Docs. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph. (Accessed on 09/21/2023).

Michael W. Godfrey. 2015. Understanding software artifact provenance. *Science of Computer Programming* 97 (2015), 86–90. https://doi.org/10.1016/j.scico.2013.11.021 Special Issue on New Ideas and Emerging Results in Understanding Software.

Google. 2021a. BigQuery dataset | Open Source Insights. https://docs.deps.dev/bigquery/v1/#packageversiontoproject. (Accessed on 04/24/2024).

Google. 2021b. Frequently Asked Questions | Open Source Insights. https://docs.deps.dev/faq/. (Accessed on 04/24/2024).

Google. 2021c. Open Source Insights. https://deps.dev/. (Accessed on 09/01/2023).

James A Hanley, Barbara J Mc Neil, and A Meaning. 1982. use of the area under a receiver Operating Characteristics (ROC) curves. *Radiology* 143, 1 (1982), 29–36.

Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Christoph Treude. 2021. Same File, Different Changes: The Potential of Meta-Maintenance on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 773–784. https://doi.org/10.1109/ICSE43902.2021.00076

Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A Large-Scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3468264.3468571

P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1509–1526. https://doi.org/10.1109/SP46215.2023.10179304

Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting Third-Party Libraries: The Practitioners' Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3368089.3409711

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 672–684. https://doi.org/10.1145/3510003.3510142

Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 143–154. https://doi.org/10.1109/MSR.2019.00031

Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data. *Empirical Softw. Engg.* 26, 2 (mar 2021), 42 pages. https://doi.org/10.1007/s10664-020-09905-9

H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

Francisco Melo. 2013. *Area under the ROC Curve.* Springer New York, New York, NY, 38–39. https://doi.org/10.1007/978-1-4419-9863-7_209

Microsoft. 2020. microsoft/OSSGadget: Collection of tools for analyzing open source packages. https://github.com/microsoft/OSSGadget. (Accessed on 09/01/2023).

Microsoft. 2023. OSS Find Source · microsoft/OSSGadget Wiki. https://github.com/microsoft/OSSGadget/wiki/OSS-Find-Source. (Accessed on 09/20/2023).

Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020), 110460. https://doi.org/10.1016/j.jss.2019.110460

Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.). Springer International Publishing, Cham, 23–43.

OpenSSF. 2023. OpenSSF Scorecard. https://securityscorecards.dev/. (Accessed on 09/20/2023).

Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E. Hassan. 2022. Automated Unearthing of Dangerous Issue Reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 834–846. https://doi.org/10.1145/3540250.3549156

Sebastiano Panichella, Gerardo Canfora, and Andrea Di Sorbo. 2021. "Won't We Fix this Issue?" Qualitative characterization and automated identification of wontfix issues on GitHub. *Information and Software Technology* 139 (2021), 106665. https://doi.org/10.1016/j.infsof.2021.106665

Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. https://doi.org/10.1145/3239235.3268920

Ansible project contributors. 2023. Releases and maintenance — Ansible Documentation. https://docs.ansible.com/ansible/devel/reference_appendices/release_and_maintenance.html. (Accessed on 09/22/2023).

PyPA. 2023a. Core metadata specifications — Python Packaging User Guide. https://packaging.python.org/en/latest/specifications/core-metadata/. (Accessed on 09/13/2023).

PyPA. 2023b. Glossary — Python Packaging User Guide. https://packaging.python.org/en/latest/glossary/. (Accessed on 09/16/2023).

PyPA. 2023c. Packaging and distributing projects — Python Packaging User Guide. https://packaging.python.org/en/latest/guides/distributing-packages-using-setuptools/#packaging-and-distributing-projects. (Accessed on 09/13/2023).

PyPI. 2023. Warehouse documentation. https://warehouse.pypa.io/. (Accessed on 09/01/2023).

David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The Extent of Orphan Vulnerabilities from Code Reuse in Open Source Software. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2104–2115. https://doi.org/10.1145/3510003.3510216

Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. 2020. Software provenance tracking at the scale of public source code. *Empirical Software Engineering* 25, 4 (01 Jul 2020), 2930–2959. https://doi.org/10.1007/s10664-020-09828-5

Chacon Scott and Straub Ben. 2023. Git - Submodules. https://git-scm.com/book/en/v2/Git-Tools-Submodules. (Accessed on 09/19/2023).

Giuseppe Silvestri, Jie Yang, Alessandro Bozzon, Andrea Tagarelli, et al. 2015. Linking Accounts across Social Networks: the Case of StackOverflow, Github and Twitter.. In *KDWeb*. 41–52.

Snyk. 2023. Snyk Open Source Advisor | Snyk. https://snyk.io/advisor/python. (Accessed on 09/01/2023).

Yiming Sun, Daniel German, and Stefano Zacchiroli. 2023. Using the uniqueness of global identifiers to determine the provenance of Python software source code. *Empirical Software Engineering* 28, 5 (20 Jul 2023), 107. https://doi.org/10.1007/s10664-023-10317-8

swsc. 2023. swsc / overview — Bitbucket. https://bitbucket.org/swsc/overview/src/master/. (Accessed on 09/18/2023).

Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *Network and System Security*, Mirosław Kutyłowski, Jun Zhang, and Chao Chen (Eds.). Springer International Publishing, Cham, 112–131.

Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What Makes a Good Commit Message?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2389–2401. https://doi.org/10.1145/3510003.3510205

Tidelift. 2015. Libraries.io - The Open Source Discovery Service. https://libraries.io/. (Accessed on 09/01/2023).

Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 356–366. https://doi.org/10.1145/2568225.2568315

Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 644–655. https://doi.org/10.1145/3236024.3236062

Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2013. StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge. In *2013 International Conference on Social Computing*. 188–195. https://doi.org/10.1109/SocialCom.2013.35

Duc-Ly Vu. 2021. py2src: Towards the Automatic (and Reliable) Identification of Sources for PyPI Package. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1394–1396. https://doi.org/10.1109/ASE51524.2021.9678526

Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792. https://doi.org/10.1145/3468264.3468592

Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 509–514. https://doi.org/10.1109/EuroSPW51379.2020.00074

Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/3377811.3380426

Jianyu Wu, Weiwei Xu, Kai Gao, Jingyue Li, and Minghui Zhou. 2023. Characterize Software Release Notes of GitHub Projects: Structure, Writing Style, and Content. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 473–484. https://doi.org/10.1109/SANER56733.2023.00051

Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork? Finding Hidden Code Clones in Npm. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2415–2426. https://doi.org/10.1145/3510003.3510168

Wenxin Xiao, Hao He, Weiwei Xu, Xin Tan, Jinhao Dong, and Minghui Zhou. 2022. Recommending Good First Issues in GitHub OSS Projects. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1830–1842. https://doi.org/10.1145/3510003.3510196

Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 860–871. https://doi.org/10.1145/3540250.3549125

Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. 2023. Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Kirchberg, Luxembourg) *(ASE '23)*. Association for Computing Machinery, New York, NY, USA.

Minghui Zhou and Audris Mockus. 2012. What make long term contributors: Willingness and opportunity in OSS community. In *2012 34th International Conference on Software Engineering (ICSE)*. 518–528. https://doi.org/10.1109/ICSE.2012.6227164

Minghui Zhou and Audris Mockus. 2015. Who Will Stay in the FLOSS Community? Modeling Participant's Initial Behavior. *IEEE Transactions on Software Engineering* 41, 1 (2015), 82–99. https://doi.org/10.1109/TSE.2014.2349496

Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2016. Effectiveness of Code Contribution: From Patch-Based to Pull-Request-Based Tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 871–882. https://doi.org/10.1145/2950290.2950364

Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman

Received 2023-09-28; accepted 2024-04-16