On the Accuracy of GitHub's Dependency Graph

Daniele Bifolco
University of Sannio
Benevento, Italy

Sabato Nocera
University of Salerno
Fisciano, Italy

Simone Romano
University of Salerno
Fisciano, Italy

Massimiliano Di Penta
University of Sannio
Benevento, Italy

Rita Francese
University of Salerno
Fisciano, Italy

Giuseppe Scanniello
University of Salerno
Fisciano, Italy

ABSTRACT

GitHub's dependency graph shows dependency relationships between repositories. This feature is leveraged by tools such as Dependabot, or GitHub's feature to export SBOM (Software Bill of Materials) files. Also, it has been used in empirical studies. Inaccuracies in the dependency graph might negatively affect both the effectiveness of tools and the results of the conducted studies. In this paper, we present the results of a mining study to assess the accuracy of GitHub's dependency graph in Java and Python open-source software projects. In particular, on April 16^{th}, 2023, we randomly sampled 297 software projects developed in Java and 338 developed in Python (all hosted on GitHub), each using GitHub's dependency graph. Then, we performed three analyses to assess how accurate GitHub's dependency graph is: (i) backward analysis, focusing on the accuracy of the dependencies of a given repository, as reported in GitHub's dependency graph; (ii) forward analysis, focusing on the accuracy of the dependents of a given repository, as reported in GitHub's dependency graph; and (iii) manifest/lock file analysis, focusing on the correspondence between the dependencies reported in the dependency graph of a given repository and what was reported in the corresponding manifest/lock files. The obtained results highlight several inaccuracies in GitHub's dependency graph, which might affect the output of tools based on GitHub's dependency graph (*e.g.,* Dependabot and SBOM generators) as well as the outcomes of past empirical studies. We also provide qualitative insights into these inaccuracies and implications for practitioners and researchers.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

GitHub, Dependency graph, Empirical study

ACM Reference Format:
Daniele Bifolco, Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. 2024. On the Accuracy of GitHub's Dependency Graph. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661175

EASE 2024, June 18–21, 2024, Salerno, Italy

ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661175

1 INTRODUCTION

According to *StackOverflow*'s 2022 Developer Survey, *GitHub* is the most popular version-control and software-development-hosting platform for both personal and professional use [22]. GitHub's users receive a steady stream of new features regularly. Among the most interesting features, there is one that, if enabled, allows showing the *dependency graph* [6] for each repository/project hosted on GitHub. In particular, such a kind of graph shows:

- the *dependencies*, namely the ecosystems and packages the repository depends on;
- the *dependents*, namely the ecosystems and packages that depend on the repository.

The dependency graph feature was introduced in 2017, yet only recently it has been adopted by many projects.

The information about the dependencies/dependents of a repository is inferred from the manifest and lock files—*i.e.,* files, such as `pom.xml` for Maven or `requirements.txt` for pip, specifying project build dependencies. Moreover, GitHub provides a feature to explicitly list dependencies for a repository by using the *dependency submission API* (beta). The dependency graph is automatically updated when either a commit changing a (supported) manifest/lock file to the default branch is pushed on GitHub or when anyone pushes a change to a dependent repository.

GitHub's dependency graph is leveraged by both researchers and practitioners. On the research side, there exist several studies analyzing software ecosystems that leverage the dependency graph [1, 12, 17, 18]. From the practitioner's perspective, some tools exploit GitHub's dependency graph to attain dependencies among software components. For example, *Dependabot* [7], monitors security vulnerabilities in the dependencies of a given repository and keeps the dependencies up-to-date. Based on statistics from 2022, *Dependabot* has, so far, opened over 75 million pull requests [24]. Another example of tools based on GitHub's dependency graph is the one that allows generating *SBOM* (*Software Bill of Materials*) files [9], released on GitHub in March 2023. Briefly speaking, if a repository leverages *Dependabot* or uses the GitHub SBOM generator, it inherently relies on the dependency graph.

Inaccurate information in the GitHub dependency graph would compromise the results of any analysis based on it. As for Dependabot, this would reduce its ability to send alarms to the dependents of vulnerable components, therefore opening them to security threats. Similarly, an incorrect dependency graph may result in an

incomplete or incorrect SBOM file, leading to security issues and licensing violations, in a scenario where the use of SBOMs has paramount importance [23, 26], given also the existing governmental regulations—*e.g.,* the US Federal Government, per Executive Order 14028, laid down that any company releasing software to federal agencies must provide an SBOM for the released software [14].

To assess the accuracy of GitHub's dependency graph, we conducted a mining study on Java and Python open-source software projects (all hosted on GitHub). We built two statistically significant random samples, with a confidence level of 95% and a margin of error of 5%, for a total of 635 software projects (297 developed in Java and 338 developed in Python). Then, we analyzed the accuracy of GitHub's dependency graph of these projects through three quantitative analyses. First, we performed a *backward analysis*, focusing on the accuracy of the dependencies, as they are reported in GitHub's dependency graph of a given repository. Second, we performed a *forward analysis*, focusing on the accuracy of the dependents. Third, we performed a *manifest/lock file analysis* in which we verified the correspondence between the dependencies reported in GitHub's dependency graph of a given repository and those reported in the corresponding manifest or lock files.

We observed inaccuracies in GitHub's dependency graph and provided qualitative insights into these inaccuracies. Based on the obtained results, we distilled practical implications for both researchers and practitioners. The former should pay attention to the dependency information provided by GitHub's dependency graph when conducting empirical studies based on it. The latter should be aware that the inaccuracy of GitHub's dependency graph might impact the effectiveness of any tool built upon it, such as recommenders, GitHub's Dependabot, or GitHub's feature to generate SBOM files. Finally, we deem that our qualitative insights might serve the purpose of improving the accuracy of GitHub's dependency graph.

Paper Structure. In Section 2, we discuss research related to ours. In Section 3, we show the design of our mining study. The obtained quantitative results are reported in Section 4, while we provide some qualitative insights in Section 5. The practical implications deriving from our results are discussed in Section 6 while the threats that might affect the validity of our results are discussed in Section 7. In Section 8, we conclude the paper and outline directions for future work.

2 RELATED WORK

In the following, we discuss related literature about studies leveraging GitHub's dependency graph and other studies on the analysis of dependencies in software ecosystems.

2.1 Studies Leveraging GitHub's Dependency Graph

Alfadel *et al.* [1] examined the degree to which developers adopt Dependabot for updating the vulnerable dependencies referring to a dataset composed of 2,904 JavaScript open-source projects subscribed to Dependabot. As a result, they observed that most of the suggested security pull requests (65.42%) were accepted and merged within a day. Mohayeji *et al.* [16] performed an empirical study on the use of Dependabot and its effectiveness in maintaining the dependencies secure in JavaScript projects. Their dataset was composed of 978 JavaScript projects with 4,195 security updates. Security advisories were retrieved by using GitHub's dependency graph. The results revealed that developers merge the majority of security updates signaled by Dependabot. In addition, when a security update is not automatically performed, developers often manually solve the identified security vulnerability.

He *et al.* [12] conducted an exploratory analysis on 1,823 popular and active projects hosted on GitHub, followed by a survey with 131 developers, to evaluate the effectiveness of Dependabot. The investigation revealed that the use of Dependabot reduces the technical lag and that developers are highly receptive to its pull requests. On the other hand, Dependabot revealed several limitations, including overcoming update suspicion and notification fatigue.

The GitHub's dependency graph feature was also exploited by Montandon *et al.* [17] to unveil the technical roles of developers on GitHub—specifically, the authors gathered projects' dependencies from GitHub's dependency graph.

Nocera *et al.* [18] analyzed repositories hosted on GitHub to identify those that adopted SBOM generators offered by *SPDX* and *CycloneDX*. The authors observed a low adoption of SBOM: only 186 repositories, even if the trend was increasing. The study exploited GitHub's dependency graph to retrieve repositories dependent on SBOM generators owned by SPDX and CycloneDX. The authors reported that they had excluded 11 repositories because they were not true cases of dependents for the studied SBOM generators. This raises questions about the reliability of the dependency graph, yet it does not provide a quantification of such inaccuracy, nor about its possible root causes. This outcome justifies studies like ours.

2.2 Studies on Software Ecosystem Dependencies

Besides studies leveraging GitHub's dependency graph, the existing literature reports other studies on the analysis of dependencies in software ecosystems. Bavota *et al.* [2] observed that the developers of the Apache Java ecosystem try to lower the impact that dependency upgrades would have when triggered by major releases and critical bug fixes.

Bogart *et al.* [4] focused on the R and npm ecosystems. The authors showed that developers do not use systematic approaches to cope with changes in these two ecosystems.

Zahan *et al.* [27] analyzed the metadata of 1,63 million JavaScript npm packages and proposed six alerts of security weaknesses in a software supply chain. One of their most important qualitative results is that the developers would want to be notified about weak link signals when using third-party packages.

3 STUDY DESIGN

The **goal** of our mining study is to analyze GitHub's dependency graph with the **purpose** of assessing its accuracy. The **perspective** is that of practitioners who have been using GitHub's dependency graph or tools based on it (*e.g.,* the one for generating SBOM files or Dependabot), and of researchers who have been founding their studies on GitHub's dependency graph. The **context** consists of Java and Python open-source software projects hosted on GitHub, and using Maven and pip/Poetry as package managers, respectively.

3.1 Research Questions

Based on the aforementioned study goal, we formulated and studied the following Research Question (RQ).

RQ. *How accurate is GitHub's dependency graph?*

The goal of this RQ is to assess the accuracy of both dependencies and dependents provided by GitHub's dependency graph. The rationale is understanding whether it can be used successfully in both research and practical development contexts, depending on whether or not the information it provides is correct and reliable. For example, if researchers base their studies on GitHub's dependency graph and find promising results, it does not matter how potentially useful they are because they could be incorrect. As for practitioners, an incorrect dependency graph, for example, may result in an incomplete or incorrect SBOM file, thus leading to possible security issues or licensing violations. Similarly, tools leveraging GitHub's dependency graph may provide inaccurate information to developers and therefore it could lead to improper decisions, *e.g.,* related to failing to upgrade a vulnerable software library.

3.2 Study Context and Planning

The context of our study is represented by Java and Python open-source software projects hosted on GitHub, the former using Maven as a package manager, the latter employing pip or Poetry as a package manager. We focused on Java and Python because of the popularity of these two programming languages. According to *2023 PYPL index*, Python is the most popular programming language, followed by Java [20]. Regarding the choice of the package managers, it was driven by the compatibility with GitHub's dependency graph—Maven was the only package manager compatible with Java, while pip and Poetry were the only package managers compatible with Python [6].

To search for software projects on GitHub, we leveraged *GHS* [5]—a tool to facilitate the selection of projects from GitHub for use in mining software repository studies. While doing so, we kept in mind the perils of mining GitHub by Kalliamvakou *et al.* [15]. Therefore, with the support of GHS, we searched for projects that met the following inclusion criteria:

(1) **Developed in Java or Python.** This filter lets us focus on popular programming languages widely used both in academic and industrial contexts.
(2) **Not archived and with at least one commit in the month preceding the date of the query (April 16^{th}, 2023).** This filter avoids selecting software projects no longer active [15].
(3) **With at least 100 stars, at least 100 commits, and at least one fork.** This mitigates the risk of selecting personal projects [15].
(4) **Not fork.** This limits the risk of selecting duplicate projects [15].

The search with GHS returned 7,172 software projects (2,296 Java projects and 4,876 Python projects) satisfying Criteria 1-4. Later, we applied an additional inclusion criterion that GHS did not support. Specifically, we made sure that the considered software projects were:

(5) **Using Maven, pip, or Poetry.** This criterion was introduced to ensure that the included projects used package managers compatible with the chosen programming languages and GitHub's dependency graph.

To check whether a software project met Criterion 5, we detected the use of the package manager (*i.e.,* Maven, pip, and Poetry) by looking for the corresponding manifest/lock files in that software project. More specifically, we detected the use of Maven in Java projects by checking—using the *PyGithub* library [13]—for the presence of `pom.xml` files. As far as Python projects are concerned, we detected the use of pip by verifying that they contained `requirements.txt`, `pipfile`, or `pipfile.lock` files, while we detected the use of Poetry from the presence of `poetry.lock` files. Note that pip and Poetry can leverage other manifest/lock file formats (*e.g.,* `setup.py` and `pyproject.toml`) different from those above mentioned. However, their use is not recommended with GitHub's dependency graph [6]. This was why we only focused on `requirements.txt`, `pipfile`, `pipfile.lock`, and `poetry.lock` files for Python projects.

We retrieved 1,330 Java projects using Maven and 2,948 Python projects using pip/Poetry, which represented our populations of interest. Due to the large size of these populations, we built two statistically significant random samples with a confidence level equal to 95% and a margin of error equal to 5%. As for the population of Java projects using Maven, the required sample size resulted to be 298. The sample size required for the Python projects using pip/Poetry was equal to 340. When sampling software projects from the populations of interest, we manually checked (by reading the project's description/README) whether or not each software project was:

(6) **Intended for software development.** This was to avoid selecting books, tutorials, or students' assignments [15].

If a project did not meet Criterion 6 or the project's description/README was not in English, we discarded it and randomly selected a new candidate project to be included in the sample. We decided to postpone the check of Criterion 6 at the time of sampling to avoid manually verifying each project in the populations of interest. This design choice does not negatively affect the statistical significance of our samples. This is because, if a project was not for software development, it should be removed also from the population of interest. Note that during the forward analysis (next section), we found three projects (one developed in Java and two developed in Python) that were archived after our sampling. Consequently, we had to discard these three projects. We did not replace these projects in the random samples because, after recomputing the required sizes of the random samples in light of the projects discarded for not meeting Criterion 6, our random samples were still large enough. That is, our random samples were still statistically significant with a confidence level of 95% and an error margin of 5%. In summary, we ended up with 297 Java and 338 Python projects. In Table 1, we report some descriptive statistics—*i.e.,* sum, mean, Standard Deviation (SD), min, median, and max—on the number of commits, contributors, stars, and forks of the studied projects.

Table 1: Some descriptive statistics of the studied Java and Python projects.

Language	Metric	Sum	Mean	SD	Min	Median	Max
Java	# Commits	1,070,792	3,605.36	10,448.09	100	1,262	134,597
	# Contributors	11,822	39.8	40.47	1	27	306
	# Stars	260,554	877.29	1,509.73	100	331	11,879
	# Forks	83,520	281.21	474.5	8	127	4,121
Python	# Commits	460,759	1,363.19	2,392.21	101	662	21,259
	# Contributors	11,354	33.59	49.3	1	17	402
	# Stars	382,797	1,132.54	2,639.82	100	339	25,777
	# Forks	66,757	197.51	448.52	4	76	4,461

Table 2: Some descriptive statistics for the analyzed dependencies.

Language	% Repositories with Dependencies	# Dependencies					
		Sum	Mean	SD	Min	Median	Max
Java	100% (297 out of 297)	60,803	204.72	425.36	1	57	3,027
Python	100% (338 out of 338)	14,901	44.09	85.58	1	16	1,158

Table 3: Some descriptive statistics for the analyzed dependents.

Language	% Repositories with Dependents	# Dependents (Excluding Repositories Without Dependents)					
		Sum	Mean	SD	Min	Median	Max
Java	63% (188 out of 297)	1,761,072	9,367.40	33,147.31	1	342	279,714
Python	57% (192 out of 338)	307,369	1,600.88	8,167	1	48	97,074

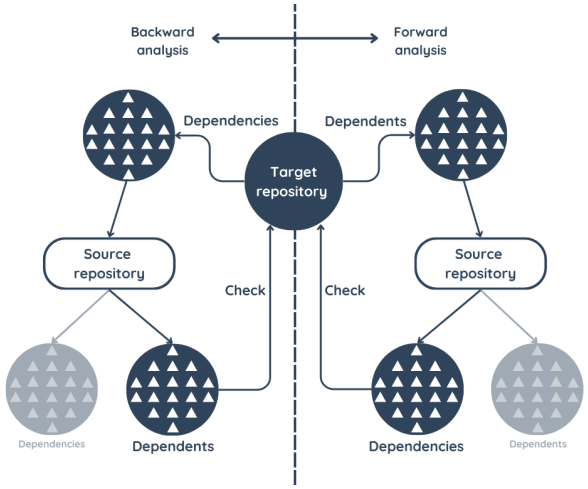

Figure 1: Backward and forward analysis.

3.3 Data Analysis

Before describing the methodology behind our data analysis, we introduce the terminology used in the remainder of the paper. In particular, with *target repository*, we refer to the repository object of the analysis, while with *source repository*, we refer to any repository that depends on or is dependent on a target repository.

To answer our RQ, we needed to retrieve the dependencies and dependents, and related information, of each software project from its dependency graph. To this aim, given the lack of GitHub-specific APIs for retrieving dependency graphs information, we parsing HTML pages of dependency graphs on GitHub leveraged a web scraping library by using *BeautifulSoup* [21]. After retrieving dependencies, dependents, and related information of each software project, we performed the following three quantitative analyses.

Backward analysis. This analysis focuses on the accuracy of the dependencies reported in GitHub's dependency graph. A graphical representation of the process to perform the backward analysis is depicted on the left-hand side of Figure 1. Given a target repository, we first looked at its dependencies, as reported in GitHub's dependency graph. Then, for each dependency, we accessed the corresponding source repository and checked whether the dependents of the source repository, as reported in GitHub's dependency graph, included the target one. As shown in Table 2, we analyzed 60,803 dependencies for Java projects and 14,901 dependencies for Python projects. Both Java and Python projects had at least one dependency, with a maximum of 3,027 dependencies for Java and 1,158 dependencies for Python projects. The average number of dependencies was equal to 204.72 and 44.09 for Java and Python projects, respectively. Further descriptive statistics concerning the dependencies studied are reported in Table 2.

Forward analysis. The focus of this analysis is on the accuracy of the dependents reported in GitHub's dependency graph. On the right-hand side of Figure 1, we show a graphical representation of the process for the forward analysis. Given a target repository, we first looked at its dependents, as reported in GitHub's dependency graph. Then, for each dependent, we accessed the corresponding source repository and checked whether the dependencies of the source repository, as reported in GitHub's dependency graph, included the target one. As shown in Table 3, 63% and 57% of the studied Java and Python projects, respectively, had at least one

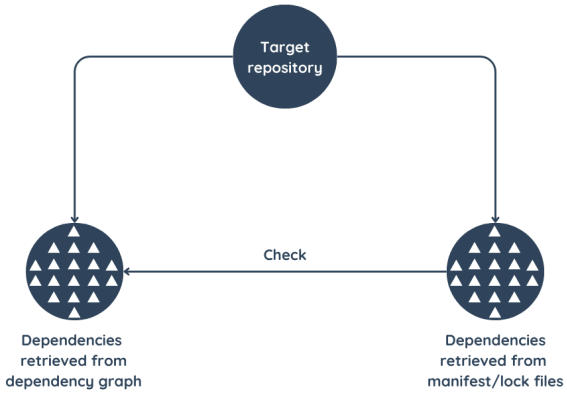

Figure 2: Manifest/lock file analysis.

Table 4: Backward analysis results of Java and Python dependencies.

Detection	Java		Python	
	Sum	%	Sum	%
Correct	45,707	75.17	12,589	84.48
Incorrect	15,096	24.83	2,312	15.52
Total	*60,803*	-	*14,901*	-

dependent. If we exclude the projects without dependents, the average number of dependents for the remaining Java and Python projects was equal to 9,367.40 and 1,600.88, respectively. In total, we analyzed 1,761,072 dependents for Java projects and 307,369 dependents for Python projects. Further descriptive statistics concerning the dependents studied are reported in Table 3.

Manifest/lock file analysis. This analysis focuses on how accurately GitHub's dependency graph reports the dependency information contained in manifest/lock files. A graphical representation of the process to perform the manifest/lock file analysis is shown in Figure 2. For each target repository, we retrieved its manifest/lock files at the time they were analyzed by GitHub's dependency graph, as we intended to analyze the same file versions. To that end, we considered the commit on which GitHub's dependency graph analyzed the manifest/lock files—GitHub's dependency graph makes available the dependency detection date—and queried GitHub through its API to retrieve the manifest/lock files at that version. Then, to obtain the dependencies listed in the manifest/lock files, we parsed these files. Finally, we checked, through a script, whether the dependencies retrieved from the manifest/lock files had a match in the dependency graph for that target repository. The match took into account (i) repository name, (ii) dependency name, (iii) path to the manifest/lock file; and (iv) dependency version.

A syntactic comparison was performed for each of the aforementioned features. If all the comparisons resulted in a positive match, we talk about a *match found*; otherwise, we talk about a *match not found*. It is worth noting that the dependency version could be omitted, while the other features could not. Therefore, if the dependency version is absent from both the dependency graph and manifest/lock files, we still talk about a match found, as long as there are positive matches for the other three features.

Table 5: Incorrect dependencies (out of total dependencies) grouped by error cases.

Case	Java		Python	
	Sum	%	Sum	%
Occurrence not found	687	1.13	1,612	10.82
Package not found	1,945	3.2	165	1.11
Source repository not found	53	0.09	30	0.2
No link to source repository	12,411	20.41	505	3.39

4 QUANTITATIVE RESULTS

We report here the quantitative results of our study per analysis, namely backward, forward, and manifest/lock file analyses.

4.1 Backward Analysis

As reported in Table 4, we analyzed a total of 75,704 dependencies, most of which were related to Java projects (60,803) and less to Python (14,901). This might be likely explained by the different granularity of Java and Python libraries, as also shown by the mean number of dependencies, greater for Java than for Python. As for Java projects, 75.17% of dependencies were correctly detected, so there is a match between the dependencies being analyzed and the dependents of those dependencies. The dependency graph provides incorrect information in 24.83% of the cases. As for Python projects, the percentage of correct dependencies is greater than for Java (84.48% vs. 75.17%). The causes for the incorrect dependency detection are:

(1) **Occurrence not found.** The dependency of a target repository does not have that target repository among its dependents.
(2) **Package not found.** GitHub's dependency graph may split up a target repository into *packages*. This error occurs when the dependency of a package of a target repository does not have that package listed among the packages of its dependents.
(3) **Source repository not found.** The dependency of a target repository is linked to a source repository that does not exist.
(4) **No link to source repository.** The dependency of a target repository is not linked to any source repository.

In Table 5, we report how often these causes occur in the analyzed projects. We report the total number of incorrectly detected dependencies and the percentage of incorrectly detected dependencies out of the total number of analyzed dependencies. For Java projects, the most frequent category of incorrectly detected dependencies is related to *no link to source repository* (20.41% of the total dependencies), whereas for Python projects the most frequent category is related to *occurrences not found* (10.82%).

In Table 6, we show the number of repositories for which at least one of the errors listed in Table 5 occurred during the backward analysis. Errors affected 90.57% of Java repositories and 73.08% of Python repositories. Therefore, the dependency graph provides accurate information on all the dependencies of Java and Python repositories in 9.43% and 26.92% of cases, respectively.

Table 6: Number of repositories affected by errors during the backward analysis.

Repositories	Java		Python	
	Sum	%	Sum	%
With error	269	90.57	247	73.08
Without errors	28	9.43	91	26.92
Total	*297*	-	*338*	-

Table 7: Forward analysis results of dependents.

Detection	Java		Python	
	Sum	%	Sum	%
Correct	1,701,626	96.62	218,044	70.94
Incorrect	59,446	3.38	89,325	29.06
Total	*1,761,072*	-	*307,369*	-

Table 8: Incorrect dependents (out of total dependents) grouped by error cases.

Case	Java		Python	
	Sum	%	Sum	%
Occurrence not found	31,885	1.81	86,292	28.07
Source repository not found	27,410	1.56	3,028	0.99
Repository not available	151	0.01	5	0

4.2 Forward Analysis

As shown in Table 7, we analyzed a total of 2,068,441 dependents, and most of them were related to Java repositories (1,761,072) rather than to Python ones (307,369). For Java projects, on the one hand, 96.62% of dependents were correctly detected, so there is a match between the dependents being analyzed and the dependencies of those dependents. On the other hand, the dependency graph provides inaccurate information in 3.38% of the cases. For Python projects, the percentage of correct dependents (70.94%) is lower than for Java projects. The causes for the incorrect dependent detection are:

(1) **Occurrence not found.** The dependent of a target repository does not have that target repository among its dependencies.
(2) **Source repository not found.** The dependent of a target repository is linked to a source repository that does not exist.
(3) **Repository not available.** The dependent of a target repository is unavailable because of policy violations—*e.g.,* GitHub's Terms of Service, DMCA (Digital Millennium Copyright Act) takedown.

As done for the backward analysis, we report in Table 8 how often the causes of incorrect dependent detection occur in the analyzed projects. For both Java and Python projects, the prevalent cause of incorrect dependent detection is *occurrence not found* (with 1.81% and 28.07% of cases, respectively).

As shown in Table 9, the incorrect detection of dependents affects 86.70% of Java projects and 78.13% of Python projects. Therefore, the dependency graph provides accurate information on all the dependents of Java and Python repositories in 13.30% and 21.88% of cases, respectively.

Table 9: Number of repositories affected by errors during the forward analysis.

Repositories	Java		Python	
	Sum	%	Sum	%
With error	163	86.7	150	78.13
Without errors	25	13.3	42	21.88
Total	*188*		*192*	

Table 10: Number of dependencies retrieved from manifest/lock files and dependency graph.

Source	# Dependencies	
	Java	Python
Manifest/lock files	68,004	14,858
Dependency graph	60,803	14,901

Table 11: Comparison between the number of dependencies retrieved from manifest/lock files and dependency graph.

Case	Java		Python	
	Sum	%	Sum	%
Lower	21	7.07	19	5.62
Equal	84	28.28	284	84.02
Greater	192	64.65	35	10.36
Total	*297*	-	*338*	-

4.3 Manifest/lock File Analysis

During this analysis, we failed to retrieve six manifest/lock files, four concerning Java projects and two concerning Python, from the corresponding commits (see Section 3.3). We retrieved and then analyzed 7,166 manifest/lock files. Out of these, 6,223 referred to Java projects and 943 to Python projects.

As shown in Table 10, the dependency graph identified 60,803 Java and 14,901 Python dependencies, respectively. On the other hand, the number of dependencies retrieved from manifest/lock files was higher for Java projects (7,201 additional dependencies than the dependency graph) and lower for Python projects (43 false positive dependencies).

In Table 11, we show the results of a comparison between the number of dependencies retrieved from manifest/lock files and those retrieved from GitHub's dependency graph, such a comparison takes into account each repository. In this way, we can see if there are differences between the number of dependencies detected from manifest/lock files and those that GitHub shows in the dependency graphs. For Java projects, the number of dependencies retrieved from manifest/lock files was greater than the number of dependencies retrieved from the dependency graph for 64.65% of repositories, while lower for 7.07% of them. The retrieved dependencies matched between manifest/lock files and the dependency graph for only 28.28% of the repositories. For Python projects, the number of retrieved dependencies was the same for 84.02% of repositories.

Table 12: Match between the dependencies retrieved from the manifest/lock files and those from the dependency graph.

Match	Java		Python	
	Sum	%	Sum	%
Found	52,490	77.19	14,220	95.71
Not Found	15,514	22.81	638	4.29
Total	*68,004*	-	*14,858*	-

For 10.36% and 5.62% of repositories, the manifest/lock files listed more and fewer dependencies, respectively, than those identified from GitHub's dependency graph.

Table 12 indicates to what extent the dependencies retrieved from manifest/lock files match those retrieved from the dependency graph. This analysis allows us to understand how consistent the information from the two sources is with each other. For Java projects, 77.19% of dependencies retrieved for manifest/lock files match those retrieved from GitHub's dependency graph; thus, there is a match between the following features: repository name, dependency name, file path, and dependency version (see Section 3.3). On the other hand, there is a mismatch in 22.81% of the cases. For Python projects, the percentage of correct matches (95.71%) is higher than that of Java projects (consequently, the percentage of mismatches, 4.29%, is lower).

Answer to RQ: The majority of the analyzed projects have inaccuracies on their GitHub's dependency graph, either in dependents or dependencies. Moreover, the dependency information from manifest/lock files and GitHub's dependency graph does not always match: 77% of Java dependencies have a match, while Python dependencies have a match in 96% of cases. GitHub's dependency graph seems to be more accurate for Python projects, likely due to the different ability to analyze dependency versions from Python requirements as opposed to `pom.xml` files.

5 DISCUSSION

In this section, we provide (qualitative) insights into the causes of the observed errors in GitHub's dependency graph by discussing some examples. The insights are arranged by grouping the cases we detected from the raw data obtained for each of the three kinds of conducted analyses: backward, forward, and manifest/lock file.

5.1 Backward Analysis Cases

Analyzing the raw data obtained from the backward analysis, we found different examples of inaccurate information, some of which are described in the following:

- ***Occurrence not found:*** as shown in Figure 3, the repository *apache/unomi* presents a *com.graphql-java:graphql-java* dependency. The dependency link correctly points to the repository *graphql-java/graphql-java*: however, the dependents section of the dependency graph reports *"We haven't found any dependents for this repository yet"*, despite *com.graphql-java:graphql-java* has at least one dependent. Such a kind of imprecision could be a concern if developers rely on mechanisms such as Dependabot to stay up-to-date and maintain the repository against security vulnerabilities in its dependencies. That is, this type of error can harm the accuracy of Dependabot's alerts.
- ***Package not found:*** the repository *awslabs/aws-saas-boost* has a *software.amazon.awssdk:dynamodb* dependency correctly linked to the source repository of *aws/aws-sdk-java-v2*; however, the dependents section does not list the corresponding package. We conjecture that this problem may be caused by the *visualization limits* (*i.e.,* the dependency graph only displays 100 manifests/lock files) of GitHub's dependency graph [11].
- ***Source repository not found:*** the repository *google/caliper* presents a *com.google.code.java-allocation-instrumenter:java-allocation-instrumenter* dependency linked to its source repository */QPC-WORLDWIDE/allocation-instrumenter*, but the latter does not exist. In this case, the issue may be due to incorrect linking to the source repository, or, because the repository has been deleted or made private.
- ***No link to source repository:*** as shown in Figure 4, the repository *apache/phoenix* presents the *org.jruby.jcodings:jcodings* dependency in two of its package manager files. As for the first row, the dependency is correctly linked to its source repository *jruby/jcodings* (see the left-hand side of Figure 4), while in the second row, the same dependency is not linked to its source repository (see the right-hand side). When analyzing the `pom.xml` files, the only difference between the two files is the presence of the dependency version in the former, while it is not present in the latter.

5.2 Forward Analysis Cases

We bumped into the following cases of inaccuracy:

- ***Occurrence not found:*** As shown in Figure 5, the repository *aws/aws-xray-sdk-python* presents a *vitoKdata/private_snowflake_streamlit* dependent correctly linked to his source repository. Still, in the dependencies section of the dependency graph of the latter, we do not find *aws-xray-sdk*. The possible cause of such a problem could be how both the dependency and dependent information is stored. In other words, when a link is deleted from one of the sides, such information is not updated on the other.
- ***Source repository not found:*** The package *org.glassfish.jersey.bundles:jaxrs-ri* of the repository *eclipse-ee4j/jersey* presents a *solskinIsak/3semSYS_Backend* dependent linked to its source repository, however the latter does not exist. In this case, the problem may be caused by improper linking to the source repository, or due to the repository being deleted or made private.
- ***Repository not available:*** The repository *apache/httpcomponents-client* presents a *timing1337/Mineplex* dependent correctly linked to its source repository; however, the latter is unavailable due to DMCA takedown. That is the act of requesting to take down copyrighted content from a website. If the repository *timing1337/Mineplex* contains any authorized copyrighted material, the copyright owner or its representative can

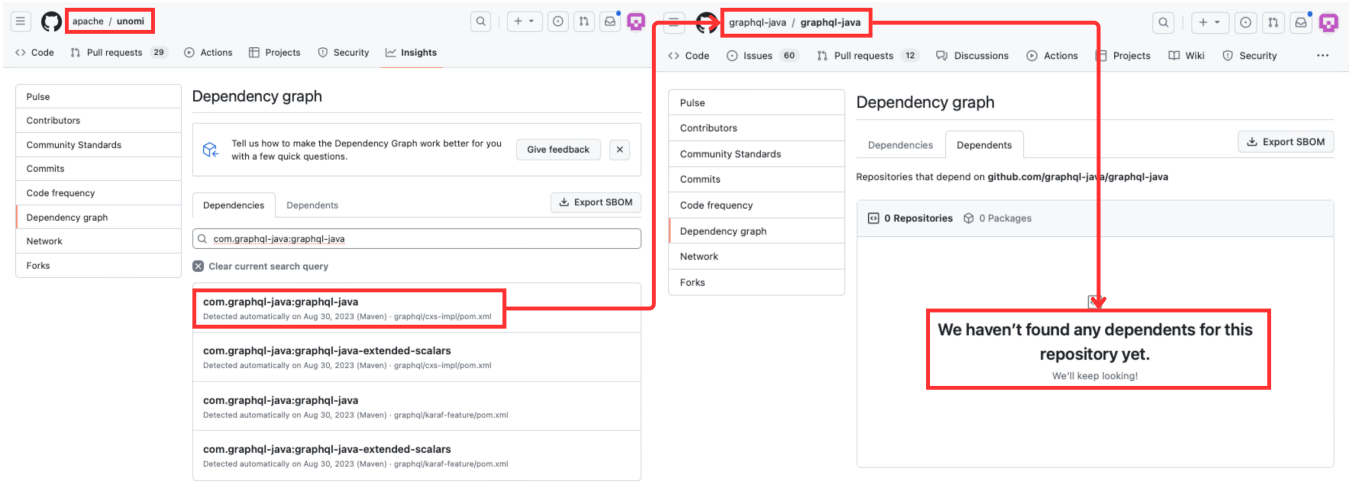

Figure 3: Example of an *occurrence not found* error in backward analysis.

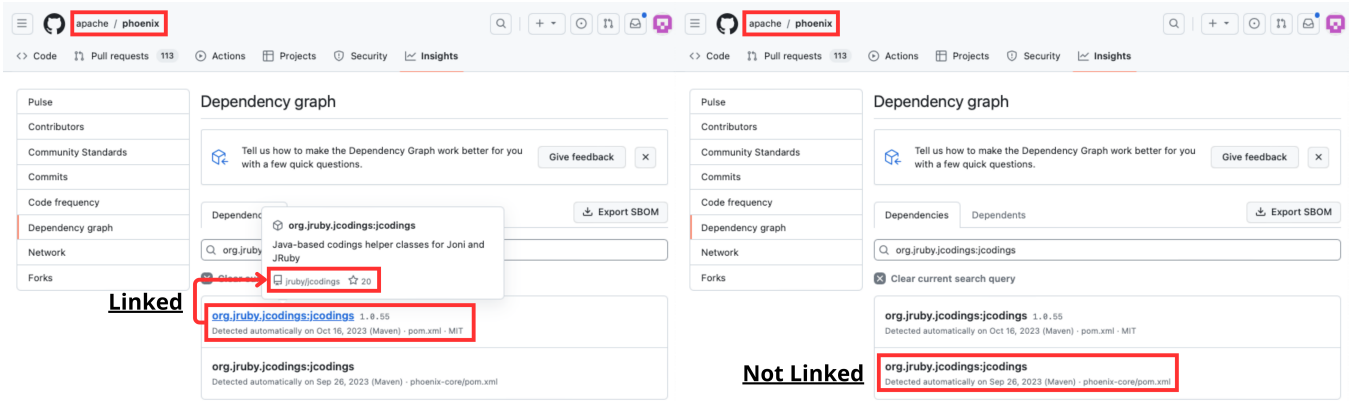

Figure 4: Example of a *no link to source repository* error in backward analysis.

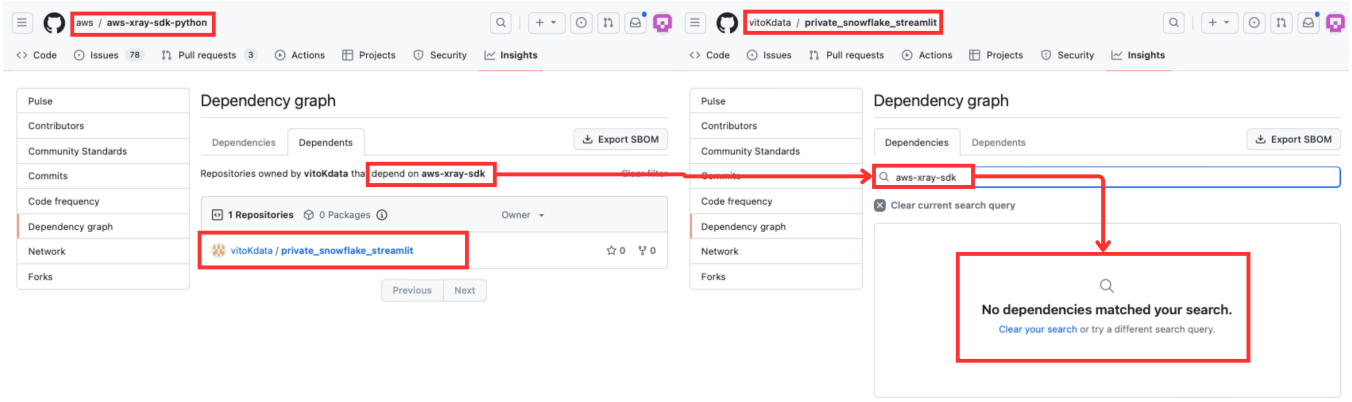

Figure 5: Example of an *occurrence not found* error in forward analysis.

issue a DMCA takedown request to GitHub [8]. Following such a request, GitHub would disable public access to the repository.

It is worth noting that between the time we automatically mined and analyzed the data and the time we qualitatively looked at data (five months after), error cases related to the *source repository not found* and *repository not available* categories have been resolved by GitHub. This means that some problems on GitHub's dependency

graph are being resolved and that the accuracy of such a graph may further improve in the future.

5.3 Manifest/lock File Analysis Cases

As shown in Section 4.3, there are several cases in which we do not find a perfect match between the dependencies retrieved manifest/lock files and those retrieved from GitHub's dependency graph. These cases mostly concern Java projects. In this regard, we noticed that when the `groupID` element of a dependency is not specified in the `pom.xml` file of a Java project, the dependency graph does not include that dependency. However, avoiding specifying the `groupID` element is not a mistake, because its value, if omitted, implies that it is defined in the hierarchically-superior `pom.xml` file.

A similar problem occurs when, in a `pom.xml` file, the `version` element of a dependency is specified with a variable, rather than a constant value. Again, in this case, GitHub's dependency graph may fail to report the right version of that dependency.

Another problem concerns the lack of a standard way to define version tags, as far as both Java and Python projects are concerned. For example, according to GitHub's dependency graph, the project *dromara/MaxKey* depends on the version *2.1.5* of *org.springframework.security.oauth.boot:spring-security-oauth2-autoconfigure*. However, in the corresponding `pom.xml` file, the right version is *2.1.5.RELEASE*. Other examples of version tags, for which we observed a mismatch with respect to the dependency version the dependency graph detected, are: *1.9.0-alpha2*, *1.7.5-SNAPSHOT*, *3.15.6.Final*, or *v2-rev65-1.17.0-rc*. It is worth mentioning that these examples are not exhaustive.

Finally, as reported in Section 4.3, we failed to retrieve six manifest/lock files from the corresponding commits. An example is represented by the repository *bcollazo/catanatron*. In particular, although this repository has the *flask* dependency correctly linked to its source repository when accessing the file in which *flask* dependency was detected, we obtained a "404 File Not Found" error. Since the file from which the dependency was detected is no longer available, the dependency graph should have been updated, however, this was not the case. This implies that the dependency graph does not constantly reflect the up-to-date status of a repository's dependencies. The timeliness of updating the dependency graph is critical for developers who rely on it to make informed decisions about dependency management and security.

6 IMPLICATIONS

For what concerns **researchers**, our study points out the imprecision of GitHub's dependency graph. For this reason, research work leveraging it for different purposes, for example studying software ecosystems or SBOMs [18] should, at the minimum, perform a manual validation of at least a significant sample of the extracted graph (if not the entire one) to determine the extent to which the work's findings are subject to imprecision. This, for example, was done in the work by Nocera *et al.* [18]. In addition to that, researchers willing to achieve a better level of accuracy should leverage further sources for dependencies—*e.g.,* by implementing more accurate analyzers for manifest/lock files.

For what concerns **practitioners**, as discussed before, the inaccuracy of GitHub's dependency graph can impact the recommender systems leveraging it. First of all, this impacts tools such as Dependabot. As a result, developers may not be able to update dependencies when needed, leading to potential vulnerability exposure with obvious economic fallout for the company. Similar considerations apply to developers that need to generate SBOM files and only leverage tools based on GitHub's dependency graph such as the SBOM generator. The inaccuracy of GitHub's dependency graph could also lead to non-identification of software license violations and this could result in legal problems with serious business consequences.

Last, but not least, our results can be of interest to the developers of the **GitHub** infrastructure. Related to that, recently GitHub has distributed a survey to its users, requesting feedback on their use of GitHub's dependency graphs to help improve it [10]. In our work, we describe the type of errors that can occur in the dependencies and dependents from GitHub's dependency graph. These error cases might, indeed, contribute to improving the accuracy of GitHub's dependency graph.

7 THREATS TO VALIDITY

In this section, we discuss the threats that might affect our results. We do not discuss threats to internal validity since we do not investigate causality, if not by discussing some qualitative examples.

Construct Validity. Threats to construct validity concern the relation between theory and observation [25]. In our study, such a kind of threat regards the quantitative analyses—*i.e.,* backward, forward, manifest/lock file analyses—we performed to assess the accuracy of GitHub's dependency graph, along with the used metrics. We leveraged widely used libraries to analyze the artifacts necessary for the analysis, carefully tested our scripts, and complemented our quantitative analysis with a qualitative one.

Conclusion Validity. Threats to conclusion validity concern the statistical conclusions, including sample composition and size [25]. We built two statistically significant random samples, one for Java projects and another one for Python projects, with a confidence level equal to 95% and a margin of error equal to 5%, rather than analyzing any project in the populations of interest. Although this is a very common procedure to avoid analyzing any subject in a population of interest (*e.g.,* [19, 28]), it might pose a threat to conclusion validity.

External Validity. Threats to external validity concern the generalizability of results [25]. Our results might not be generalized to software projects developed in programming languages different from Java and Python. In this respect, our results suggest that the accuracy of GitHub's dependency graph differs when considering Java and Python projects. Therefore, our results suggest future work focusing on other programming languages. Finally, our results might not reflect the current accuracy of GitHub's dependency graph, since we mined GitHub in April 2023, and some inaccuracies might have been resolved in the meantime.

8 CONCLUSION

We presented the results of a mining study to assess the accuracy of GitHub's dependency graph in Java and Python open-source software projects. To this end, we performed a: *(i) backward analysis*, focusing on the accuracy of a given repository's dependencies, as reported in the dependency graph; *(ii) forward analysis*, focusing

on the accuracy of the dependents, as reported in the dependency graph of a given repository; and *(iii) manifest/lock file analysis* focusing on the correspondence between the information reported in the dependency graph and what was reported in the corresponding manifest/lock files. The three different analyses reported at least one inaccuracy in the majority of the analyzed projects. We also qualitatively discussed the nature and possible causes of these inaccuracies. The results of our study have several practical implications. For example, imprecision in GitHub's dependency graph could affect recommenders (*e.g.,* Dependabot) and other tools based on it such as GitHub's SBOM generation tool.

As possible future work for our research, we will conduct a study to understand to what extent imprecision in GitHub's dependency graph affects past studies based on it. We also plan to replicate our study to understand if GitHub improved over time the way the dependency graphs are built and updated.

DATA AVAILABILITY

The study replication package, containing raw data, scripts, and details on the studied software projects, is available online [3].

ACKNOWLEDGMENTS

This project has been financially supported by the European Union NEXTGenerationEU project and by the Italian Ministry of the University and Research MUR, a Research Projects of Significant National Interest (PRIN) 2022 PNRR, project n. D53D23017310001 entitled 'Mining Software Repositories for enhanced Software Bills of Materials (MSR4SBOM)'. Daniele Bifolco is partially funded by the PNRR DM 118/2023 Italian Grant for Ph.D. scholarships.

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. In *Proceedings of International Conference on Mining Software Repositories*. IEEE, 254–265.
[2] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empir. Softw. Eng.* 20 (2015), 1275–1317.
[3] Daniele Bifolco, Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. 2023. On the Accuracy of GitHub's Dependency Graph: A Replication Package. https://figshare.com/s/81e96d4864f4ebc5e25c. https://doi.org/10.6084/m9.figshare.24441289
[4] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering Workshop*. IEEE, 86–89.
[5] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *Proceedings of International Conference on Mining Software Repositories*. IEEE, 560–564.
[6] GitHub. 2023. About the Dependency Graph. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.
[7] GitHub. 2023. Dependabot. https://docs.github.com/en/code-security/dependabot.
[8] GitHub. 2023. DMCA Takedown Policy. https://docs.github.com/en/site-policy/content-removal-policies/dmca-takedown-policy.
[9] GitHub. 2023. Exporting a Software Bill of Materials for Your Repository. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/exporting-a-software-bill-of-materials-for-your-repository.
[10] GitHub. 2023. *Help improve GitHub dependency graph with your feedback!* https://github.com/orgs/community/discussions/43364
[11] GitHub. 2023. Troubleshooting the dependency graph. https://docs.github.com/en/enterprise-server@3.9/code-security/supply-chain-security/understanding-your-software-supply-chain/troubleshooting-the-dependency-graph#are-there-limits-which-affect-the-dependency-graph-data.
[12] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Trans. Softw. Eng.* 49, 8 (2023), 4004–4022.
[13] Vincent Jacques. 2023. PyGithub. https://pygithub.readthedocs.io/en/stable/.
[14] Joe Biden. 2021. *Executive Order on Improving the Nation's Cybersecurity.* https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of Mining Software Repositories*. ACM, 92–101.
[16] Hamid Mohayeji, Andrei Agaronian, Eleni Constantinou, Nicola Zannone, and Alexander Serebrenik. 2023. Investigating the Resolution of Vulnerable Dependencies with Dependabot Security Updates. In *Proceedings of 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 234–246.
[17] João Eduardo Montandon, Marco Tulio Valente, and Luciana L. Silva. 2021. Mining the Technical Roles of GitHub Users. *Inf. Softw. Technol.* 131 (2021), 106485.
[18] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. 2023. Software Bill of Materials Adoption: A Mining Study from GitHub. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE.
[19] Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. 2021. Why Do Developers Reject Refactorings in Open-Source Projects? *ACM Trans. Softw. Eng. Methodol.* 31, 2 (2021), 1–23.
[20] PYPL. 2023. 2023 PYPL Index. https://pypl.github.io/PYPL.html.
[21] Leonard Richardson. 2023. BeautifulSoup. https://www.crummy.com/software/BeautifulSoup/.
[22] Stack Overflow. 2022. 2022 Developer Survey. https://survey.stackOverflow.co/2022.
[23] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. 2024. BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems. In *Procedings of International Conference on Software Engineering*. ACM.
[24] Eric Tooley and Erin Havens. 2023. A Smarter Quieter Dependabot. https://github.blog/2023-01-12-a-smarter-quieter-dependabot/.
[25] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer.
[26] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2630–2642.
[27] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What Are Weak Links in the Npm Supply Chain?. In *Proceedings of International Conference on Software Engineering: Software Engineering in Practice*. ACM, 331–340.
[28] Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. An empirical characterization of software bugs in open-source Cyber–Physical Systems. *J. Syst. Softw.* 192 (2022), 111425.