Assessing the Quality of GitHub-Generated SBOMs: A Detailed Bill of Health

Grant Enderson
Computer Science Department
William & Mary
Email: gmenderson@wm.edu

Nathan Wintersgill
Computer Science Department
William & Mary
Email: njwintersgill@wm.edu

Trevor Stalnaker
Computer Science Department
William & Mary
Email: twstalnaker@wm.edu

Oscar Chaparro
Computer Science Department
William & Mary
Email: oscarch@wm.edu

Denys Poshyvanyk
Computer Science Department
William & Mary
Email: dposhyvanyk@wm.edu

***Abstract*—Software Bills of Materials (SBOMs) are emerging as vital tools for software supply chain management, providing a means of tracking dependencies with their licenses and promoting transparency. Despite increased regulatory pressure for these documents, accompanying tool support is still nascent. Recently, GitHub introduced SBOM generation for repositories via its built-in dependency tracking system. In this paper, we investigate this tool to ascertain its efficacy in generating accurate SBOMs, identifying the causes of mistakes in generated SBOMs and reasoning about challenges currently facing SBOM generation. We create a dataset of 50 manually-verified SBOMs for Python projects and compare them against those generated by GitHub, identifying areas where the tool struggles. Our results shed light on the GitHub tool's strengths and weaknesses and suggest directions for future improvements.**

I. Introduction

Software Bills of Materials (SBOMs) are machine-readable artifacts that outline the libraries, frameworks, and other components utilized by software systems [1]. These inventories are meant to enhance accountability, traceability, and transparency within the software supply chain, enabling better vulnerability management, license compliance, and dependency tracking [2]. These benefits, along with regulatory guidance, have increased adoption of SBOMs [3–7].

Currently, two major SBOM standards exist: SPDX [8] and CycloneDX [9]. These standards define the fields, values, structure, and file formats necessary for creating SBOMs. Numerous tools have been developed to automatically generate SBOMs in these standards [10], for software systems written in various programming languages. GitHub's SBOM generation tool [11], for example, is integrated into the GitHub platform and leverages GitHub's Dependency Graph [12] to create SPDX SBOMs, making it straightforward for developers and users of any GitHub-hosted project to generate an SBOM by simply navigating to the Dependency Graph page and clicking the "Export SBOM" button, or by using GitHub's SBOM API [13].

However, incompleteness and inaccuracy of the generated SBOMs impedes their adoption, as highlighted by recent studies with SBOM creators and consumers [3, 14–17]. Current SBOM generation tools often fail to capture all dependencies or provide correct or complete information about them (*e.g.*, versions and licenses). Low-quality SBOMs can lead to unaddressed vulnerabilities, license conflicts, and/or (in the case of false positives) wasted time, which can have major repercussions for organizations and users [15].

No systematic evaluations explore in detail the underlying reasons for these problems. While previous work has provided a preliminary analysis of SBOM quality and the quality differences across tools, such as SBOM tools for Java [18] and Javascript [19], the literature lacks a thorough manual investigation into the causes of quality issues and the challenges faced by humans and tools in producing high-quality SBOMs.

In this paper, we report an in-depth evaluation of the completeness and accuracy of SBOMs generated by GitHub for 50 popular and active Python projects of various sizes and domains. To accomplish this, we produced a novel dataset of human-curated SBOMs to use as a benchmark, created through manual analysis of the targeted software components and their dependencies. We focus on GitHub's SBOM generator tool due to its widespread availability and ease of use, and on Python due to its popularity and potentially unique challenges associated with generating SBOMs for interpreted languages based on manifest files.

Using this novel ground truth dataset, we analyzed the GitHub SBOM generator's ability to accurately produce SBOMs for these target software repositories. We found that while GitHub's tool performs well at identifying attributes of source repositories, it struggles to accurately capture dependency information. Attributes like versioning and licensing information are frequently lost. We investigate cases where the SBOM generator failed to include relevant components or included components that it should not have, and classify the causes of deficient SBOMs through a novel taxonomy. The results show that the tool can be overzealous in including sources which it should not, while improper documentation from users can also play a role. Through comprehensive results analysis, we identify key challenges in producing high-quality

SBOMs both manually and with tools, and suggest directions to address these challenges.

The main contributions of our work are:

- A manually curated ground truth dataset of SPDX SBOMs for Python repositories from GitHub
- A taxonomy of dependency detection failure cases in GitHub's dependency graph tool
- A taxonomy of complications in license determination

II. Background

A. Software Bills of Materials

Recently, Software Bills of Materials (SBOMs) have grown in popularity as a means of providing a window into the composition of software and facilitating software analysis. Often analogized as nutrition labels for software, SBOMs provide visibility into the software supply chain, allowing interested parties to analyze software through the lens of its dependencies.

A spate of recent supply chain attacks, including the SolarWinds attack [20] and a recent attack on npm packages [21], have highlighted the need for improved visibility into the software supply chain. When a vulnerability is located or a malicious software package is discovered, it can be critical for maintainers to be able to quickly identify whether they are affected by the event. The complexity of modern software can mean that making this determination can be difficult for software maintainers, let alone for individual users or organizations who seek to know if they are at risk through their use of the software. SBOMs' ability to provide a standardized view of the composition of a piece of software significantly facilitates such analysis.

The current focus on SBOMs as a solution to this problem has emerged in part due to regulatory pressure for their creation and dissemination. In the United States, the Executive Order on Improving the Nation's Cybersecurity [6] included guidance requiring SBOMs for software sold to the government—though this has been somewhat curbed by more recent developments [22, 23].

Similarly, in Europe, the EU Cyber Resilience Act (CRA) [7] also includes requirements for SBOMs, demonstrating continued interest in the use of SBOMs for software analysis.

SBOMs also facilitate another type of software analysis which focuses on licensing. Software licensing tasks are known to be difficult for developers [24]. However, as the practitioners who work most closely with software components made available under licenses, developers often bear responsibility for making decisions about software licenses and participate in resolving licensing issues [25].

By collecting information about software components in a standardized fashion, SBOMs can facilitate software licensing analysis through their cataloging of the licensing status of a software project and its dependencies. In software that has many dependencies, even identifying which licensing obligations are present in one's software can pose a challenge. SBOMs and SBOM tools can collect this information for developers, who can employ it to understand these obligations as well as to identify any conflicts between them.

B. SBOMs in the SPDX format

GitHub generates SBOMs in the SPDX format, which has three main parts: (1) the document creation section, acting as a header, (2) the package section, listing all the dependencies of a software package, and (3) the relationships section, describing relationships between the components.

The document creation section describes the SPDX document itself. Fields within the document creation information section include:

- SPDXVersion: the SPDX standards version of the document.
- DataLicense: the license under which the SPDX version is distributed.
- SPDXID: the local portion of the unique SPDX identifier referring to the document itself.
- DocumentName: user-defined name of the SPDX document.
- DocumentNamespace: a unique uniform resource identifier (URI) for the SPDX document.
- Creator: the person, organization, or tool that created the SPDX document.
- Created: the timestamp of when the SPDX document was created.

The package section, which we focus on in this study, lists the project dependencies and relevant information about them. The first item in this list is the project being cataloged by the SBOM, which we refer to as the primary package [26]. All subsequent items are packages or dependent components of the primary package. The following fields apply to each individual package in this section (those in bold are relevant to our study):

- **PackageName**: the human readable name for a package as defined by its creator.
- SPDXID: the unique addressable identifier for the package within the SPDX document.
- **PackageVersion**: the version of the package defined by its creator.
- PackageSupplier: the person, organization, or tool that created the package.
- **PackageDownloadLocation**: a unique uniform resource locator (URL) or version control system location where the package can be obtained.
- FilesAnalyzed: a boolean value that describes whether or not a files section is included for the package.
- **LicenseDeclared**: the licenses that have been declared by the authors of the package.
- **LicenseConcluded**: the licenses the SPDX file creator has concluded as governing the package if the governing license cannot be determined.
- CheckSum: an independently reproducible mechanism that allows for the unique identification of a package in the SPDX document. Included as a tampering alert mechanism.

Finally, the relationships section contains information on how SPDX SBOM elements are associated with one another using internal identifiers. For example, the relationship between the primary package and the document creation information is that the document creation information DESCRIBES the primary package. Other examples of relationship identifiers include CONTAINS, DEPENDS_ON, RUNTIME_DEPENDENCY_OF, GENERATES, and others.

C. Open source software licensing

Software in the United States and around the world is protected under copyright law [27]. The copyright holder for a piece of software retains the rights to distribute, reproduce, and create derivative works of that piece of software. Additionally, they can also extend those permissions to others, allowing them to exercise those same rights [28]. This typically occurs by granting a license to the interested party. While these licenses can give third-parties permission to use, modify, or redistribute software, they can also place restrictions and limitations on those permissions [29]. Any usage of the software that falls outside of these explicitly stated permissions could constitute copyright infringement.

Developers of open source projects tend to use off-the-shelf licenses for their projects. These licenses fall into two broad categories: permissive licenses, and restrictive licenses (sometimes referred to as "copyleft" or "viral" licenses). Permissive licenses (*e.g.*, MIT, BSD, Apache 2.0) impose few, if any, limitations on software reuse, typically only requiring attribution notice. Restrictive licenses (*e.g.*, GPL, LGPL), however, place distribution limitations on derived or modified works, namely requiring that the full source code of derived works be made available under the same license. Failure to comply with the terms of all relevant licenses can have real world consequences and lead to financial [30, 31], reputational [32, 33], or legal consequences [34, 35]. Unfortunately, licensing issues of many different forms can often find their way into software, such as missing or incomplete licensing information [36] or inconsistencies between a repository's declared license and those of its dependencies [37]. Developers might also make licensing changes as software evolves [38], though inappropriate changes which conflict with ancestors can cause additional issues [39].

Recent work has begun to investigate machine learning-based means of addressing licensing problems [40]. LiDetector [41] utilizes a learning-based method to read and interpret the text of software licenses, allowing for the identification of potential licensing conflicts. However, as new approaches emerge, there exists an unmet demand for both benchmarks on which to test tools and assessments of existing tools' behavior.

III. Study Design

The goal of our study is to assess the completeness and accuracy of SBOMs generated by GitHub, highlight the challenges faced during SBOM creation, and offer potential solutions to address these challenges. This study aims to answer three research questions (RQs):

RQ_1: *How complete and accurate are GitHub SBOMs?* Here, we seek to determine the ability of GitHub's SBOM tool to produce SBOMs that include all of the project's actual dependencies and accurately provide information regarding those dependencies.

RQ_2: *What causes quality issues in GitHub SBOMs?* Given a set of SBOMs from GitHub, this question seeks to identify and understand cases in which GitHub failed to collect all relevant dependencies, or where it incorrectly collects components which are not dependencies.

RQ_3: *What are the challenges of creating quality SBOMs?* Finally, having created SBOMs ourselves and assessed SBOMs created by GitHub, we seek to enumerate factors which can make it difficult to produce or acquire a high-quality, correct SBOM.

A. Collecting Candidate Python Projects

We gathered a dataset of SPDX SBOMs and manifest files for Python projects hosted on GitHub. We show the process used to do so in Table I. In a Python project, a manifest file is a list that records third-party dependencies used to develop or execute the project. Typical Python manifest files include `requirements.txt`, `setup.py`, `pipfile.lock`, and `poetry.lock`.

The GitHub Search (GHS) tool, developed by *Dabic et al.* [42], was used to gather a list of GitHub repositories suitable for analysis. GHS is a search tool that continuously mines GitHub to index projects and their metadata. The tool considers 25 characteristics from GitHub repositories (number of stars, last commit, *etc.*). The GHS tool has mined over 1,466,679 GitHub repositories, which includes 286,598 Python projects. We filtered for active and popular Python repositories, with a minimum of 500 stars, a detected software license, and the last commit made within a year of the date we collected the data (May 5th, 2024). This resulted in a list of 3,511 eligible repositories. After filtering to repositories with a GitHub SBOM through GitHub's dependency tracker tool (generated using GitHub's SBOM API [13], with source verifiable through the "creators" metadata field in the SBOM), the process resulted in set of 2,611 Python repositories in which both the SBOM and manifest files were present. We note that these SBOM documents are retrieved directly from GitHub's API, and are not part of the main file structure of the repository. While this allows us to verify the source of the SBOMs, the documents retrieved may be fetched from a previously-generated version [13], which could result in stale information. We discuss the implications of this in Section IV-B.

B. Manually Creating Ground Truth SBOMs

1) Project selection and SBOM fields: We used stratified sampling to select 50 projects from the 2,611 Python projects to create a manageable subset that could be rigorously reviewed. First, the dataset was divided into quartiles based on the number of dependencies detected by Github's SBOM generator. Half of the projects contained 10 to 38 dependencies. The mean number of dependencies was 86.8, while the mode was 10.

Total GHS Repositories	Python Repositories	Min 500 stars, licensed, last commit within 1 year	Has GitHub SBOM	Stratified Sample
1,466,679	286,598	3,511	2,611	50

TABLE I
GROUND TRUTH PROJECT IDENTIFICATION

From the quartiles, 50 projects were randomly selected following this distribution: 40% from the first quartile, 30% from the second quartile, 20% from the third quartile, and 10% from the fourth quartile. We weighted our analysis this way to account for cases where GitHub's analysis failed to identify dependencies, resulting in shorter SBOMs. As such, considering the total number of dependencies that would need to be reviewed across the total of all repositories examined, focusing more heavily on repositories with fewer identified dependencies allowed us to both analyze more repositories in total while also affording the opportunity to more thoroughly examine the ways in which GitHub's tool might fail. To maintain feasibility of manual analysis, after defining the quartiles, we set an upper limit of 50 dependencies per repository, as there was no reason to believe that a thoroughly documented repository with many dependencies would experience more or unique tool failures.

The SPDX SBOM fields we targeted include the *PackageName*, *PackageDownloadLocation*, and *LicenseDeclared* of the primary package, and the *PackageName*, *PackageVersion*, and *LicenseConcluded* of the dependencies of the primary package. These fields were selected because they contain relevant and actionable information regarding the dependencies of a repository. Fields that contain unique identifiers for objects within the SBOM, such as a package's SPDXID and internal SPDX checksum, are transient and will change depending on the tool used to generate the SBOM. The *PackageDownloadLocation* field for dependencies was not considered since the GitHub SBOMs do not include it.

2) Manual Review: The 50 projects selected for manual review were then divided among three authors, with each author reviewing between 16 and 17 projects. Each author collected dependency information required to produce complete and accurate SBOMs for their projects with the fields previously described. This process involved cross-referencing files contained within the repository (`requirements.txt`, `setup.py`, *etc.*), the output of scanning tools, and the GitHub Dependency Graph. We used *Pipreqs* [43], which statically scans a Python package for import statements and attempts to generate a complete `requirements.txt` to capture dependencies that may have been missed by the original developers. We also used *requirements-detector* [44], which scans and aggregates information from the various dependency files that Python supports (`requirements.txt`, `setup.py`, `pip.lock`, *etc.*). Neither of these tools is perfect, and they were used along with manual verification and validation to provide a more complete picture of the software's dependencies.

For each potential dependency identified by the aforementioned tools, the author marked whether it should be included or excluded from the final SBOM. We only included production dependencies needed for regular software functioning, thus excluding all dependencies that were related strictly to testing, documentation, or continuous integration, as well as dependencies that only appeared in example files and dependencies specified as optional. These exclusions also allow for a fair comparison with GitHub, which only uses the direct "DEPENDS_ON" SPDX relationship value, forgoing relationship types corresponding with optional dependencies, testing dependencies, *etc.* For the remaining dependencies, the author noted whether they were "sure" or "unsure" that the dependency should be included. We established guidelines for the reviewing authors: if an import statement referencing a module was present in a repository without that module being listed in a manifest, we included that module and labeled the entry as "unsure;" when it was unclear if one of our constraints was violated, *e.g.*, whether the usage of the dependency was in an example, we also included it as "unsure." The final labeling was left to the reviewing authors' judgment, taking the totality of the repository into consideration. After manual review, each dependency was checked by a second author. In the event of a disagreement on labeling, the authors discussed and made a final determination, resulting in the full list of packages utilized by the 50 projects.

With dependencies now identified, the authors analyzed dependency licenses as declared on both PyPI and GitHub (if available), noting any discrepancies. If one source provided more specific licensing information (*e.g.*, BSD vs BSD-3-Clause), we included the more specific version in our SBOM. If there was an explicit conflict (*e.g.*, Apache vs MIT or BSD-2-Clause vs BSD-3-Clause) or no licensing information was provided, we entered No-Assertion for the license field (*i.e.*, the default "unknown" value in SPDX SBOMs).

C. Comparing Ground Truth Against GitHub's SBOMs

We used *precision* and *recall* to measure the 50 GitHub-generated SBOMs' completeness and correctness by comparing field names and values between the ground truth SBOM and the corresponding GitHub SBOM. Instances in which the GitHub SBOM field (both name and value) matched the ground truth were recorded as true positives (TP). False positives (FP) occurred when the GitHub SBOM contained a field name+value different from the ground truth. False negatives (FN) were recorded in cases where the GitHub SBOM omitted a field+value that appeared in the ground truth.

We computed precision ($TP/(TP + FP)$) and recall ($TP/(TP+FN)$) for all fields in aggregate across the SBOM to obtain an overall quality measurement of GitHub SBOMs. They were also computed for individual fields to identify those for which GitHub's generator performed well or struggled. We also computed the metrics for fields corresponding to the primary package and to all its dependencies in aggregate.

Field	Precision	Recall	TP	FP	FN
PackageName	0.80	0.85	527	130	96
PackageVersion	0.76	0.95	286	92	16
LicenseConcluded	0.54	0.23	123	104	422

TABLE II
SBOM FIELD QUALITY FOR THE PRIMARY PACKAGE'S DEPENDENCIES.

To more accurately identify situations in which GitHub's tool fails, we took a statistically significant sample (95% confidence level with a 5% error margin) of dependency name FPs and FNs, *i.e.*, modules that were wrongly included/excluded from the GitHub SBOM, respectively. Of 226 such cases, we took a random sample, proportionally stratified across all repositories, for an analysis of 143 cases divided among 3 authors. The cause of each case was labeled via *open coding* [45].

IV. RESULTS

A. $\boldsymbol{RQ}_1$: SBOM completeness and accuracy

The aggregate precision (median of 0.88) and recall (median of 0.6) across projects are found in Figure 1. While this high precision demonstrates that the tool can reasonably capture dependency information in many cases, the low recall and high variability among repositories illustrates a wide range of SBOM quality. The lowest-quality SBOM (0.23 precision, 0.32 recall) is for the project `pytorch/text` [46], which has eight dependencies. The highest-quality SBOM (0.88 precision, 1.0 recall) is for the project `p0n1/epub_to_audiobook` [47], which also contains eight dependencies. The quality difference between these two projects stems from the incorrect inclusion of optional, documentation, and testing dependencies in the generated SBOM for `pytorch/text`. However, given that we deliberately oversampled low-dependency projects (Section III-B), we note that these results may not reflect the GitHub generator's performance on larger projects with deeper dependency trees.

The quality results for individual SBOM fields for primary package dependencies are shown in Table II. The results suggest that the GitHub SBOM tool performs the best at capturing dependency names while performing worse for dependency versions, and significantly worse for dependency licenses. The high number of false negatives for the license field demonstrates that SBOMs miss important license information.

As for field results for the primary package, we found a perfect quality for `PackageName` and `PackageDownloadLocation`, as expected (since this information is present in the GitHub repository), and 0.93 precision and 0.89 recall for `LicenseDeclared`, which indicates GitHub's SBOM tool struggles to identify the (correct) license on the GitHub repository. Without much information in GitHub's official documentation, we conjecture GitHub may rely on outdated/removed repository information for generating SBOMs (more details below).

B. $\boldsymbol{RQ}_2$: Observed causes of deficient SBOMs

Our analysis of why modules were incorrectly included or excluded based on a statistically significant sample of 143 dependency name FPs and FNs yielded several key insights into the ways GitHub's tool may fail. These reasons are taxonomized in Figure 2 – see our replication package for full definitions, examples, and annotated instances [48].

The largest contributor to components being mistakenly included in the GitHub SBOM was *Overinclusive Requirements Mining* (48), *i.e.*, the inclusion of components defined in auxiliary manifests such as ``docs/requirements.txt'' and similar files. Such dependencies, excluded from our SBOMs, are included by GitHub without assigning appropriate relationship types.

In contrast, relatively few entries were the result of *Incorrect Detection* as dependencies by the tool (4). We found three dependencies that were included based on their presence in *Obsolete / Removed Manifest* files that had been deleted years prior. One repository, FrEIA [49], had previously included an auxiliary requirements file for documentation-related dependencies. Although this file was deleted in 2018, GitHub's Dependency Graph still registered its last contents as dependencies and provided a link to the nonexistent file. Similarly, the Dependency Graph of another repository, skll [50], still reports modules from a file called `requirements_rtd.txt` despite the fact that the file was deleted in 2019. As of May 26, 2026 (two years after initial data collection), dependencies from these no-longer-existent files were still being reported by the dependency graph. This suggests that GitHub's dependency graph may not update properly when files are deleted, but further investigation is required to understand this behavior. We, however, were unable to replicate this strange error in a controlled repository.

Thirty-one errors resulted from *Labeling Problems* with dependency information. Of these, 29 cases were *Unnecessary Dependencies Included by the Developer* in the main project manifest alongside those required for normal functioning. This is consistent with the known problem of bloated dependencies [51], in which included dependencies may not be necessary to build and run the software. Beyond this, we observed two instances with an *Incorrect Name* entered into the `requirements.txt` file instead of the appropriate PyPI name (*e.g.*, `bs4` instead of `beautifulsoup4`). This is particularly problematic when the incorrect name is also a valid downloadable module. (In this case, bs4 is a dummy module that prevents namesquatting attacks.)

The vast majority of FNs were dependencies being *Imported (in the source code) but not included in the manifest* (45). We also encountered one example where the source code for a dependency was *Directly Embedded* into the repository, without documentation (excluding the copied license information in the files). Beyond this, we observed 14 instances of a dependency not being detected due to being *Specified in a README file* (9) or through non-trivial *Unsupported Requirements Syntax* in the manifest which GitHub's tool failed to detect (5), such as syntax specifying that a dependency be installed directly from a GitHub repository or providing an alternative download source URL.

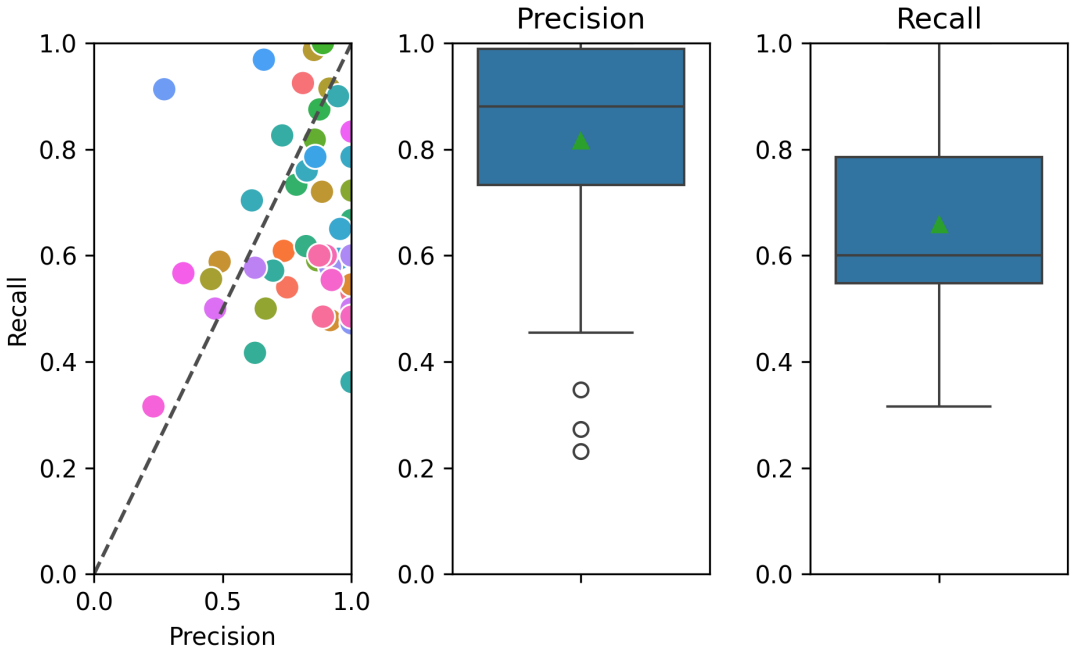

Fig. 1. Overall SBOM Quality Across Projects

C. RQ_3: Challenges in creating SBOMs

Finally, using our experience in manually creating SBOMs as well as our assessment of GitHub-generated SBOMs, we present an analysis of the situations we identified which present challenges for creating high-quality, accurate SBOMs.

1) Determining dependencies and versions: Various SBOM tools rely on information explicitly provided in manifest files (`requirements.txt`, `setup.py`, *etc.*) [52]. Poor documentation, which commonly afflicts software [53–55], thus hinders SBOM creation, such as when modules are declared without being used in the source code or used without suitable documentation. Static analysis tools can detect such modules, but without project familiarity, it is hard to make concrete determinations about their usage (*e.g.*, determining whether a component is a transitive dependency, an essential runtime component, or an obsolete component). Tools can also struggle when a package's name on PyPI is distinct from its import name (*e.g.*, the module *opencv-python* and its associated import *cv2*). Modules might also be documented outside of accepted manifest files, such as in the project's README file, which, while being human-readable, is nonetheless missed by GitHub's Dependency Graph. If module version information is not supplied, recovering it may require difficult, detailed execution path analysis.

2) Determining project licenses: Nearly all dependencies encountered during our analysis were hosted by and expected to be downloaded from PyPI, but the quality of licensing information provided by project developers on PyPI was inconsistent. The location of the most precise licensing information varied, sometimes occurring in the sidebar along with other meta-data, sometimes found in the tags at the top/bottom of the page, and sometimes just included in the module textual description. For 78 packages (21.4%), the information available on PyPI was insufficient to determine the appropriate license. This, in part, resulted from developers not utilizing standardized SPDX license identifiers [56]. It was not uncommon for a broad license category to be provided, such as BSD, but then lack specifics on which version applied. Also, when exceptions were noted, it was not uncommon to see text like: "LGPL with exceptions," without indication of the specific exceptions. Additionally, sometimes deprecated license identifiers or variant names for the same license were used.

Fortunately, most PyPI pages linked to a public GitHub repository with the source code and associated license. However, this too introduced new challenges. In our manual analysis of 364 modules, we found that the licensing information on PyPI and GitHub disagreed in 97 cases (27%). Furthermore, while GitHub's license detection tool was helpful in determining some licenses, it was unable to automatically: 1) differentiate between the various GPL "or later" and "only" versions, 2) detect exceptions in license text, 3) identify niche variants such as BSD-2-Clause-with-views, 4) detect any license if the format had been slightly altered, *e.g.*, if a header was missing or only a link to the license text was provided, and 5) determine whether the relationship between two licenses in a multi-licensing scheme was an AND or an OR. This multi-licensing information was also ambiguously represented on PyPI, often with only a comma separating licenses.

We report the most common licenses present in our dataset in Figure 3. As shown, the projects in our dataset were overwhelmingly permissively licensed: MIT (142), Apache-2.0 (80), BSD-3-Clause (66). All other licensing schemes

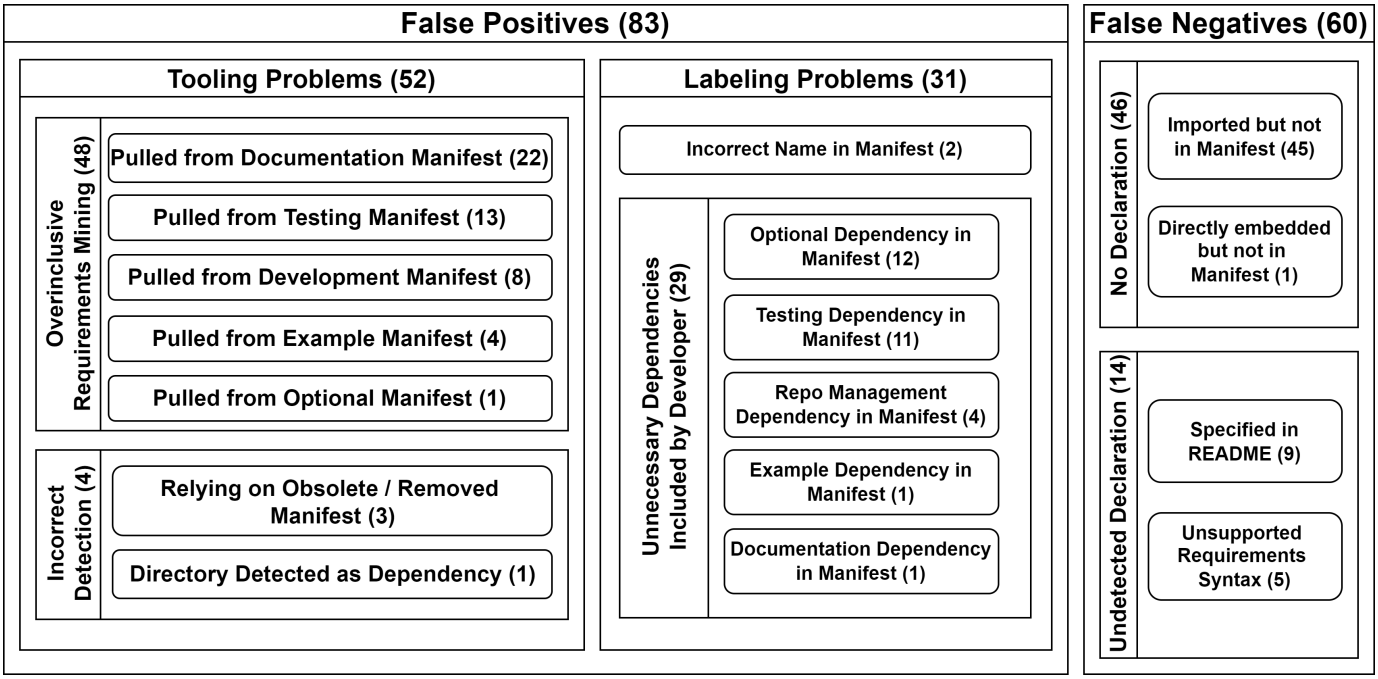

Fig. 2. Taxonomy of Component Name FP/FN Causes

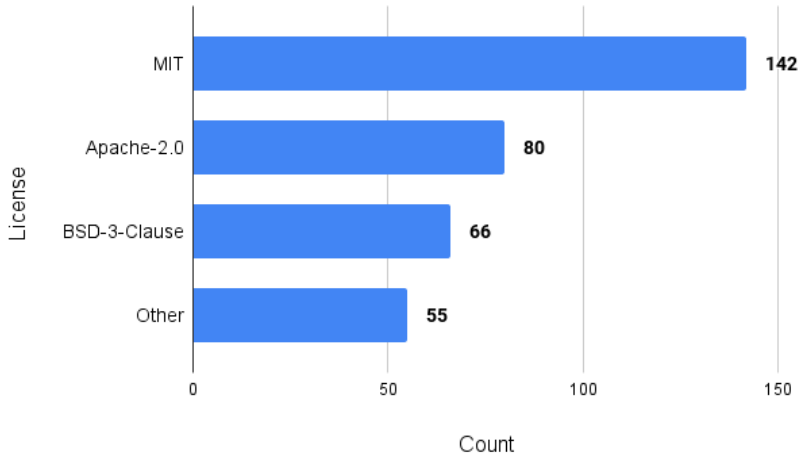

Fig. 3. Most Common Component Licenses in our Dataset

had four or fewer occurrences. There were 21 modules for which we were unable to determine a license and thus labeled No-Assertion, detailed in Figure 4. Many of the problems arose from ambiguous license versioning (6) or multi-licensing (4), while some repositories had non-standard licensing (6), conflicting licensing information across different sources (7), or no declared license at the source (3). These findings are consistent with previous work on licensing inconsistencies, which notes how licensing can change over time [39].

V. Discussion

Here, we discuss our findings and the results of our analysis, and offer actionable recommendations to improve SBOM tooling in the future.

A. The need for validating dependency detections.

By its nature, the GitHub dependency graph trusts package files. If a dependency is listed in package files, it is included in the output regardless of whether it is used in the actual code base. This process will also include dependencies and versions that do not actually exist on the Python Package Registry. There appears to be no validation beyond what the developer supplied in the codebase. Particularly for tools that rely on documentation supplied by the author, there is more work to be done to validate that information. Specifically, this includes cross-referencing packages to verify that they

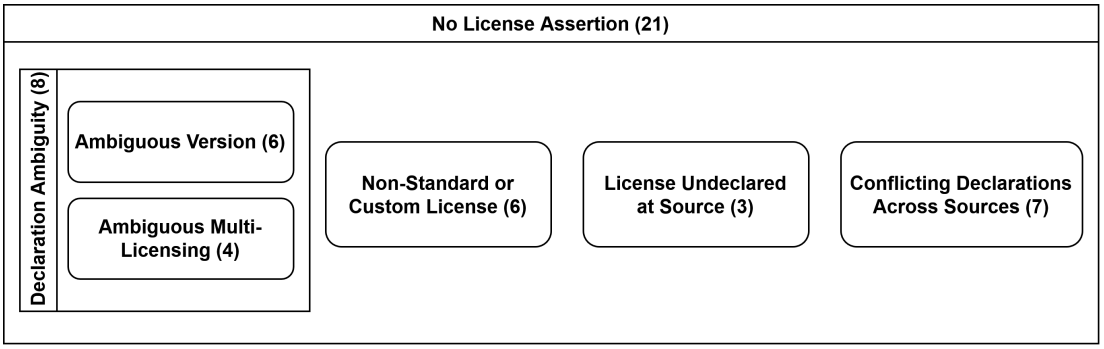

Fig. 4. Taxonomy of Reasons for No-Assertion

exist in the upstream registry and that they are actually used in the software product. Including dependencies that do not meet these criteria in the SBOM is unhelpful and, indeed, counterproductive.

B. Agents for license detection.

Given that license compliance is an important non-functional requirement of software development and that even the act of determining dependency licenses can be an arduous and time consuming task, it seems that improved solutions for software licensing problems are necessary. Promising modern techniques seem to have the appropriate capabilities for compiling this information for review, and indeed, recent approaches such as LiDetector [41] utilize learning-based approaches to address software licensing issues. However, our results showed that licensing information could be found on a host of different platforms and in a variety of formats. As such, this may be a problem suitable for agentic automation. AI agents may be able to search for these licenses, read them, and identify them quickly, not only by analyzing a given repository, but also by examining different relevant platforms to retrieve pertinent information from disparate sources.

C. Clear labeling for component relationships.

The most common cause of components improperly being included in SBOMs that we could identify was the inclusion of components from other locations in the repository which were indicative of other types of relationships, such as from other manifests for documentation or testing. Our results did not indicate that the GitHub SBOM tool could distinguish between these different types of component relationships. This is significant, as the way in which a component is used (such as for testing) can have implications regarding its licensing status. Regardless of tooling, a means of formalizing and/or recognizing different types of component relationships in software is critical for this purpose. Indeed, SPDX does have a field to specify relationships, but it went severely underutilized according to the SBOMs we examined.

D. Recommendations for tool developers.

Practitioners who develop SBOM tools should take heed of the identified problems in SBOM generation when considering how to improve current tooling. However, fundamentally, SBOMs are intended to be machine-readable documents, perhaps to the detriment of their human-readability. As such, it can be difficult for developers to understand what an SBOM is telling them. This may also induce difficulties in identifying cases where the SBOM generator produced something which was incorrect or faulty. However, we have demonstrated that many such cases exist. When such issues are identified, developers should have an option to rectify them. No tool is perfect, and SBOMs are difficult to create or edit manually. Tool developers could improve this situation by providing accessible means for practitioners to update or correct SBOMs after they have been generated.

In our analysis, we noted that the component relationship field in the SBOM was neglected. However, as noted above, this field has powerful implications for supply chain components: certain dependencies are only included optionally, for testing, or for development. These different relationships may have different licensing implications, such as testing dependencies which might not be distributed with the final software and thusly do not implicate licensing clauses regarding software redistribution. Furthermore, given that SBOMs are used heavily in vulnerability management, it is important to limit the number of false positives. In other words, if a vulnerability is discovered in a package, it is important for those who depend on that package to quickly identify whether they are at risk, and incorrectly assessing that you depend on the vulnerable component could have adverse consequences. In short, dependencies that are not present in the final distribution should either be absent from the final SBOM or clearly labeled. Tools should provide inherent support for different types of component relationships to mitigate these issues.

Many libraries also include example code within their repositories that demonstrates how their project works. Often these examples may include requirements files or import third-party libraries. Such examples should similarly be excluded from generated SBOMs if they are not truly a component of the software itself.

Package managers can facilitate complex inter-component relationships, but this complexity might not always be captured by tools. Given the variation across implementations even

within a given ecosystem, it is important to be aware of a given tool's weaknesses and potential points of failure. Tool developers should consider disclosing known failure points to advise users of any applicable limitations.

E. Recommendations for tool users.

Those who use SBOM tools should be mindful that tool outputs will not always be perfect. If one intends to use tooling, such as the GitHub dependency graph, they should be aware of what the tool takes as input and ensure that the inputs they provide are high-quality. In this case, that entails making sure that one's software documentation is up-to-date and accurate to facilitate accurate SBOM generation and software analysis.

As part of this, developers can ensure that example, testing, or development dependencies are not being included in their final SBOMs without appropriate documentation.

Furthermore, they should ensure that they are following best practices and declaring dependencies in standard ways, such as though package managers (*e.g.*, pip, poetry, conda). Developers should not simply list dependencies in a README file, where current tools may not be able to find them as easily.

VI. Related Work

We complement the current body of SBOM work by comparing GitHub-generated SBOMs with *manually curated SBOMs*, rigorously analyzing the reasons for low-quality SBOMs generated by GitHub's SBOM tool, for 50 Python projects of different sizes and domains. We also discuss challenges and potential solutions in creating SBOMs.

A. SBOM studies and tool evaluation

Recent studies [3, 14–17] investigated adoption, benefits, and challenges for SBOM production and use in practice, via surveys and interviews with practitioners (*e.g.*, industry and open-source developers, managers, and organizations). Practitioners perceive tooling as an essential requirement for SBOM adoption [14, 15], as it is necessary for handling large and complex software systems. To support SBOM management, many vendors have developed tools that generate, parse, query, and convert SBOMs [10] (*e.g.*, [57–60]). Practitioners use many of these tools, without any subset used consistently [15]. Inaccurate/incomplete SBOMs are a commonly reported challenge [15, 61] which force manual SBOM inspection and correction [14]. This is primarily due to unreliable tooling and difficulty in verifying accuracy and correctness [14–17].

B. SBOM tooling

A number of studies have empirically assessed the reliability of existing SBOM generation tools for various languages and in various contexts [4, 10, 18, 19, 52, 62–66]. For example, Balliu *et al.* [18] compared six tools for Java systems and found that they capture different dependencies, missing much of the Maven dependency tree. The sbomqs [67] tool provides a quality score for SBOMs, but without evaluation in the literature, it is yet unclear how accurate or useful this metric is. Rabbi *et al.* [19] evaluated the accuracy of four SBOM tools for 50 JavaScript projects, compared to the NPM tree of dependencies. Significant performance variations across tools were found, with cnn being superior in identifying component names and versions and the other tools having major dependency omissions. Yu *et al.* [52] compared the SBOMs generated by four tools, including GitHub's, for 535 Python projects and 7k+ projects written in other languages, and found that the dependencies listed in the SBOMs have little overlap, indicating low consistency across tools. Outputs of GitHub's dependency graph, which supports its SBOM generator [11], do not always match dependencies listed in manifest files due to package mismatches or outdated dependency information [68]. Where this previous work examines GitHub's dependency graph directly, it does not assess the SBOM generation tool or identify where SBOM generation may struggle.

C. SBOM Datasets

There have also been several other recent efforts to construct datasets that can support SBOM research and tooling evaluation. *Ohm et al.* created a large dataset of SPDX SBOMs for 100,000 randomly sampled public GitHub repositories [69] by using project dependency graphs. While this dataset is impressive in scope, there was no manual validation of the SBOM contained therein. Shortcomings in GitHub's dependency graph tooling are left unexplored and unaddressed.

Another work by *Gamage et al.* compiled an SBOM dataset mined from Maven Central [70]. Relatedly, *Kishimoto et al.* created a dataset of 46 SBOMs describing real-world Java projects for the purpose of evaluating SBOM consumption tools [71]. SBOM generation issues can vary dramatically across languages and tools [15], and so the issues encountered in producing SBOMs in the Java-based, Maven ecosystem are likely distinct from those we observe in the less structured Python community.

Finally, in their work on SBOM quality, *Torres et al.* note that there are "currently no standard SBOM datasets to be used by empirical software researchers," later concluding that "it is hard to research SBOMs when there are so few SBOMs made publicly available and suitable for research" [72].

VII. Threats to Validity

A. External Validity

Our study focuses on a sample of Python repositories and SBOMs created with GitHub's SBOM generator, and thus may not generalize to projects in other languages or other SBOM generators. A sample size of 50 SBOMs may not capture all possible SBOM generation problems, but by weighting our analysis towards projects with fewer detected components, we capture more repositories where GitHub's tool struggled.

B. Internal Validity

The nature of our data collection required analyzing software repositories to determine which components they used, introducing a risk of failing to identify dependencies or introducing errors during data collections. We mitigated this risk by having the values of each hand-created SBOM verified by an additional

author and discussing any disagreements between the two authors' results.

C. Construct Validity

Our approach to identifying issues in GitHub SBOMs and the GitHub SBOM generator required analysis of unfamiliar codebases to formulate ground truth data. Without an intimate understanding of the target repositories, it is possible that the annotators labeled dependencies in ways which do not align with their actual use. We mitigate this risk by applying standards to the means by which components were labeled and discussing disagreements, though a deeper analysis would require participation from developers who were more familiar with the target software projects.

VIII. Conclusion

As SBOMs continue to spread amidst high-profile supply chain attacks and growing regulatory acceptance, we provide resources to assess SBOM tooling and analyze an SBOM generator built into a highly popular system, identifying problem areas and failure cases.

We provide a novel, publicly-available, extendable dataset of 50 manually verified SBOMs for 50 Python repositories [48], enabling analysis of SBOM generation tools against a ground truth in an SBOM landscape where correct, verified SBOMs are rare. We utilize this dataset to perform an analysis on GitHub's built-in SBOM creation functionality. The results show that GitHub's tool sometimes misses information about dependencies, performing especially poorly at determining a dependency's license. We identify 17 key reasons that SBOMs generated by GitHub can be deficient and present these in a novel taxonomy. We apply our experience in manually creating SBOMs to identify challenges facing SBOM creation.

Considering our findings, we present a number of recommendations for those who use SBOM tools and those who make them, including providing options to edit SBOMs, validating inputs and dependency detections, and potential for future improvements, such as though agentic systems.

References

[1] NTIA, "SBOM at a glance," https://tinyurl.com/txyvbhfu, National Telecommunications and Information Administration, 2021.

[2] ——, "Roles and Benefits for SBOM Across the Supply Chain," https://ntia.gov/files/ntia/publications/ntia_sbom_use_cases_roles_benefits-nov2019.pdf, 2019.

[3] S. Hendrick, "Software bill of materials (SBOM) and cybersecurity readiness," https://tinyurl.com/293v3xte, The Linux Foundation, 2022.

[4] S. Nocera, S. Romano, M. Di Penta, R. Francese, and G. Scanniello, "Software Bill of Materials Adoption: A Mining Study from GitHub," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 39–49, DOI: 10.1109/ICSME58846.2023.00016.

[5] N. Sheppard, "SBOMs (Software Bill of Materials): Why Do They Matter?" 3 2023. [Online]. Available: https://www.leanix.net/en/blog/sboms-matter

[6] "Executive order 14028," 2021. [Online]. Available: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[7] European Parliament and Council of the European Union, "Regulation (EU) 2024/2847 of the European Parliament and of the Council on horizontal cybersecurity requirements for products with digital elements (Cyber Resilience Act)," https://eur-lex.europa.eu/eli/reg/2024/2847/oj, 2024, entered into force 2024; SBOM obligations apply from December 2027.

[8] "SPDX specifications," The Linux Foundation, 2023. [Online]. Available: https://spdx.dev/specifications/

[9] "Cyclonedx specifications," 2023. [Online]. Available: https://github.com/CycloneDX/specification

[10] M. Mirakhorli, D. Garcia, S. Dillon, K. Laporte, M. Morrison, H. Lu, V. Koscinski, and C. Enoch, "A landscape study of open source and proprietary tools for software bill of materials (sbom)," *arXiv preprint arXiv:2402.11151*, 2024, DOI: 10.48550/arXiv.2402.11151.

[11] "Introducing self-service SBOMs," https://tinyurl.com/mt9jwcdx, 3 2023.

[12] "About the dependency graph," https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.

[13] "REST API endpoints for software bill of materials (SBOM)," 11 2022. [Online]. Available: https://docs.github.com/en/rest/dependency-graph/sboms

[14] B. Kloeg, A. Y. Ding, S. Pellegrom, and Y. Zhauniarovich, "Charting the Path to SBOM Adoption: A Business Stakeholder-Centric Approach," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1770–1783, DOI: 10.1145/3634737.3637659.

[15] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, and D. Poshyvanyk, "BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13, DOI: 10.1145/3597503.3623347.

[16] N. Zahan, E. Lin, M. Tamanna, W. Enck, and L. Williams, "Software Bills of Materials Are Required. Are We There Yet?" *IEEE Security & Privacy*, vol. 21, no. 2, pp. 82–88, 2023, DOI: 10.1109/MSEC.2023.3237100.

[17] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2630–2642, DOI: 10.1109/ICSE48619.2023.00219.

[18] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger, "Challenges of producing software bill of materials for java," *IEEE Security & Privacy*, 2023, DOI: 10.1109/MSEC.2023.3302956.

[19] M. F. Rabbi, A. I. Champa, C. Nachuma, and M. F. Zibran, "Sbom generation tools under microscope: A focus on the npm ecosystem," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1233–1241, DOI: 10.1145/3605098.3635927.

[20] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. B. Michael, "Perspectives on the SolarWinds Incident," *IEEE Security & Privacy*, vol. 19, no. 2, pp. 7–13, 2021.

[21] L. Franceschi-Bicchierai, "Hackers have compromised dozens of popular open source packages in an ongoing supply-chain attack," https://techcrunch.com/2026/05/19/hackers-have-compromised-dozens-of-popular-open-source-packages-in-an-ongoing-supply-chain-attack/, 2026.

[22] "Sustaining select efforts to strengthen the nation's cybersecurity and amending executive order 13694 and executive order 14144," 2025. [Online]. Available: https://www.whitehouse.gov/presidential-actions/2025/06/sustaining-select-efforts-to-strengthen-the-nations-cybersecurity-and-amending-executive-order-13694-and-executive-order-14144/

[23] "Sustaining Select Efforts To Strengthen the Nation's Cybersecurity and Amending Executive Order 13694 and Executive Order 14144 ," https://www.federalregister.gov/documents/2025/06/11/2025-10804/sustaining-select-efforts-to-strengthen-the-nations-cybersecurity-and-amending-executive-order-13694, 2026.

[24] D. A. Almeida, G. C. Murphy, G. Wilson, and M. Hoye, "Investigating whether and how software developers understand open source software licensing," *Empirical Software Engineering*, vol. 24, no. 1, pp. 211–239, 2019, DOI: 10.1007/s10664-018-9614-9.

[25] N. Wintersgill, T. Stalnaker, L. A. Heymann, O. Chaparro, and D. Poshyvanyk, ""the law doesn't work like a computer": Exploring software licensing issues faced by legal practitioners," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 882–905, 2024, DOI: 10.1145/3643766.

[26] S. Winslow, "Spdx and ntia minimum elements for sbom howto," 2023. [Online]. Available: https://spdx.github.io/spdx-ntia-sbom-howto/#_3_understanding_an_ntia_minimum_elements_spdx_sbom

[27] "Copyright registration of computer programs," https://www.copyright.gov/circs/circ61.pdf, 3 2021, accessed: 2023-25-09.

[28] "U.s. code title 17, section 106," https://www.govinfo.gov/app/details/USCODE-2021-title17/USCODE-2021-title17-chap1-sec106/summary, 2021, accessed: 2023-25-09.

[29] H. Meeker, *Open source for business: a practical guide to open source software licensing*. CreateSpace, 2017.

[30] A. Vance, "The defenders of free software," https://www.nytimes.com/2010/09/26/business/26ping.html, 9 2010, accessed: 2023-14-09.

[31] G. Gross, "Open-source legal group strikes again on busybox, suing verizon," https://www.computerworld.com/article/2537947/open-source-legal-group-strikes-again-on-busybox--suing-verizon.html, 12 2007, accessed: 2023-14-09.

[32] T. Claburn, "John deere urged to surrender source code under gpl," https://www.theregister.com/2023/03/17/john_deere_sfc_gpl/, 3 2023, accessed: 2023-14-09.

[33] N. Gunningham, R. A. Kagan, and D. Thornton, "Social license and environmental protection: Why businesses go beyond compliance," *Law & Social Inquiry*, vol. 29, no. 2, pp. 307–341, 2004, DOI: 10.1111/j.1747-4469.2004.tb00338.x.

[34] T. Claburn, "Gpl legal battle: Vizio told by judge it will have to answer breach-of-contract claims," https://www.theregister.com/2022/05/16/vizio_gpl_contract/, 5 2022, accessed: 2023-14-09.

[35] J. Vincent, "The lawsuit that could rewrite the rules of ai copyright," https://www.theverge.com/2022/11/8/23446821/microsoft-openai-github-copilot-class-action-lawsuit-ai-copyright-violation-training-data, 11 2022, accessed: 2023-14-09.

[36] J. Wu, L. Bao, X. Yang, X. Xia, and X. Hu, "A large-scale empirical study of open source license usage: Practices and challenges," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. MSR '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 595–606, DOI: 10.1145/3643991.3644900.

[37] T. Wolter, A. Barcomb, D. Riehle, and N. Harutyunyan, "Open source license inconsistencies on github," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, Jul. 2023, DOI: 10.1145/3571852.

[38] C. Vendome, G. Bavota, M. D. Penta, M. Linares-Vásquez, D. German, and D. Poshyvanyk, "License usage and changes: a large-scale study on github," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1537–1577, 2017, DOI: 10.1007/s10664-016-9438-4.

[39] Y. Wu, Y. Manabe, T. Kanda, D. M. German, and K. Inoue, "Analysis of license inconsistency in large collections of open source projects," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1194–1222, 2017, DOI: 10.1007/s10664-016-9487-8.

[40] X. Cui, J. Wu, X. Ling, T. Luo, M. Yang, and W. Ou, "Exploring large language models for analyzing open source license conflicts: How far are we?" in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2025, pp. 291–302, DOI: 10.1109/ICSE-Companion66252.2025.00083.

[41] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, and H. Ji, "Lidetector: License incompatibility detection for open source software," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, Feb. 2023, DOI: 10.1145/3518994.

[42] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564, DOI:10.1109/MSR52588.2021.00074.

[43] bndr, "pipreqs," https://github.com/bndr/pipreqs, 2024.

[44] landscapeio, "requirements-detector," https://github.com/landscapeio/requirements-detector, 2023.

[45] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[46] pytorch, "Torch text," https://github.com/pytorch/text, 2024.

[47] p0n1, "Epub to audiobook converter," https://github.com/p0n1/epub_to_audiobook, 2024.

[48] G. Enderson, N. Wintersgill, T. Stalnaker, O. Chaparro, and D. Poshyvanyk, "Online replication package," https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/nwintersgill/sbom_quality_study&visit_type=git, 2026.

[49] vislearn, "Freia," https://github.com/vislearn/FrEIA, 2024.

[50] EducationalTestingService, "skll," https://github.com/EducationalTestingService/skll, 2024.

[51] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, p. 45, 2021, DOI: 10.1007/s10664-020-09914-8.

[52] S. Yu, W. Song, X. Hu, and H. Yin, "On the correctness of metadata-based sbom generation: A differential analysis approach," in *2024 54th Annual IEEE/IFIP International conference on dependable systems and networks (DSN)*. IEEE, 2024, pp. 29–36, DOI: 10.1109/DSN58291.2024.00018.

[53] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210, DOI: 10.1109/ICSE.2019.00122.

[54] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 590–601, DOI: 10.1145/3377811.3380405.

[55] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards better dependency management: A first look at dependency smells in python projects," *IEEE Transactions on Software Engineering*, 2022, DOI: 10.1109/TSE.2022.3191353.

[56] "SPDX License List," 5 2024. [Online]. Available: https://spdx.org/licenses/

[57] "Build a software bill of materials (sbom) for open source supply chain security," 2022, accessed: 2024-25-06. [Online]. Available: https://snyk.io/blog/building-sbom-open-source-supply-chain-security/

[58] CycloneDX, "cyclonedx-maven-plugin," https://github.com/CycloneDX/cyclonedx-maven-plugin, 2024.

[59] anchore, "syft," https://github.com/anchore/syft, 2024.

[60] Microsoft, "sbom-tool," https://github.com/microsoft/sbom-tool, 2024.

[61] T. Bi, B. Xia, Z. Xing, Q. Lu, and L. Zhu, "On the way to sboms: Investigating design issues and solutions in practice," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, Jun. 2024, DOI: 10.1145/3654442.

[62] S. Cofano, G. Benedetti, and M. Dell'Amico, "Sbom generation tools in the python ecosystem: An in-detail analysis," in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2024, pp. 427–434, DOI: 10.1109/TrustCom63139.2024.00077.

[63] M. Wu, Y. Zhao, X. Hu, X. Zhan, S. Li, and X. Xia, "More than meets the eye: On evaluating sbom tools in java," *ACM Transactions on Software Engineering and Methodology*, 2025, DOI: 10.1145/3766073.

[64] A. A. Bangash, T. Ge, Z. Zhao, A. Singh, Z. Wang, and B. Adams, "The State of the SBOM Tool Ecosystems: A Comparative Analysis of SPDX and CycloneDX," *arXiv preprint arXiv:2512.21781*, 2025, DOI: 10.48550/arXiv.2512.21781.

[65] C. Wang, J. Wu, H. Lyu, X. Ling, T. Luo, Y. Wu, and C. Zhao, "A Large Scale Empirical Analysis on the Adherence Gap between Standards and Tools in SBOM," *ACM Transactions on Software Engineering and Methodology*, 2026, DOI: 10.1145/3788692.

[66] L. Lin *et al.*, "Generating Software Bill of Material for Vulnerability Management and License Compliance," 2023.

[67] Interlynk, "sbomqs: Quality metrics for sboms," https://github.com/interlynk-io/sbomqs, 2024.

[68] D. Bifolco, S. Nocera, S. Romano, M. Di Penta, R. Francese, and G. Scanniello, "On the accuracy of github's dependency graph," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 242–251, DOI: 10.1145/3661167.3661175.

[69] M. Ohm and A. Sykosch, "Compilation of SBOM dataset from 100,000 public GitHub repositories," 2025, researchGate preprint. Dataset available at Zenodo, DOI: 10.5281/zenodo.15334733.

[70] Y. Gamage, N. G. Fernandez, M. Monperrus, and B. Baudry, "Software bills of materials in maven central," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 339–343, DOI: 10.1109/MSR66628.2025.00062.

[71] R. Kishimoto, T. Kanda, Y. Manabe, K. Inoue, S. Qiu, and Y. Higo, "A dataset of software bill of materials for evaluating sbom consumption tools," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 576–580, DOI: 10.1109/MSR66628.2025.00090.

[72] S. Torres-Arias, D. Geer, and J. S. Meyers, "A viewpoint on knowing software: Bill of materials quality when you see it," *IEEE Security & Privacy*, vol. 21, no. 6, pp. 50–54, 2023, DOI: 10.1109/MSEC.2023.3315887.