2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 979-8-3315-0620-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/TrustCom63139.2024.00077

SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis

Serena Cofano
IMT Lucca & University of Genoa
Genoa, Italy
0009-0006-6539-9931
serena.cofano@imtlucca.it

Giacomo Benedetti
University of Genoa
Genoa, Italy
0000-0003-2609-6787
giacomo.benedetti@dibris.unige.it

Matteo Dell'Amico
University of Genoa
Genoa, Italy
0000-0003-3152-4993
matteo.dellamico@unige.it

***Abstract*—Software Bills of Material (SBOMs), which improve transparency by listing the components constituting software, are a key countermeasure to the mounting problem of Software Supply Chain attacks. SBOM generation tools take project source files and provide an SBOM as output, interacting with the software ecosystem. While SBOMs are a substantial improvement for security practitioners, providing a complete and correct SBOM is still an open problem. This paper investigates the causes of the issues affecting SBOM completeness and correctness, focusing on the PyPI ecosystem. We analyze four popular SBOM generation tools using the CycloneDX standard. Our analysis highlights issues related to dependency versions, metadata files, remote dependencies, and optional dependencies. Additionally, we identified a systematic issue with the lack of standards for metadata in the PyPI ecosystem. This includes inconsistencies in the presence of metadata files as well as variations in how their content is formatted.**

I. Introduction

Recent Software Supply Chain attacks [10], such as the *SolarWinds* incident [15] and the *xz* attack [14], have drawn significant interest from academics, industry players, and governmental organizations. The requirement for increased transparency on the Software Supply Chain, (i.e., the visibility on its components [18]), has given rise to the Software Bill of Materials (SBOM) [26] as a defense against these attacks.

An SBOM is a comprehensive list of the components constituting a piece of software [25]. Maintainers can use this list to quickly identify malicious or vulnerable software packages by feeding it into security analysis tools to find vulnerabilities. The US Government identified the SBOM as a primary tool for transparency and made it compulsory for government software in 2021 [17]. On March 12th, 2024, the European Parliament approved the EU Cyber Resilience Act (CRA) [11]. The CRA uses the SBOM to describe, record, and monitor the security of products.

Numerous studies have been conducted to examine the practitioners' perceptions of the SBOM [28, 29, 31]. Most users who include the SBOM into their security analyses agree that the presence of false positives and negatives makes SBOM quality a recurrent problem. An SBOM, to be useful from the point of view of security, should be *complete*, i.e., include all the components, and *correct*, i.e., provide exact information about them. Tools that generate the SBOM are often responsible for losing these properties. Although the issue is well known, few measures have been implemented, and little research has been done on the study of generation methodologies. Some works propose a study of the performance of the most commonly used open-source generation tools. They mostly focus on a single language, such as Java [2] and Javascript [24]. The decision to focus on a single language is driven by the fact that SBOM generation is significantly influenced by how the ecosystem manages a project's dependencies. Other works aim to give an overview of SBOM generation: Yu et al. [30] perform a large-scale differential analysis on four SBOM generation tools. The differential analysis allows them not to have to deal with ground truth and thus be able to consider multiple languages. On the other hand, such an approach has less detail about each individual language and on the causes that lead to different tools producing different SBOMs for the same piece of software.

Our study focuses on the elements that primarily affect SBOM creation in Python projects in terms of correctness and completeness. In this work, we identify problems in SBOM generation and determine their root causes in both ecosystem and SBOM generation tools.

The choice of Python is driven by the following reasons:

(i) Its widespread use [27, 8, 9].
(ii) Its flexibility: there is no standard for project creation, only guidelines (PEPs) that may not be followed. There are several tools for creating and managing Python projects and dependencies, and these tools require metadata files that specify dependencies in different ways; this flexibility has prompted the community to develop package managers capable of creating and managing Python project and its dependencies. However, these tools can create projects with differing metadata files containing project dependencies.
(iii) Dynamic dependency resolution. Developers can specify a range of versions or no version at all for dependencies, and this will be resolved at installation time. For example, when the version is not specified, the package manager will use the last available version. In this case, the version will be known only after installation and will depend on when the package is installed.

2324-9013/24/$31.00 ©2024 IEEE
DOI 10.1109/TrustCom63139.2024.00077

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

In detail, we provide the following contribution:

- A study on the impact of the Python ecosystem on the generation of SBOMs. In detail, we investigate how, given the same set of dependencies, the methods used to generate the Python project influence the SBOM computed on the projects. For this goal, we create a set of Python projects with the same dependencies but using different package managers.
- A study on how the approach used by SBOM generation tools changes the final output of the SBOM. We select four open-source SBOM generation tools and we run them on the projects.

We find substantial differences in the generated SBOMs, due to tool-related causes, different Python project configurations, or intrinsic to the ecosystem. We identify the lack of standards in the Python ecosystem and the defects in SBOM generation tools as main takeaways. In particular, we recommend that *(i)* Python package managers provide metadata in a consistent and stable format, and *(ii)* SBOM generation tools improve the support for the most recent and recommended project configuration file.

II. Background

The security of a Software Supply Chain depends on the security of its components. To monitor each component and conduct security analysis, it's essential to have a complete and correct SBOM. The SBOM serves as input for security analysis tools, allowing the identification of vulnerabilities [1]. This is crucial for security maintainers to address any issues promptly. This section contains notions of what a SBOM is and how it can be generated. Also, since our study focuses on SBOMs related to Python projects, it describes how Python projects are composed and how package managers handle dependencies and resolve versions.

A. Software Bill Of Material

The SBOM is an inventory of software components and dependencies, information about those components, and their hierarchical relationships [25]. There are multiple standards for SBOM. The most used are CycloneDX [19] and SPDX [12]. We focus our work on CycloneDX, because it is designed primarily to generate the SBOM and is more suited for providing precise software components, which is why it is more effective for vulnerability management. Conversely, SPDX was designed as a tool for managing licenses, and it is now primarily used for development [3].

The CycloneDX standard supports JSON, XML, and Protobuf formats as output [20]. The standard covers multiple aspects of the Software Supply Chain composition, such as the human factor, licensing, vulnerabilities, metadata, and software components. In particular, the latter is about the software dependencies represented by the SBOM. A *component* is represented along with information useful to identify it, e.g., name, version, package URL (*purl*), and licenses. A purl is a URL string used to identify and locate a software package universally and uniformly across programming languages, package managers, packaging conventions, tools, APIs and databases.[1] SBOMs are produced by SBOM generators, which are either static or dynamic. Static tools generate SBOMs by examining the binary or package dependencies without executing the software. They aim to efficiently scan large codebases, extract metadata, and identify a wide range of components and licenses, with a low impact on computing resources. Dynamic tools generate SBOMs by simulating and interacting with the system where the software is installed. The approaches used by these tools include environment scanning, i.e., simulating an installation by creating a virtual environment and looking for installed dependencies, and runtime monitoring, i.e., instrumenting the execution. While computationally more expensive, dynamic SBOM generation should provide a more accurate reflection of the software's actual composition in production environments. However, this technique is not based on a real installation process, but on a simulated one which may happen at a different time from real installations. As a result, there may be differences in the installed dependencies based on the particularities of the installation environment's setup or on installation time.

B. Python

PyPI[2] is the Python ecosystem's package registry. It indexes Python packages, allowing developers to add them as dependencies to their applications. Python packages can be distributed as either build or source distributions, which are both archives: build (also known as *wheel*) distributions are zip archives, whereas source distributions are *tarballs* (i.e., archives compressed with the `tar` utility). A Python project is composed of source code and metadata files. Metadata files are required to define the project properties and to enable the package manager to build the project. These files can be `pyproject.toml`, `setup.py`, `requirements.txt`, and lockfiles. The `pyproject.toml` and `setup.py` files are mandatory for a project to be defined as a package and uploaded to the PyPI registry. The former is the standard metadata file since 2016 [5], while the latter is a legacy version still in use. The `requirements.txt` file contains a list of the direct dependencies necessary for the correct functioning of the final project. Finally, the *lockfile* is the most complete representation of the software. It contains dependencies with exact versions, including transitive dependencies (i.e., dependencies of dependencies), and guarantees integrity by hashing package content. It is automatically generated while dependencies are declared in the project.

The metadata files include dependencies specified with versions in various formats [7]. Versions can be either *pinned* or *unpinned*. Pinned versions specify the exact version using the version matching operator "==" and are the way dependencies are represented in lockfiles. Alternatively, versions can be pinned or left unpinned in the `requirements.txt`, `pyproject.toml`, and `setup.py` files. The *unpinned* dependencies are either *unversioned* or *constrained*, i.e., they specify a

[1] https://github.com/package-url/purl-spec
[2] https://pypi.org/

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

range of possible versions. Constrained dependencies can have different clauses: the compatible release clause "~=",[3] which matches any candidate version that is expected to be compatible with the specified version, the version exclusion clause "!=", and the inclusive and exclusive comparison clauses "<=", ">=", "<", and ">".

The metadata files provide a way to distinguish *required* dependencies from *optional* ones. Required dependencies are those necessary to run the project; optional ones are either used during the development phase (*development* dependencies), or are related to additional features that can be made available users. The final user of a Python project can decide whether to install any optional dependency. While the `pyproject.toml` and the lockfile group the two in different sections of the files, `requirements.txt` does not distinguish the two. In this case, the alternative is to list optional dependencies in a separate file.

Different package managers can handle Python projects. Package managers can be divided in *front-ends* and *back-ends*. The front-end manages metadata and dependencies and communicates with the back-end using the project's metadata file (`pyproject.toml` or `setup.py`). The back-end is responsible for building the actual project in a distributable package. Choosing a back-end that aligns with the selected front-end is important, as not every front-end supports every back-end. Additionally, front- and back-ends can differ in their support for non-Python code, the presence of a lockfile, and their ability to obtain packages from a registry other than PyPI.

III. Study methodology

Here, we illustrate how we conduct our study on SBOM completeness and correctness in Python projects. We aim to identify the issues with SBOM generation and determine their relation with the SBOM generation tool or the ecosystem.

We follow a three-step methodology. The first step concerns the *experimental setup*, in which we create a dataset of Python projects and select a set of SBOM generation tools for running the experiment. The second step is *SBOM generation* for our Python projects dataset using the selected tools. The third step is *analysis*; this phase involves inspecting and comparing the SBOMs to discover significant issues in their generation and their causes. This is done by thoroughly diving into the implementation of the selected SBOM generation tools and comparing it with the specific Python projects for which the tool is executed.

A. Experimental Setup

This step consists of creating a dataset[4] of Python projects and selecting SBOM generation tools.

1) Dataset creation: Our Python projects dataset is synthetic. We make this choice rather than using existing active projects because we:

(i) want to test patterns of dependency handling in the PyPI ecosystem by observing SBOM generation tools on projects with the same dependencies but created with different technologies;

(ii) do not have a way to create a trusted ground truth for the components of a large existing project [30].

[3]https://packaging.python.org/en/latest/specifications/version-specifiers

[4]https://github.com/serenacofano/SBOM-python-ecosystem

TABLE I
BACK-ENDS (BE) AND FRONT-ENDS (FE) SELECTED FOR PYTHON PROJECTS GENERATION

BE \ FE	Pip	Hatch	Pdm	Pipenv	Poetry
Hatchling	✓	✓	✓	-	-
Flit	-	-	✓	-	-
Pdm	✓	✓	✓	✓	-
Poetry	-	-	-	-	✓
Setuptools	✓	✓	✓	-	-

Therefore, for the creation of the dataset we have to *(i)* select the tools in Python that allow us to create a project, and *(ii)* select the content of the projects, i.e., what dependencies they should contain.

We base the project setups on an assessment of online articles on package managers to match reality. The following factors are taken into account while choosing the front-end and back-end:

- number of Python projects that adopt them;
- number of articles that use them in a comparison or suggest them for building a Python package;
- amount of documentation present about them;
- activity on maintenance of them: we take into account only actively maintained package managers;
- scope of the package managers: we consider those limited to the managing Python projects.

Due to these criteria, we select *Setuptools* and *Poetry*, which are the most used tools [21]. Moreover, we select *Pdm*, *Flit*, *Pipenv*, and *Hatch* which are less used, but are cited in articles about package managers for Python [22, 4, 23, 13]. We end up with 5 back-ends and 5 front-ends. Of the 25 possible combinations, 12 are compatible, as shown in Table I. We analyze all of them.

Next, we select the dependencies to add to each project. The purpose is to study SBOM generation tools behavior in as many case scenarios as possible. As a result, we choose to include pinned, unversioned, and unconstrained dependencies. In addition, we consider origins other than the default PyPI, such as GitHub and other registries identified with URLs. We also consider optional and development dependencies. This allows us to choose the dependencies from Table II. Only one dependency, Numpy, is imported and used in the code. This helps us understand whether there are differences between just declaring a dependency and actually using it.

Each element in the sample is created following this procedure:

1) Initialize the project according to the selected front-end.

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

TABLE II
SET OF INSTALLED DEPENDENCIES INSIDE OF EACH PROJECT IN THE SAMPLE.

Name	Version	Type
numpy	Unversioned	Imported and used in the code
docopt	Pinned	Remote
black	Pinned	Remote from Git
seaborn	Pinned	Not used in the code
matplotlib	Contrained	Not used in the code
urllib3	Unversioned	Not used in the code
nltk	Unversioned	Optional dependency
pytest	Unversioned	Development dependency

2) Add the selected dependencies.
3) Add a main file with example code importing an installed dependency, and using it.
4) Use the package manager to package the project in a distributable wheel and a source tarball (to test the correctness of the project setup).

2) SBOM generation tools selection: We select SBOM generation tools among those adopting the CycloneDX standard. From the complete list of 219 tools hosted by the CycloneDX Tool Center,[5] we filter for those having:

- an open-source implementation;
- a command line interface;
- a recent release (i.e., at least a new release in 2023);
- Python support;
- a standalone implementation (i.e., not being an extension of other tools);
- support for standard installation scenarios (e.g., we excluded tools scanning just container images);
- support for recent CycloneDX versions (at least version 1.3).

We select *Trivy*, *Syft*, *Cdxgen*, and *Ort*.

Trivy and Syft are static tools, which only read and parse a selection of metadata files containing the dependencies. They differ in how they resolve versions; more details and the impact of this will be discussed in Section IV-A.

Cdxgen has a hybrid approach between static and dynamic. It simulates an installation by creating a Python virtual environment and installing the dependencies; it then scans the content of the environment and fetches the installed dependencies to generate the SBOM. If this procedure fails, it proceeds parsing metadata files.

Ort parses only the `requirements.txt` file as metadata; if it is not present but the project was created using Poetry and Pipenv, it uses the capability of those tools to generate `requirements.txt`. Since transitive dependencies are not present in `requirements.txt`, Ort exploits the PIP logic and directly queries PyPI to get them.

[5]https://cyclonedx.org/tool-center/

TABLE III
LIST OF SELECTED SBOM GENERATION TOOLS

	Tool version	CycloneDX version	Generation method
Trivy	0.49.1	1.5	metadata parsing
Syft	1.0.1	1.5	metadata parsing
Cdxgen	10.2.2	1.5	metadata parsing, environment analysis
Ort	17.0.0	1.5	metadata parsing, PyPI querying

The main characteristics of the tools are summarized in Table III.

B. SBOM Generation

We generate SBOMs for each project in our sample. Syft and Trivy run on the host machine, while Cdxgen and Ort run on a Docker container.

We obtain a total of 48 SBOMs. The 12 SBOMs produced by Ort were in XML format; we converted them to JSON using a Python script. We manually verified that the conversion did not affect the SBOM content.

C. Analysis

This phase is divided into two sub-phases: *(i)* observing the SBOMs and identifying the correctness (i.e., wrong versions) and completeness (i.e., lost dependencies) issues; *(ii)* looking for the causes of these issues, which are either due to the ecosystem itself (i.e., how the projects are structured) or to the SBOM generation tools (because of bugs and/or root causes of the methodology used).

1) SBOM Analysis: We manually analyze each SBOM. We verify completeness by checking whether direct, transitive, remote, and optional dependencies are present; for correctness, we check if the versions are exact. For unpinned dependencies, since we cannot determine a correct version, we compare the version numbers output by different SBOMs.

2) Investigation of Causes: After determining the critical issues in the generated SBOMs, we identify the causes. To do this we analyze tools through static code analysis, documentation study, and an assessment of the community reaction to some of the tool issues by searching for issues on GitHub. From this analysis, we can determine whether an issue is due to implementation or methodological flaws of the tools, or from the way the package managers handle dependencies. We discuss the results in Section IV.

IV. RESULTS

In this section, we analyze SBOMs for completeness and correctness, as discussed in Section III. We present identified issues and their causes, differentiating between ecosystem and SBOM generation tools origins. Table IV gives an overview of our findings, showing how SBOM generation tools behave based on the package manager.

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
RESULTS OVERVIEW.
NOTES: * ONLY DEPENDENCIES THAT ARE ALSO USED IN THE APPLICATION CODE , ** IT DOES NOT WORK DUE TO IMPLEMENTATION ERRORS.
CDXGEN = ▲, ORT = ⧗, SYFT = ●, TRIVY = ■

	Hatch	Pdm	Pipenv	Poetry	Pip
Find direct dependencies	▲ *	▲	▲ ● ■	▲ ⧗ ● ■	▲ ⧗ ● ■
Find transitive dependencies	-	▲	▲ ● ■	▲ ⧗ ● ■	▲ ⧗ ● ■
Find remote dependencies	-	▲	▲ ● ■	▲ ● ■	▲
Find optional dependencies	-	▲	● ■	▲ ⧗ ● ■	▲ ⧗ ● ■
Implement resolution of not present versions	NA	NA	NA	NA	▲ ⧗
Implement resolution of constrained versions	NA	NA	NA	NA	▲ ⧗ ●
Implement parsing of lockfiles	NA	▲	▲ ⧗ ** ● ■	▲ ⧗ ● ■	NA
Implement parsing of requirements.txt	NA	NA	NA	NA	▲ ⧗ ● ■
Implement parsing of pyproject.toml	-	-	-	⧗ ■	-

We identify two scenarios when these issues arise: *version management* and *metadata file handling*. Therefore, we group the issues into these two categories and consequently divide the section. Table V illustrates how issues within these categories impact the completeness and correctness of SBOMs across the ecosystem (E) and the SBOM generation tools (T).

Recall Section III, we imported a dependency in the code, i.e., Numpy. It is noteworthy that the source code is never analyzed by SBOM generation tools, with the only exception of Cdxgen for the project generated by Hatch. This is a fallback approach Cdxgen uses when there are not *supported* metadata files. We conducted some manual experiments on this behavior noticing that it works only in specific cases. Since SBOMs are not affected by the actual importing of a dependency in the source code, we do not report any specific result for that.

A. Version Management Issues

Version management is the process of assigning a version to a package. If the version is explicit, the tools should detect it; otherwise, they should determine it.

As discussed in Section II-B, package managers allow declaring dependencies without specifying an exact version (E9). Specifically, they allow omitting the version or indicating a range of possible versions; the exact version is defined at installation time. SBOM generation tools implement techniques to resolve the version.

The tools based solely on static analysis of metadata files, Syft and Trivy, ignore unversioned dependencies and do not include them in the final SBOM (T3), resulting in a loss of information.

When the version is indicated through a range, Syft applies guessing techniques (T6). Specifically, the guessing technique identifies comparison operators, discarding those not including "=" and converting the remaining ones to "==". For versions like "`1.1.*`", the asterisk is replaced with a "`0`". Considering that Yu et al. found that on average only about 46% of the dependencies in the `requirements.txt` files are pinned [30], this technique leads to a significant number of false positives in the SBOM, affecting its correctness.

Tools such as Cdxgen and Ort try to exploit the same version resolution mechanism used by package managers. Cdxgen simulates the installation, while Ort contacts the PyPI API. Constraints are resolved in the same way that they are during installation. Theoretically, this technique allows replicating what is done at installation time; however, it is valid only if installation and SBOM generation happen at the same time. For example, if an unspecified version is resolved as "latest" at a particular moment and the project is installed later, the two "latest" versions may not coincide. This would result in a loss of correctness and completeness.

B. Metadata File Management Issues

Metadata file management includes identifying files, parsing them, or using them to install dependencies in a simulated environment.

We categorize the discussion based on the specific files involved: lockfile, `pyproject.toml`, and `requirements.txt`. Each paragraph highlights the challenges and inconsistencies associated with these files, and how they impact the completeness and correctness of the SBOM. Finally, we highlight some specific aspects of addressing optional and remote dependencies in metadata file handling.

1) Lockfiles: Most SBOM generation tools rely on the lockfile to collect dependencies. As discussed in Section II, a lockfile is supposed to be the most complete representation of software, freezing both direct and transitive dependencies to specific instances. These files, although detailed, lack a standardized format in the Python ecosystem (E8), making their usage difficult. Python projects can use various formats for lockfiles, such as those from Poetry, Pipenv, and Pdm, resulting in tools developing ad hoc implementations for each (E8). When the tool does not implement a specific lockfile, it cannot parse it and thus ignores it, losing the dependencies it contains (T2). This lack of standardization causes variability and inconsistencies in the generated SBOMs, affecting both

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

TABLE V
COMPREHENSIVE OVERVIEW OF ISSUES, DEPENDING ON TOOLS (T#) AND ECOSYSTEM (E#).

Property	Effect	Issue
Completeness	**Missing Dependencies**	**T1** SBOM generation tool does not consider pyproject.toml file.
		T2 SBOM generation tool does not consider lockfile.
		T3 SBOM generation tool ignores dependencies without pinned version.
		T4 SBOM generation tool fails in correct parsing of the optional dependency.
		T5 SBOM generation tool does not properly parse the URL of the packages.
		E1 Ecosystem has two build interfaces, setup.py and pyproject.toml.
		E2 The use of a lockfile is not mandatory.
		E3 The ecosystem does not provide a standard file name for the requirements.txt file.
		E4 Metadata files contain only direct dependencies.
		E5 Package manager does not create a lockfile.
		E6 Lack of a univocal standard for declaring optional dependencies.
		E7 Package managers do not explicitly declare version of the package.
Correctness	**Wrong Dependencies**	**T4** SBOM generation tool fails in parsing of the optional dependency.
		T6 SBOM generation tool guesses the dependency's version.
		T7 SBOM generation tool does not report the origin of the packages.
	Missing Versions	**T5** SBOM generation tool does not properly parse the URL of the packages.
		E7 Package managers do not explicitly declare version of the package.
		E8 The format for the lockfile is not standardized.
		E9 Version can be omitted in metadata files.

completeness and correctness. Moreover, the lockfile is not mandatory for Python projects (E2). Thus, some package managers (e.g., Hatch) do not generate this file at all (E5).

Ort does not directly parse the lockfile, but instead converts them to `requirements.txt` format via the package managers' commands. This applies solely to Poetry and Pipenv—i.e., "`poetry export`" and "`pipenv requirements`". However, SBOM generation does not work for Pipenv due to a known implementation issue.[6]

2) `pyproject.toml`: `pyproject.toml` file is a standard for modern Python projects, hence package managers create it during project initialization. However, SBOM generation tools generally ignore this file (T1), except for Ort and Trivy, which consider the `pyproject.toml` file generated by Poetry. They parse the table containing dependencies, which is formatted with a syntax specific to Poetry. This approach leads to significant gaps in dependency collection, thus affecting both the completeness and correctness of the SBOM.

An example of this issue is the Hatch project, where neither lockfiles nor `requirements.txt` files are present. Even though the `pyproject.toml` file contains the full list of direct dependencies, the SBOM is empty because tools do not parse this file, resulting in a total loss of completeness.

Another issue is the coexistence of the old standard `setup.py` alongside `pyproject.toml` (E1). While the Python community strongly encourages updating to the new standard [5, 6], some tools still support only `setup.py`, resulting in incomplete SBOMs for new projects.

3) `requirements.txt`: All of the tools we analyze can retrieve direct dependencies from this file. However, if a developer chooses a different name for this file, these tools cannot detect it due to the absence of a standardized naming convention (E3). This inconsistency leads to a loss of completeness as the information in the `requirements.txt` file might be ignored.

Apart from lockfile, metadata files contain only direct dependencies by default (E4). Moreover, `requirements.txt` files typically include only direct dependencies (E4). Because of that, tools relying on static parsing of metadata files miss out on transitive dependencies when there is no lockfile present, thus affecting the completeness and correctness of the SBOM.

Remote Dependencies Remote dependencies are univocally identified by: name, version, and URL. This information can be either *(i)* wrapped by the URL or *(ii)* listed by field, depending on the lockfile format. Considering these two cases: *(i)* When SBOM generation tools have to extract the remote dependency information out of the URL, often do not correctly parse the URL (T5), leading to missing versions and incomplete dependencies. *(ii)* Because there are no standards in the format of lockfiles (E8), the parsing methods of some tools may not comply with how the dependency is declared. This can lead to missing or incorrect dependency information.

[6]https://github.com/aboutcode-org/python-inspector/issues/11

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

For example, the Pipenv lockfile omits versions for remote dependencies, causing SBOM generation tools to miss the dependency (E7).

Optional Dependencies The lack of a univocal standard for declaring optional dependencies (E6) causes misidentification and loss of these dependencies in the SBOM. SBOM generation tools sometimes fail to correctly parse optional dependencies (T4), resulting in missing dependencies. This issue is particularly evident in Pipenv, where Syft and Trivy fail to detect optional dependencies due to the formatting of the Pipenv lockfile. The lockfile divides dependencies into two groups: "default" and "develop", with the latter containing the optional dependencies. Syft and Trivy's parsing implementations only consider the "default" group, excluding optional dependencies from the SBOM. This causes a lack of correctness in the generated SBOM.

V. Related Works

Despite the emergence of the SBOM technology and the well-known problems of its generation [28, 31, 29], there is little literature on the precise definition of these problems and identification of the causes. We can classify the works on SBOM generation into two groups: those considering different programming languages and those focusing on a single language.

Mirakhorli et al. [16] perform an empirical analysis of tools related to SBOMs. They classify the open- and closed-source tools based on their role, then focus on the tools for SBOM generation and analyze five open-source tools testing them on a Java control project. Among the main takeaways they identify the lack of a reliable ground truth, inconsistency among SBOMs generated by different SBOM generation tools and a low accuracy in case of dependencies malpractices, e.g., hard-coding or dynamically loading dependencies.

Yu et al. [30] conduct a large-scale differential analysis of the correctness of four popular SBOM generators on 7,876 open-source projects written in Python, Ruby, PHP, Java, Swift, C#, Rust, Golang and JavaScript. The differential analysis makes it possible to avoid generating a ground truth at the cost of not evaluating the precision of the tools. Considering different languages allows them to have an overview of the correctness of the SBOM generation tools, but fewer details. Among the contributions, they focus on Python by demonstrating a possible attack; however, they do not discuss the differences among projects that use different package managers. Two works focus only on a single language: Balliu et al. [2] focus on Java, Rabbi et al. [24] focus on Javascript; they are related, with the second being inspired by the first. In this case, the authors recognize the need to focus on a single language. They focus on the impact the generation methodologies of the tools can have on the final SBOM, and on the critical aspects of the SBOM itself, rather than on the impact the ecosystem can have on the generated SBOM.

VI. Discussion

This study aims to understand the relationship between SBOM generation tools and the Python software ecosystem.

We identify two root causes common to all the generation issues explored in Section IV: *(i)* The lack of standards in the Python ecosystem, and *(ii)* the approximation of dependency solving of SBOM generation tools. From them, we provide consequent recommendations.

Major Takeaways

Lack of standards for Python. SBOM generation is largely helped by the presence of standards. Knowing which files must be in a project and how those files must be defined allows SBOM generation tools to automatically explore the filesystem and retrieve information necessary to generate a complete and correct SBOM. The lack of this property in Python makes SBOM generation hardly consistent with the expected standards. The clearest example is the lack of a standard for the lockfile. A lockfile should represent the most accurate representation of a software's dependency network; since, however, the format varies depending on the package manager, it is problematic for SBOM generation tools to use it to obtain a coherent SBOM.

Defects in SBOM generation tools. While the ecosystem is problematic, tools can also be blamed in various cases: *(i)* they do not consider the `pyproject.toml` file, missing dependencies in Python projects following the most recent standard; *(ii)* they implement inaccurate version-solving techniques that affect the SBOM correctness; *(iii)* they do not provide warnings when they cannot ensure completeness and correctness of the SBOM.

Recommendations

We provide two recommendations:

For the Python ecosystem: The Python ecosystem should push for initiatives proposing standards. Issues in SBOM generation can be largely addressed once the content of a Python project and its files is standardized.

For the SBOM generation tools: SBOM generation tools are required to consider the new Python standard build interface by parsing the `pyproject.toml` file. Most SBOMs will achieve completeness with this feature.

Additional Takeaway: Version Management

Version management is a complex theme because the version of unpinned dependencies is solved at installation time. An SBOM created at a different time from software installation can be incorrect (e.g., the latest version of a library changed), but also incomplete (e.g., the new library version introduced a new recursive dependency). Currently, SBOM generation tools seem to merely skim the issue; we think that in the future alternative solutions should be explored (e.g., deeper integration with package managers to pin all versions according to the SBOM, or associating SBOMs with complete installations rather than source versions).

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.

VII. Conclusion

We perform a study of SBOM generation in the Python ecosystem, identifying causes of missing completeness and correctness. We find that the lack of standards in the Python ecosystem is the main cause of inaccurate SBOMs, but also that SBOM generation tools do not use the `pyproject` file and do not implement proper techniques for obtaining correct dependency versions. Based on our findings, we provide recommendations that can solve issues affecting SBOM generation for Python projects, providing better SBOMs and, consequently, better transparency of the Software Supply Chain.

Acknowledgments

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Anchore. *Guide to SBOMs: What They are and Their Role in Cybersecurity*. URL: https://anchore.com/sbom/what-is-an-sbom/.

[2] Musard Balliu et al. "Challenges of Producing Software Bill of Materials for Java". In: *IEEE Security & Privacy* (2023). DOI: 10.1109/MSEC.2023.3302956.

[3] Barak Brudo. *SPDX vs. CycloneDX: SBOM Formats Compared*. URL: https://scribesecurity.com/blog/spdx-vs-cyclonedx-sbom-formats-compared/.

[4] Giacomo Caironi. *Python project managers comparison*. URL: https://tinyurl.com/ybd44u7d.

[5] Brett Cannon, Nathaniel J. Smith, and Donald Stufft. *PEP 518 - Specifying Minimum Build System Requirements for Python Projects*. URL: https://peps.python.org/pep-0518/.

[6] Brett Cannon et al. *PEP 621 – Storing project metadata in pyproject.toml*. URL: https://peps.python.org/pep-0621/.

[7] Alyssa Coghlan and Donald Stufft. *PEP 440 – Version Identification and Dependency Specification*. URL: https://peps.python.org/pep-0440/.

[8] Kyle Daigle. *Octoverse: The state of open source and rise of AI in 2023*. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/.

[9] Alexandre Decan, Tom Mens, and Maelick Claes. "On the topology of package dependency networks: a comparison of three programming language ecosystems". In: *ECSAW '16*. DOI: 10.1145/2993412.3003382.

[10] ENISA. *Threat Landscape For Supply Chain Attack*. 2021. DOI: 10.2824/168593.

[11] EU. *Cyber Resilience Act*. URL: https://www.cyberresilienceact.eu/.

[12] The Linux Fountation. *SPDX*. URL: https://spdx.dev.

[13] Shubham Gandh. *Managing Python Dependencies*. URL: https://www.fuzzylabs.ai/blog-post/managing-python-dependencies.

[14] Roman Kublin. *The XZ Attack – A Software Supply Chain Earthquake*. URL: https://myrror.security/the-xz-attack-a-software-supply-chain-earthquake/.

[15] Jeferson Martínez and Javier M Durán. "Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study". In: *International Journal of Safety and Security Engineering* (2021). DOI: 10.18280/ijsse.110505.

[16] Mehdi Mirakhorli et al. *A Landscape Study of Open Source and Proprietary Tools for Software Bill of Materials (SBOM)*. 2024. DOI: 10.48550/arXiv.2402.11151.

[17] NIST. *Executive Order 14028, Improving the Nation's Cybersecurity*. 2021. URL: https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.

[18] Chinenye Okafor et al. "SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties". In: *SCORED '22*. 2022. DOI: 10.1145/3560835.3564556.

[19] OWASP. *CycloneDX*. URL: https://cyclonedx.org.

[20] OWASP. *Specification Overview*. URL: https://cyclonedx.org/specification/overview/.

[21] pyOpenSci. *Python Packaging Tools*. URL: https://www.pyopensci.org/python-package-guide/package-structure-code/python-package-build-tools.html.

[22] PyPA. *Packaging Python Projects*. URL: https://tinyurl.com/4pam8jj4.

[23] PyPA. *Tool recommendations*. URL: https://packaging.python.org/en/latest/guides/tool-recommendations/.

[24] Md Fazle Rabbi et al. "SBOM Generation Tools Under Microscope: A Focus on The npm Ecosystem". In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (2024). DOI: 10.1145/3605098.3635927.

[25] *Software Bill Of Material*. URL: https://www.ntia.gov/page/software-bill-materials.

[26] *Software Bill Of Materials.* URL: https://www.cisa.gov/sbom.

[27] Stackoverflow. *Developer Survey*. URL: https://tinyurl.com/yhbf84m2.

[28] Trevor Stalnaker et al. "BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems". In: *ICSE '24*. 2024. DOI: 10.1145/3597503.3623347.

[29] Boming Xia et al. "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead". In: *ICSE '23* (2023). DOI: 10.1109/ICSE48619.2023.00219.

[30] Sheng Yu et al. "On the Correctness of Metadata-based SBOM Generation: A Differential Analysis Approach". In: *DNS '24*. DOI: 10.1109/DSN58291.2024.00018.

[31] Nusrat Zahan et al. "Software Bills of Materials Are Required. Are We There Yet?" In: *IEEE Security and Privacy* (2023). DOI: 10.1109/MSEC.2023.3237100.

Authorized licensed use limited to: CNR Biblioteca Centrale. Downloaded on July 28,2025 at 15:33:53 UTC from IEEE Xplore. Restrictions apply.