# The Impact of SBOM Generators on Vulnerability Assessment in Python: A Comparison and a Novel Approach

Giacomo Benedetti[1⋆][0000−0003−2609−6787],
Serena Cofano[1,2][0009−0006−6539−9931],
Alessandro Brighente[3][0000−0001−6138−2995], and
Mauro Conti[3][0000−0002−3612−1934]

[1] University of Genoa, Genoa, Italy
giacomo.benedetti@dibris.unige.it
[2] IMT School for Advanced Studies Lucca, Lucca, Italy
serena.cofano@imtlucca.it
[3] University of Padua, Padua, Italy
{alessandro.brighente,mauro.conti}@unipd.it

**Abstract.** The Software Supply Chain (SSC) security is a critical concern for both users and developers. Recent incidents, like the SolarWinds Orion compromise, proved the widespread impact resulting from the distribution of compromised software. The reliance on open-source components, which constitute a significant portion of modern software, further exacerbates this risk. To enhance SSC security, the Software Bill of Materials (SBOM) has been promoted as a tool to increase transparency and verifiability in software composition. However, despite its promise, SBOMs are not without limitations. Current SBOM generation tools often suffer from inaccuracies in identifying components and dependencies, leading to the creation of erroneous or incomplete representations of the SSC. Despite existing studies exposing these limitations, their impact on the vulnerability detection capabilities of security tools is still unknown.

In this paper, we perform the first security analysis on the vulnerability detection capabilities of tools receiving SBOMs as input. We comprehensively evaluate SBOM generation tools by providing their outputs to vulnerability identification software. Based on our results, we identify the root causes of these tools' ineffectiveness and propose PIP-SBOM, a novel pip-inspired solution that addresses their shortcomings. PIP-SBOM provides improved accuracy in component identification and dependency resolution. Compared to best-performing state-of-the-art tools, PIP-SBOM increases the average precision and recall by 60%, and reduces by ten times the number of false positives.

**Keywords:** Software Bill of Materials, Vulnerability Assessment, Dependency Network, Software Supply Chain Security

⋆ Corresponding Author

## 1 Introduction

The security of the Software Supply Chain (SSC) is an increasing concern for users and developers as reported by both ENISA [34] and the UE Executive Order on Improving the Nation's Cybersecurity [31]. Indeed, incidents such as the infection of SolarWind's Orion platform demonstrated how far-reached and impactful the distribution of compromised software is [37]. The security of the SSC depends on multiple factors, including, but not limited to, the use of open-source software as dependencies included in the developed application [32]. In a 2023 study of 1,703 commercial codebases across 17 industry sectors, Synopsys found that 96% of them leverage open-source code, and 76% of the total application code was open-source [45]. Therefore, targeting software components, such as libraries, allows attackers to affect a wide range of software using a single entry point [32, 48, 27, 33].

To improve the security posture of the SSC, the Executive Order [31] pushed the Software Bill of Materials (SBOM) as a tool to increase the transparency and verifiability of the distributed software. According to Cybersecurity and Infrastructure Security Agency (CISA), an SBOM is "*a formal record containing the details and supply chain relationships of various components used in building software*" [17], hence providing developers and enterprises with transparency in the software composition. An SBOM provides benefits for both software suppliers and consumers, as it helps identify and avoid known vulnerabilities, quantify and manage licenses, identify security and license compliance, and manage mitigation of vulnerabilities. SBOMs are generated by automated tools and created according to different formats, with the most common being Software Identification (SWID) tagging, Software Package Data Exchange (SPDX), and CycloneDx [17]. To fully leverage the information provided by SBOMs, a plethora of tools have been developed to receive SBOM as input and provide security information [8, 18, 1, 6]. To gather information on the security of components listed in the SBOM, these tools rely on open-source vulnerability databases, e.g., the NVD, to map components to vulnerabilities. Software components can be associated with different information security depending on the database [25]. Sometimes, SBOMs can also be complemented with a Vulnerability Exploitability eXchange (VEX), which provides additional information on possible specific vulnerabilities affecting software components of the SBOM. Overall, the use of SBOMs both increases the transparency of the distributed software and speeds up software adoption and testing. Indeed, retrieving known vulnerabilities of the listed software components via polling a public database is faster than running static or dynamic analysis application security testing.

**Are SBOMs Improving Security?** Although the premises are good, the SBOM is not what it is expected to be for security. Most SBOM generation tools use specification files (e.g., setup.py, gemspec, package.json) to gather the dependency index. Other approaches are based on source or binary code parsing. Due to the lack of a standardized SBOM format and the limitations in the accuracy of existing SBOM generation tools [20], it is possible to end up with different

generated SBOMs for the same software. Indeed, the SBOM generation process depends on the tool's capability of correctly identifying components' names, versions, and dependencies. Wrongly identifying one or more of these elements impairs the representation capabilities of the resulting SBOM [20]. Moreover, the claims made by SBOM generation tools on their support capabilities for different metadata file formats and their performance on real code bases are not consistent. Indeed, different and well-known SBOM generation tools are prone to parsing errors or inability to correctly gather all dependencies, resulting in an SBOM that represents a subset of the entire code base [23, 19, 42, 22]. This represents a problem not only because the SBOM does not accurately represent the code, but also because missed key dependencies may lead to missed key security issues. Such dependency resolution problem is still an open issue for SBOM generation, and it is not clear how this impacts the security analysis capabilities provided via SBOMs. Approaches such as code-centric call graph analysis and behavioral analysis may solve this problem, however, they are highly resource intensive [30, 38–40].

The limitations of the currently existing SBOM generation tools and the need for secure solutions to improve the security posture of the SSC led us to the following research questions:

- **RQ1**: *How much does the SBOM generation process impact the detection of vulnerabilities in the dependency network of an SSC?*
  - **RQ1.1**: *How does a specific vulnerability scanner perform when fed with an SBOM generated by a specific state-of-the-art SBOM generation tools?*
  - **RQ1.2**: *How much does an SBOM generation approach affect the performance of a vulnerability scanner?*
- **RQ2**: *How can we improve the SBOM generation approach to achieve better performance on the security assessment of the dependency network in an SSC?*

**Contributions.** In this paper, we evaluate for the first time how the representational capabilities of SBOM generation tools impact the identification of known vulnerabilities in the SSC. Despite existing work evaluating SBOM generation tool according to their ability to correctly identify component's name, version, and dependencies for Java [19], JavaScript [42], and Python [22], no existing work evaluated their impact on the detection of security issues. As outlined in our research questions, we expect these issues to greatly impact on the identification of known vulnerabilities in the SSC. To this aim, we selected five of the most relevant SBOM generation tools — i.e., cdxgen, GH-sbom, ORT, Syft, and Trivy— for the evaluation. Since many SBOM generation tools operate on the set of the most used programming languages (e.g., Python, JavaScript), we focus on the Python programming language, the most popular programming language in 2024 according to IEEE [21]. To solve the issues of SBOM generation tools and improve the security posture of the SSC we propose PIP-SBOM, a novel `pip`-based solution. Our solution overcomes the most relevant issues of SBOM

generation tool, i.e., it correctly identifies component names and versions and can correctly report all software dependencies. We compare the performance of our solution with that of existing state-of-the-art tools and show that the SBOM generated with our tool drastically (64% more precise than the best performing SBOM generation tool) increases the capabilities of identifying known vulnerabilities in the SSC.

We summarize our contributions as follows.

- We evaluate the capabilities of SBOM generation tools in helping increasing the SSC security posture. By providing generated SBOMs as input to a vulnerability scanner tool, we evaluate each SBOM generation tool in terms of the number of identified known vulnerabilities.
- We propose PIP-SBOM, an extension for PIP to generate an SBOM directly from the package manager, improving both usability and accuracy of the SBOM in the Python ecosystem.

## 2 Background

This section provides the necessary background on the SBOM generation process (Section 2.1), the dependency management and resolution for Python (Section 2.2), and the usage of SBOMs by vulnerability scanners (Section 2.3).

### 2.1 SBOM Generation Process

An SBOM is generated using tools commonly known as SBOM generation tools. Differently from Software Composition Analysis (SCA) tools, a SBOM generation tool does not analyse licenses and the security posture of components in the software under scrutiny. However, it may be part of the SCA, providing inputs for further analysis of the identified assets. SBOM generation tools take as input the software's project folder and produce a list (i.e., the SBOM) of software components and their dependencies, along with their version and other information useful to trace the software composition. As we focus on Python, in this paper, we showcase the SBOM generation approach for this language. However, the same process applies to all the other programming languages with just some differences related to their dependency resolution process.

Python has multiple package managers that a developer can choose to deal with dependencies and other project management operations. Each package manager chooses how to deal with the project's filesystem to coordinate the dependency management. Depending on the package manager, the project may result in very different filesystem structures.

SBOM generation tools analyze the structure of a Python package or project and produce an SBOM. How the SBOM is generated depends on the generation approach implemented by the SBOM generation tool. Most tools rely on static metadata-based generation methods. However, previous research [49, 42] reported that SBOMs have scarce accuracy when generated by tools currently

implementing this approach. Other tools, such as cdxgen, aim to reproduce an installation environment where dependencies are collected according to metadata files. NTIA established the minimum required elements for an SBOM [35]. Tools such as sbom-scorecard [9] can quantify the level of compliance. However, concerning vulnerability assessment, the required elements include only dependency identifiers — i.e., name, version, and package URL (purl).

### 2.2 Dependencies Management in Python

A Python project should contain either a `setup.py` or a `pyproject.toml` file to be managed by a package manager. Both these files contain the project's metadata and list the dependencies required to properly build and operate the project. The project's build produces an artifact, a wheel or source distribution, containing the source code files and all the additional files required in the metadata file.

A distributable artifact is installed through PIP. During installation, the dependencies listed in the metadata file are collected and installed on the user's system. Since transitive dependencies — i.e., dependencies of a dependency — are not shipped together with the distributable artifact, PIP uses the `resolvelib` package implementing a specific algorithm for dependency resolution [24].

Dependencies are listed in the metadata files by name and version. The dependency's version can be either pinned, non-pinned, or omitted — i.e., not specified at all, in this case, the package manager collects the dependency to the latest stable and available version. PyPI allows for multiple versioning schemas [11], such as semantic versioning [41], calendar versioning [29], and their combinations. The pinning vs. non-pinning choice is left to the developer by using the '`==`' operator vs. using range operators — i.e., `<=`, `>=`, `<`, `>`, `!=`.

### 2.3 Vulnerabilities Scanning with SBOM

In this paper, we refer to *vulnerability scanner* as a tool that analyzes a software artifact and provides a *security report* listing potential vulnerabilities affecting the scanned product.

Analyzing a software's dependency network is not an easy task. Modern software vastly relies on third-party software, resulting in the size of the dependency network rapidly increasing. Vulnerability databases, such as NVD and OSV, contain entries for known software vulnerabilities, making the dependency network analysis easy and fast. SBOMs enable vulnerability scanners to check the presence of known vulnerabilities without retrieving the dependency list. Currently largely used vulnerability scanners — e.g., ShiftLeftScan [8], Grype [18], KubeClarity [6], Bomber [1] — use the SBOM as source for their analysis of dependencies. They usually run a SBOM generation tool in the background and use the generated SBOM to resolve components names and retrieve their security information from public databases (e.g., NVD).

```
{
  "matches": [{
    "vulnerability": { "id": "GHSA-9wx4-h78v-vm56",
    "severity": "Medium",
    "fix": { "version": "2.32.0" }},
    "relatedVulnerabilities": [{
    "id": "CVE-2024-35195",
    "severity": "Medium" }],
    "matchDetails": { "type": "exact-direct-match",
    "package": { "name": "requests", "version": "2.31.0" },
    "found": {
      "versionConstraint": "<2.32.0␣(python)",
      "vulnerabilityID": "GHSA-9wx4-h78v-vm56"
     }
    "artifact": {
    "name": "requests",
    "version": "2.31.0",
    "purl": "pkg:pypi/requests@2.31.0" }
  }]
}
```


**Fig. 1: Example of a condensed Grype scan report for a Python SBOM.**

The use of SBOMs makes the behavior of these vulnerability scanners straightforward. They parse the SBOM collecting dependency identifiers, such as purls, and search for a match in vulnerability databases.

In this paper, we use Grype to obtain security reports from SBOMs. A Grype's security report contains the following fields for each vulnerability:

- *Vulnerability*, information on the specific matched vulnerability (e.g. ID, severity, CVSS score, fix information, links for more information)
- *RelatedVulnerabilities*, information pertaining to vulnerabilities found to be related to the main reported vulnerability, e.g., if the tool matches a vulnerability on GitHub Security Advisory, also the upstream CVE is reported.
- *MatchDetails*, the elements matching the vulnerability, such as the version constraints for which the vulnerability is matched.
- *Artifact*, information about the location of the package within the directory, package type, licensing information, purl, CPEs, etc.

Figure 1 shows an example of a Grype's security report for a Python SBOM.

## 3 Experimental Setup

In this section, we describe the setup for our evaluation methodology. At first, Section 3.1 provides the process we used to gather Python projects. Section 3.2

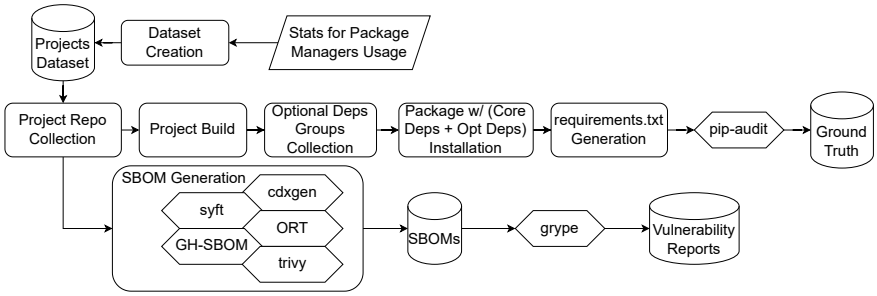


**Fig. 2: Experimental setup design. This approach provides us the necessary data to evaluate our research questions.**

describes our selection of SBOM generation tools based on their usage for security evaluation and their implemented generation method. Section 5 reports the workflow applied to the collected projects to generate SBOMs and obtain their security reports. Figure 2 depicts the overall experimental setup.

### 3.1 Projects Collection

Recalling from Section 2, Python allows the usage of multiple package managers, each implementing its way of dealing with dependencies. Since SBOM generation tools use specific parts of a Python project to generate the SBOM, we analyze tools' behavior in the context of different package managers. To collect a representative sample of projects, we extract the distribution of the package managers used in 1,351 packages randomly selected from the whole population list on `ecosystem.ms`. We discard packages that do not clearly state the package manager in their source code repository, obtaining the distribution of package managers shown in Table 1.

**Table 1: Package Managers and Their Usage**

| Package Manager | Packages | Percentage (%) |
|---|---|---|
| poetry | 38 | 6.44 |
| pdm | 9 | 1.53 |
| hatch | 85 | 14.41 |
| pipenv | 7 | 1.19 |
| conda | 0 | 0.00 |
| setuptools | 451 | 76.44 |

We collect a different sample of 1000 packages with the following process: (1) collect a random package from the entire package population hosted on PyPI;

(2) check the package manager used by the package; (3) if we already reached the quota of packages using that specific package manager we discard the package, otherwise we add the package to the sample. This process allows us to have a sample with the same proportion of package managers identified in the previous steps, and generalize our results to the entire package ecosystem with a 3.04% margin of error at 95% confidence level by standard sample size calculations.

### 3.2 SBOM generation tools selection

For the selection of SBOM generation tool we applied the following process: (1) We manually scraped the list of tools on the CycloneDX tool center.[4] We obtained a list of 169 open-source tools. (2) We analyzed each tool in the list selecting those that generate SBOMs, operate on Python, and have a command line interface. We reduce the list to 24 elements. (3) We manually tested the 24 tools to prove their utility in this work. We excluded those that do not correctly execute, require external technologies (e.g., build-root), or were not maintained in the last year. Eventually, we obtained a list of 5 tools: cdxgen, GH-sbom, ORT, Syft, and Trivy.

Table 2 lists the selected SBOM generation tools, along with the implemented generation methodology and some example vulnerability scanners making use of them. The evaluation of the SBOMs generated by these tools for a security assessment of the dependency network gives us the answer to RQ1.

**Table 2: List of the selected SBOM generation tools. Most of them are already officially used for dependency network security analysis. The selected tools can be also stratified based on the implemented generation method.**

| SBOM Gen. Tool | SBOM Gen. Met. | Example Sec. An. Tool |
|---|---|---|
| cdxgen | Environment Based | Shiftleft Scan, Macaron [15] |
| Syft | Metadata Based | Grype, KubeClarity |
| Trivy | Metadata Based | KubeClarity |
| ORT | Metadata Based | NA |
| GH-sbom | Dep. Graph Based | NA |

### 3.3 Security Report Ground Truth

A fundamental step to understand the effectiveness of SBOM generation tools in generating an SSC description that leads to the correct identification of vulnerabilities, is knowing the vulnerabilities that affect a specific project. To obtain this ground truth, we follow a multistep approach. For each package in our

[4] `https://cyclonedx.org/tool-center/`

collected sample we automatically: (1) retrieve the package's project from its code repository; (2) parse metadata files to obtain optional dependency groups; (3) install the package in a virtual environment along with both required and optional dependencies; (4) generate the requirements.txt file using the `pip freeze` command to filter out packages installed in the virtual environment by default; (5) pass the requirements.txt to pip-audit; (6) collect the security report. Thanks to this manual approach, we build the list of vulnerabilities associated with each project. A perfect SBOM generation tool will create a project representation the leads to the correct identification of this precise set of vulnerabilities.

### 3.4 SBOMs and Security Reports Generation

To generate relevant SBOMs, we feed our selected SBOM generation tools with the packages collected in our dataset. We parse SBOMs with `jq` [5] (a JSON parser) to verify they have the expected format. That is we verify they do not contain the metadata pointing out the correct analysis of the project by the SBOM generation tool.

We selected Grype [18] as the tool for the generation of security reports. Grype is a vastly used tool for the security analysis of projects [12, 3, 47, 13]. It covers multiple languages — e.g., Python, Go, Rust — and artifacts — e.g., Docker images, filesystems, SBOMs. As we do not need specific security analysis tool for this work, the tool just needs to parse the SBOM and query vulnerability databases for known vulnerabilities. Grype queries multiple databases, cross-checking vulnerabilities, and hence represents a perfect choice for our purposes.

A Grype's security report contains matches for found vulnerabilities, allowing us to compare the set of found vulnerabilities with the set of vulnerabilities reported in the ground truth.

## 4 PIP-SBOM: Our Proposed SBOM Generator

This Section describes PIP-SBOM, our PIP-based approach for native generation method. Figure 3 depicts its main steps.

PIP-SBOM is designed as part of PIP, the PyPI official package manager. We extend this specific package manager because: (1) it supports multiple front-ends and back-ends, i.e., it can build other package manager projects, such as poetry. (2) Both skilled and novel developers commonly use it.

PIP is internally based on modules representing the possible CLI commands for a user. We added a module — i.e., SBOM— containing the logic needed to generate the SBOM[5]. This allows developers to generate an SBOM for a Python project with the command: `pip sbom <project-path>`.

PIP-SBOM includes an online process and an offline process. The online process interacts with the PyPI registry obtaining the dependency network. The

[5] PIP-SBOM is the extended version of PIP, while SBOM is the specific module extending PIP. Hereafter, we refer to both of them as PIP-SBOM, for simplicity of language.

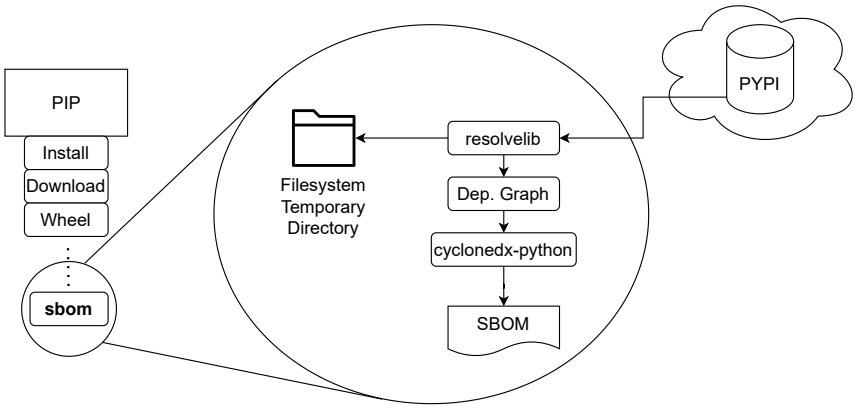


**Fig. 3: Design of PIP-SBOM. We extend the implementation of PIP to include SBOM generation in the build phase.**

offline process builds the dependency network graph and generates the SBOM document from a Python project.

**Dependency Network Solving** PIP uses the `resolve-lib` package dealing with dependencies and version constraints. This package optimizes the solving algorithm with the optimal navigation path of the dependency tree [7]. We build upon the logic already implemented in PIP for this package to mimic the same solving algorithm used during dependencies retrieval.

This process has a similar behavior to the `download` command. The project's dependencies are collected from PyPI and stored inside a directory specified by the user. In our implementation, the dependencies are stored inside a temporary directory and removed at the end of the process. This process automatically solves version constraints similarly to the process happening during project build and installation, providing a reliable representation of the dependency network at installation time. When a constraint cannot be solved because of version incompatibility, it is discarded, as would happen during the project installation.

The generation of the dependency graph is coupled with this process. We decided to store collected dependencies as a graph to allow deeper investigation of dependency relationships when required.

**Dependency Network Graph** The dependency network is defined as a direct unweighted graph $G = (V, E)$. Each element $n \in V$ is either a direct or transitive dependency of the input project $r \in V$. An edge $(u, v) \in E$ represents a dependency relationship between the nodes $u$ and $v$. This kind of dependency is a transitive binary relation. That is, from $u \to v$ and $v \to w$ it follows $u \to w$. In this case, we say that $u$ is a transitive dependency of $w$.

PIP-SBOM outputs the generated graph as a `dot` file when required through the `-g <file-name>` option. The PIP-SBOM internally uses the dependency graph to generate the SBOM.

**SBOM Generation** Once the graph is complete, PIP-SBOM module navigates the graph and creates an entry in the `components` field of the SBOM for each node. An entry contains: the bom-ref, dependency name, version and purl. We include only this information because they are necessary to the vulnerability scanner. In general, the SBOM can be enriched to comply with the minimum required elements stated by NTIA [35].

Edges of the dependency network are used to fill the *dependencies* field of the SBOM, which contains the relationship between components. When the graph exploration ends, PIP-SBOM produces a CycloneDx-compliant SBOM.

## 5 Evaluation Methodology

In this section, we present our methodology to answer our research questions. The evaluation process is divided into two parts. The first looks at the accuracy the vulnerability scanner reaches with SBOMs generated by selected SBOM generation tools. The second looks at data specific to PIP-SBOM to evaluate how an SBOM generated with a different method affects the vulnerability scanner performance.

**RQ1: SBOM Generation Impact on Vulnerability Scanning** This research question aims to understand to what extent the SBOM impacts the security analysis tool output. The SBOM should accurately represent the SSC. Thus, accurately representing the SSC is an enabling property for security analysis of the software dependency network.

*RQ1.1: How does a specific vulnerability scanner perform when fed with an SBOM generated by a specific state-of-the-art SBOM generation tools?* To answer this question, we compare the vulnerabilities identified starting from a SBOMs against the ground truth obtained through pip-audit. We use the Jaccard similarity index to compute the degree of overlap and commonality among the two vulnerability sets. For each tool and each SBOM $\mathcal{S}$, the process involves: (1) collecting the security reports generated from $\mathcal{S}$ and extract the matched vulnerabilities (`ToolVulns`); (2) getting the vulnerabilities stored in the ground truth for the project associated with $\mathcal{S}$ (`GrTrVulns`); (3) compute the Jaccard similarity index between `ToolVulns` and `GrTrVulns`, according to Equation (1).

$$J(ToolVulns, GrTrVulns) = \frac{|ToolVulns \cap GrTrVulns|}{|ToolVulns \cup GrTrVulns|} \quad (1)$$

*RQ1.2: How much does an SBOM generation approach affect the performance of a vulnerability scanner?* While Jaccard similarity represents a useful metric to assess the extent a tool suits security purposes, it cuts off details on the reasons behind the tool's performance. To obtain these missing details and answer *RQ1.2*, we compute the false positives, false negatives, precision, and recall.

Specifically, *precision* assesses the fraction of correctly identified vulnerabilities, according to the ground truth, among all identified vulnerabilities (Equation (2)).

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

*Recall* measures the fraction of correctly identified vulnerabilities to all actual vulnerabilities in the ground truth (Equation (3)).

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

These metrics provide a holistic overview of the performance of each tool's generation method: precision addresses the trustworthiness of identified vulnerabilities, and recall looks at the generation methods' efficacy in allowing security assessment to pinpoint pertinent vulnerabilities.

**RQ2: A Better SBOM Generation Approach** As we later show, the main issue with currently existing SBOM generation approaches and the resulting poor performance of vulnerability identification approaches is that SBOM generation tools are not able to correctly identify components and their dependencies. To answer *RQ2* and improve the security posture of the SSC, we extend PIP to investigate to what extent applying the same logic used in the retrieval of dependencies during a project installation for SBOM generation is beneficial. The extension leverages the already present `download` module in PIP. This module helps to download the package's archives without installing them. Modifying this module by adding functional elements for SBOM generation provides us the means for creating a novel and better approach. We provide the details of our implementation in Section 4.

To evaluate the improvement in the security analysis by using a SBOM generated with our approach, we apply the same metrics used to answer RQ1.

## 6 Evaluation Results

In this Section, we present the results of our analysis organized by research question. In particular, Section 6.1 presents the results of *RQ1*, while Section 6.2 presents the results of *RQ2*.

### 6.1 RQ1: SBOM Impact on Vulnerability Scanning

Our findings show that the SBOMs generation approach deeply impacts the performance of vulnerability scans. All the analyzed SBOM generation tools

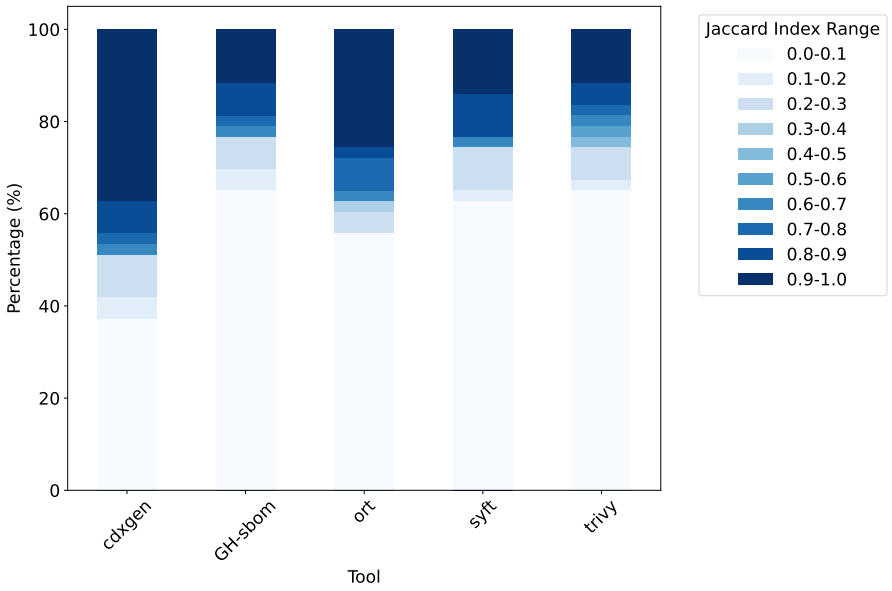


**Fig. 4: Jaccard Similarity Distributions. Each bar represents the percentage of SBOMs that lead to identification with a certain Jaccard index range.**

cannot lead to the correct identification of all vulnerabilities in more than 20% of cases, with the only exception of cdxgen achieving correct identification in almost 40% of cases.

**RQ1.1: Impact of tools on vulnerability scanner performance** The security reports generated from state-of-the-art SBOM generation tools reveal some shortcomings, particularly regarding thoroughness and accuracy in identifying vulnerabilities. The distribution of Jaccard similarity indexes in Figure 4 provides a perspective of the true positive vulnerabilities found with SBOMs generated by different tools. From the results, it is clear that SBOM generation tools badly impact on the vulnerability scans operated by vulnerability scanners.

Among the analyzed SBOM generation tools, the one providing the best vulnerability scan results is cdxgen. This is motivated by the fact that cdxgen (1) installs dependencies in a virtual environment, and (2) supports most of the package managers. These two properties result in being critical for a SBOM generation tool for a proper representation of the SSC. Hence, all tools lacking at least one of these two capabilities result in worse vulnerability scans.

ORT uses an approach similar to dependencies installation in a simulated environment. That is, it uses an external Python module to query the PyPI registry and obtain information on dependencies. However, its support for package man-

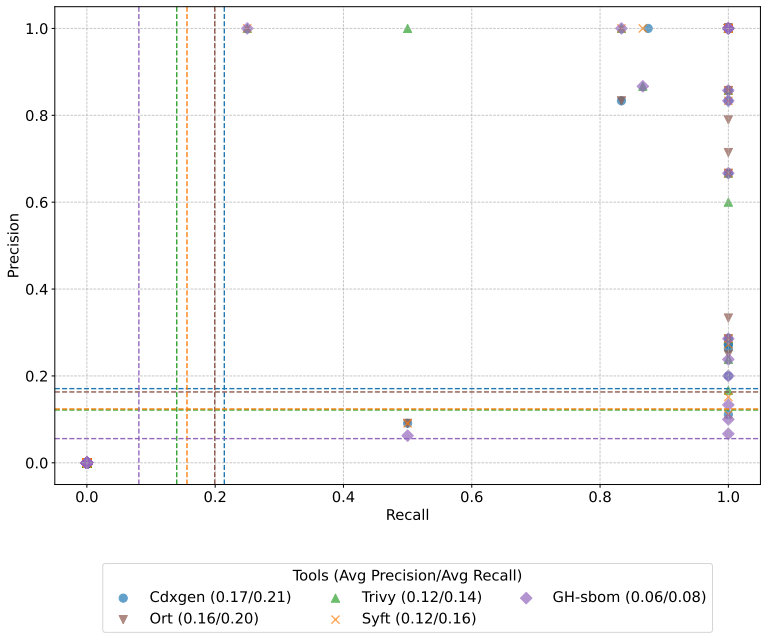


**Fig. 5: Precision and Recall for the vulnerability scans conducted through SBOMs generated by each of the selected SBOM generation tools.**

agers is limited to Poetry and Pipenv, or projects including `requirements.txt` files.

SBOM generators solely relying on static metadata — i.e., Syft and Trivy— display worse performances for vulnerability detection. This behavior is caused by the fact that they do not install dependencies, while they have good support for different package managers.

GH-sbom results are highly influenced by the settings applied by repository owners. Thus, GH-sbom works *only* when the dependency graph feature is enabled on the repository. However, its main issue is that it targets the last commit on the main branch to generate the SBOM. Hence, GH-sbom cannot acquire an SBOM for a specific commit or tag [16], causing vulnerability scanners to analyze SBOMs belonging to code different from the targeted one.

> **Takeaway** Current state-of-the-art SBOM generation tools are not suitable to generate SBOMs for the proper vulnerability assessment of Python projects.

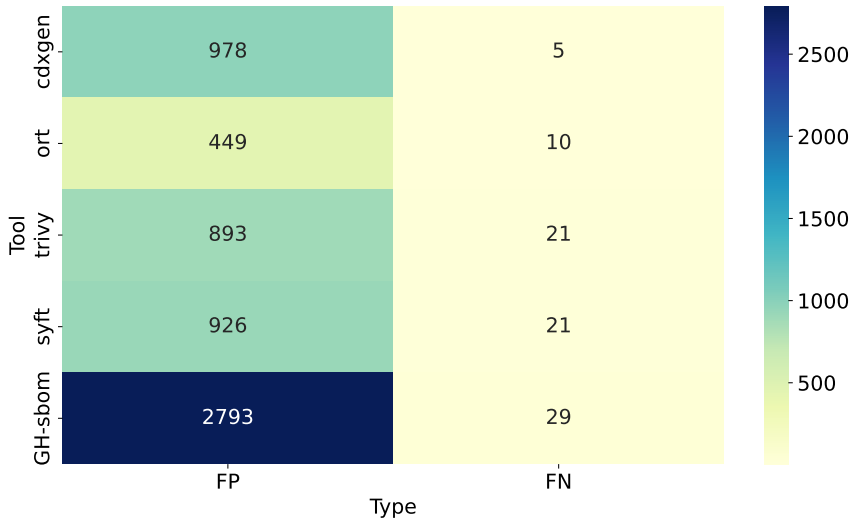


**Fig. 6: False Positives and False Negatives by Tool**

**RQ1.2: Causes of tool impact on vulnerability scanner** While the Jaccard similarity represents how much the SBOM generation approach impacts the vulnerability scan results, precision and recall measurements give us information on the factors influencing the security evaluation. Referring back to Section 5, recall is the fraction of correctly identified vulnerabilities to all actual vulnerabilities in the ground truth. Precision is the fraction of correctly identified vulnerabilities, according to the ground truth, among all identified vulnerabilities. Figure 5 shows the precision and recall values for the analyzed SBOM generation tools. All the tools show low precision and recall average values, with cdxgen leading the group with 0.17 and 0.21 average precision and recall, respectively.

As shown in Figure 6, the analyzed tools have a very high number of false positives, while false negatives are present in a more manageable magnitude. For example, 99.5% (978 FP / 5 FN) of the misclassified vulnerabilities are false positives for cdxgen and 97.8% (926 FP / 21 FN) for Syft. While an overestimation is generally considered better than losing vulnerabilities [4, 14, 2], these numbers are out-of-scale, causing burdening during vulnerability investigation.

By randomly sampling projects with at least a false positive we identified the following reasons:

1. The SBOM list dependencies that are not collected during installation,
2. the vulnerability is reported with a vulnerability identifier different from the one in the ground truth

The first reason is caused by the presence of metadata files inside of projects listing dependencies that are *not* actually collected during installation. By analysing the dependencies contained in the SBOMs we confirm that on average 75%

**Table 3: Comparison of average values for Jaccard similarity, Precision, and Recall for PIP-SBOM against state-of-the-art tools.**

| | cdxgen | ORT | Syft | Trivy | GH-sbom | **PIP-SBOM** |
|---|---|---|---|---|---|---|
| Jaccard Similarity | 49.77% | 36.50% | 26.33% | 23.63% | 23.98% | **78.39%** |
| Avg Precision | 17.08% | 16.31% | 12.39% | 12.17% | 5.57% | **80.95%** |
| Avg Recall | 21.42% | 19.93% | 15.61% | 14.01% | 8.10% | **80.26%** |
| F. Poss. / F. Negs. | 978/5 | 449/10 | 926/21 | 893/21 | 2793/29 | **47/3** |

of the dependencies listed in the SBOMs are not actually installed. When a package is built only the files specified inside of the metadata file are included in the package (see Section 2.2). Those are the dependencies that the package manager collects during the installation. This misalignment between files listing dependencies, and files used for installing dependencies, causes SBOM generation tools to generate a huge amount of false positives during vulnerability scanning.

The second reason is linked to a limitation of our methodology (see Section 8) and affects a limited number of vulnerabilities making it negligible. However, it highlights the problem of using vulnerability databases as the source for vulnerability scanning. These databases may be partially out-of-date, or experiencing problems in the collection of security advisories. Recently NVD had trouble collecting CVEs for a long time, causing many issues for tools relying on such database [36].

> **TAKEAWAY** (1) SBOM generation tools do not properly support multiple package managers. (2) SBOM generation tools do not consider how dependencies are actually collected by Python's package managers.

### 6.2 RQ2: Trying a Different SBOM Generation Method

Implementing PIP-SBOM, a PIP extension using the dependency resolution algorithm native to the package manager, we drastically improve the vulnerability assessment of the dependency network. As Table 3 reports, almost 80% of the vulnerability reports match the ground truth. *No one of the analyzed pre-existing tools is able to provide the same performance.*

PIP-SBOM achieves a 64% increase in precision and a 59% improvement in recall for vulnerability scans, outperforming the best-performing existing tool.

Having a limited delta between average precision (80.95%) and recall (80.26%), our proposed tool allows a vulnerability scanner to effectively identify vulnerabilities in the dependency network requiring a limited manual effort to discard false positive vulnerabilities. Thus, it has only 47 false positive vulnerabilities. While the false negative vulnerabilities have the same magnitude as the other

SBOM generation tools, false positives are way lower than other tools. By providing developers with a manageable number of vulnerabilities to check, we want to push towards the adoption of SBOMs as a useful means for security.

Concerning projects' security assessment differing from the ground truth: We manually reviewed these differences, and all of them are due to issues with vulnerability identifiers (See Section 8 for details).

> **Takeaway** PIP can be extended by re-using most of its code to generate an SBOM that drastically improves the vulnerability assessment results.

## 7 Discussion

SBOM generation is a hard problem. Since software is usually constructed on third-party components, having a complete and correct SBOM represents a great improvement for security, and many other aspects of software usage — e.g., licensing. Using SBOM as input for vulnerability scanners speeds up the analysis of the software's dependency network, also providing clear information on the vulnerable dependencies and their transitive dependencies. However, state-of-the-art tools do not provide SBOMs that can be efficiently used for vulnerability assessment in the Python ecosystem.

We identified two main causes for this problem:

1. SBOM generation tools do not provide support for the high number of package managers used for Python projects.
2. They do not correctly build the dependency network.

However, it is possible to greatly improve the current situation, without much of an effort. Since dependencies are collected by package managers, using them for SBOM generation represents an efficient approach. We tested this *native* generation method by implementing it in PIP.

### 7.1 Implications

Our results can be interpreted in two ways:

- Developers currently relying on tools like shiftleft scan or Grype are missing out most of the actual vulnerabilities in the dependencies of the analyzed software.
- The problems affecting the SBOM generation tools can be easily solved by adding just some changes to package managers. Once the SBOM is correctly generated, it greatly benefits the vulnerability assessment.

These implications can be easily transferred to other languages and ecosystems. Most sofware ecosystems do not support provenance and SBOM generation. According to OpenSSF only npm ships the generation of SBOM for the

hosted packages, while only npm and homebrew provide provenance information [44].

On the other hand, having a proper SBOM enables great result in the vulnerability assessment. With just a few changes to the package manager, the SBOM can be shipped with a project.

### 7.2 Recommendations

Our recommendations are addressed to communities of software ecosystems. There is a need to include SBOMs as part of projects by default. Thanks to our proposal, we showed that this is a possible and easily achievable goal. The following recommendations may help those communities:

- Using centralized SBOM may provide a common knowledge base, easily accessible, and distributed for all developers. GH-sbom provides such a feature, however, it is not always supported and it is limited to providing SBOM for the latest commit on the main branch. Fostering discussion on having such a powerful tool may relieve some of the efforts on the single communities.
- Push for a standard set of files to list the dependencies installed in a project. Most ecosystems already use this approach, for example, npm (`package.json`), and Cargo (`Cargo.toml`). Multiple data sources may be confusing and can introduce unexpected dependencies and associated vulnerabilities.
- Work for a SBOM generation tool implemented inside of the ecosystem package manager(s). As we showed, SBOM generation can largely be improved by using a native generation method. SBOMs can benefit from ecosystem-specific information that may improve SSC transparency for that specific ecosystem.

## 8 Threats to Validity

*Projects Collection.* Our sample can be generalized to the whole package population on PyPI, with the margin of error stated in Section 3.1. While it is relevant because of generalization, it may miss specific corner cases providing insightful knowledge. We internally cross-validated the sample with other random samples taken from PyPI without filtering on package managers. The cross-validation confirmed the accuracy of our results with an error of $\pm 2\%$.

*SBOM generation tools Selection.* The selection is manually conducted by filtering SBOM generation tools based on the criteria discussed in Section 3.2. The list of tools has been reviewed by more than one author, and we agreed on the five selected tools. However, a degree of subjectivity would be present in the selection, leading to the exclusion of other potentially effective tools. We tested the tools that were eligible for our study and excluded the ones that were not functional for our research goals. The manual testing may have caused the exclusion of potentially effective tools.

*Ground Truth.* The reliance on pip-audit for establishing the ground truth introduces potential limitations, as this tool may not detect all relevant vulnerabilities, which could impact the baseline used for comparing other tools. pip-audit is largely used to detect vulnerabilities in the dependency installed inside of an environment. However, it is subject to vulnerability databases as well as vulnerability scanners. Since our measurements with Grype were conducted at the same time as the ground truth generation, we argue that any potential gap was avoided.

*Evaluation Methodology.* We experienced false positives and negatives while evaluating vulnerability scan results. Some are due to a missing vulnerability identifier inside the vulnerability scan result. These errors can be easily fixed by establishing a unique vulnerability database mapping vulnerabilities to their identifiers on the various databases. We manually fixed the issue in our dataset since such an event is rare.

*Evaluation Scope.* The methodology and results are tailored to Python, while we consider these results extensible to other languages, we cannot state their transferability. Our study wants to raise concerns about the reliability of current SBOM generation tools for security analysis of dependency networks, and to push on the development of native SBOM generation method by package managers.

## 9 Related Works

Research produced a large amount of literature on SBOM in the last period. Related to this work, it can be divided into two categories: research on (1) technical challenges and (2) adoption.

*Technical Challenges* Yu et al. [49] conduct a differential analysis examining the correctness of SBOMs generated by four SBOM generation tools. The analysis is conducted over seven program languages. They highlight how the SBOM generation tools have difficulties to correctly finding dependencies.

Torres-Arias et al. [46] conduct a study on the fulfilment of minimum required elements issued by NTIA [35] for SBOMs using the SPDX standard. Similarly, Halbritter and Merli [28] do the same for CycloneDX SBOMs.

Balliu et al. [19] focus on the Java ecosystem with the analysis of the SBOM generated for a known Java application. They provide an overview of the challenges that SBOM generation tools face to generate an SBOM on Java projects. Similarly, Rabbi et al. [42] focus on the npm ecosystem. Cofano et al. [22] conduct a study on the relationship between the Python ecosystem and four SBOM generation tools. They identify challenges posed by the ecosystem and an excess of approximation by the SBOM generation tools. All these works do not look at the impact of the generated SBOM. Differently, we focused on the usage of the SBOM to understand to what extent this technology can be effectively adopted.

*SBOM adoption* While the SBOM represents a resource for SSC transparency, enabling both functional and security testing, its adoption is delayed. The efficacy of SBOM for security has been recently reported by Sharma et al. [43]. They propose a technology using SBOM to mitigate vulnerabilities affecting Java applications.

Enck et al. [26] report that the use of SBOM for security is debated among practitioners. More than a year later the landscape is not brighter. Zahan et al. [50] report that attended of the S3C2 [10] Industry Summit were sceptical about the adoption of SBOM. The problems come from including SBOM generation in the CI/CD pipeline, however, it has been suggested *to embed SBOM generation within the build template as part of a standardized build pipeline and making SBOM generation a mandatory task when setting up CI/CD templates.* We show that this approach can be easily adopted without breaking the build process, at least concerning the Python ecosystem.

## 10 Conclusion and Future Work

This study shows how SBOM generation heavily affects vulnerability scans of software. The current state-of-the-art SBOM generation tools cannot provide SBOMs accurate enough for vulnerability scanner to identify more than 20% of the vulnerabilities actually present in the software. This problem is due to the lack of support for Python and the techniques adopted to solve versions of Python project dependencies. We proposed PIP-SBOM, a proof of concept implementation extending PIP to generate an SBOM directly from the Python package manager. The performances provided by our PoC for vulnerability assessment suggest that using native resources of the ecosystem — e.g., the package manager — to generate the SBOM may largely improve the overall security posture of software.

Some future works in this direction can support the generation of SBOM in different software ecosystems. Moreover, research in the use of SBOM for security should provide further elements to push software communities and developers to adopt this technology.

## 11 Data Availability

The list of projects, ground truth, generated SBOMs, security reports, and scripts to replicate our results can be found in our external resources at `https://osf.io/9agz7/?view_only=1c4704de735b46de8595c40dfa4fb1ad`.

## Acknowledgments

This work was supported by the European Commission under the Horizon Europe Programme, as part of the project LAZARUS (https://lazarus-he.eu/) (Grant Agreement no. 101070303). 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. bomber: Scans software bill of materials (SBOMs) for security vulnerabilities
2. False negative. https://www.contrastsecurity.com/glossary/false-negative, accessed: 2024-9-5
3. Grype. https://www.cisa.gov/resources-tools/services/grype, accessed: 2024-9-5
4. Intrusion detection. https://owasp.org/www-community/controls/Intrusion_Detection, accessed: 2024-9-5
5. jqlang/jq: Command-line JSON processor. https://github.com/jqlang/jq, accessed: 2024-9-5
6. kubeclarity: KubeClarity is a tool for detection and management of software bill of materials (SBOM) and vulnerabilities of container images and filesystems
7. resolvelib. https://pypi.org/project/resolvelib/, accessed: 2024-9-5
8. sast-scan: Scan is a free & open source DevSecOps tool for performing static analysis based security testing of your applications and its dependencies. CI and git friendly
9. sbom-scorecard: Generate a score for your sbom to understand if it will actually be useful
10. Secure software supply chain center, https://s3c2.org/
11. Versioning - python packaging user guide. https://packaging.python.org/en/latest/discussions/versioning/, accessed: 2024-9-3
12. Why chainguard uses grype as its first line of defense for CVEs. https://www.chainguard.dev/unchained/why-chainguard-uses-grype-as-its-first-line-of-defense-for-cves, accessed: 2024-9-5
13. Using grype to scan container images for vulnerabilities. https://edu.chainguard.dev/chainguard/chainguard-images/working-with-images/scanners/grype-tutorial/ (Jan 1), accessed: 2024-9-5
14. False positives and false negatives in information security. https://www.guardrails.io/blog/false-positives-and-false-negatives-in-information-security/ (Aug 2022), accessed: 2024-9-5
15. Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In: Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses. pp. 29–37. ACM, Copenhagen Denmark (Nov 2023). https://doi.org/10.1145/3605770.3625213, https://dl.acm.org/doi/10.1145/3605770.3625213
16. Dependency Graph SBoM export for older repository versions? (2024), https://github.com/orgs/community/discussions/118612
17. Agency, C..I.S.: SBOM FAQ (2024), https://www.cisa.gov/resources-tools/resources/sbom-faq
18. Anchore: Grype, https://github.com/anchore/grype/
19. Balliu, M., Baudry, B., Bobadilla, S., Ekstedt, M., Monperrus, M., Ron, J., Sharma, A., Skoglund, G., Soto-Valero, C., Wittlinger, M.: Challenges of producing software bill of materials for java. IEEE Security & Privacy (2023)
20. Bi, T., Xia, B., Xing, Z., Lu, Q., Zhu, L.: On the way to sboms: Investigating design issues and solutions in practice. ACM Transactions on Software Engineering and Methodology **33**(6), 1–25 (2024)

21. Cass, S.: The Top Programming Languages 2024 (2024), https://spectrum.ieee.org/ibm-quantum-computer-2668978269
22. Cofano, S., Benedetti, G., Dell'Amico, M.: SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis (2024), https://arxiv.org/abs/2409.01214
23. Deepbits: Evaluating and Benchmarking SBOM Generators: A Systematic Approach (2023), https://www.deepbits.com/whitepaper/1
24. pip developers: More on Dependency Resolution (2024), https://pip.pypa.io/en/stable/topics/more-dependency-resolution/
25. Dietrich, J., Rasheed, S., Jordan, A., White, T.: On the security blind spots of software composition analysis (2023)
26. Enck, W., Williams, L.: Top five challenges in software supply chain security: Observations from 30 industry and government organizations. IEEE Secur. Priv. **20**(2), 96–100 (Mar 2022)
27. Guo, W., Xu, Z., Liu, C., Huang, C., Fang, Y., Liu, Y.: An empirical study of malicious code in pypi ecosystem. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 166–177. IEEE (2023)
28. Halbritter, A., Merli, D.: Accuracy evaluation of SBOM tools for web applications and system-level software. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. ACM, New York, NY, USA (Jul 2024)
29. Hashemi, M.: Calendar versioning — CalVer. https://calver.org/, accessed: 2024-9-3
30. Hejderup, J., Beller, M., Triantafyllou, K., Gousios, G.: Präzi: from package-based to call-based dependency networks. Empirical Software Engineering **27** (9 2022). https://doi.org/10.1007/s10664-021-10071-9
31. JR., J.R.B.: Executive Order on Improving the Nation's Cybersecurity (2021), https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/
32. Ladisa, P., Plate, H., Martinez, M., Barais, O.: Sok: Taxonomy of attacks on open-source software supply chains. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1509–1526. IEEE (2023)
33. Merrill, K., Newman, Z., Torres-Arias, S., Sollins, K.R.: Speranza: Usable, privacy-friendly software signing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3388–3402 (2023)
34. Network, E., Agency, I.S.: Enisa threat landscape 2021 (2021), https://www.enisa.europa.eu/publications/enisa-threat-landscape-2021
35. NTIA: The minimum elements for a software bill of materials (SBOM)
36. Ozkan, S.: NVD leaves thousands of vulnerabilities without analysis data (2024), https://securityscorecard.com/blog/national-vulnerability-database-nvd-leaves-thousands-of-vulnerabilities-without-analysis-data/
37. Peisert, S., Schneier, B., Okhravi, H., Massacci, F., Benzel, T., Landwehr, C., Mannan, M., Mirkovic, J., Prakash, A., Michael, J.B.: Perspectives on the solarwinds incident. IEEE Security & Privacy **19**(2), 7–13 (2021)
38. Plate, H., Ponta, S.E., Sabetta, A.: Impact assessment for vulnerabilities in open-source software libraries. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 411–420 (2015). https://doi.org/10.1109/ICSM.2015.7332492
39. Ponta, S.E., Plate, H., Sabetta, A.: Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 449–460 (2018). https://doi.org/10.1109/ICSME.2018.00054

40. Ponta, S.E., Plate, H., Sabetta, A.: Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empirical Software Engineering **25**, 3175–3215 (9 2020). https://doi.org/10.1007/s10664-020-09830-x
41. Preston-Werner, T.: Semantic versioning 2.0.0. `https://semver.org/`, accessed: 2024-9-3
42. Rabbi, M.F., Champa, A.I., Nachuma, C., Zibran, M.F.: Sbom generation tools under microscope: A focus on the npm ecosystem. In: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. pp. 1233–1241 (2024)
43. Sharma, A., Wittlinger, M., Baudry, B., Monperrus, M.: Sbom.exe: Countering dynamic code injection based on software bill of materials in java (2024), `https://arxiv.org/abs/2407.00246`
44. Steindler, Z.: How to Make Programming Language Package Repositories More Secure (2024), `https://openssf.org/blog/2024/07/31/how-to-make-programming-language-package-repositories-more-secure/`
45. Synopsis: 2023 ossra report, `https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2023.pdf`
46. Torres-Arias, S., Geer, D., Meyers, J.S.: A viewpoint on knowing software: Bill of materials quality when you see it. IEEE Secur. Priv. **21**(6), 50–54 (Nov 2023)
47. Wallen, J.: Scan container images for vulnerabilities with grype. `https://thenewstack.io/scan-container-images-for-vulnerabilities-with-grype/` (May 2022), accessed: 2024-9-5
48. Wermke, D., Klemmer, J.H., Wöhler, N., Schmüser, J., Ramulu, H.S., Acar, Y., Fahl, S.: " always contribute back": A qualitative study on security challenges of the open source supply chain. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1545–1560. IEEE (2023)
49. Yu, S., Song, W., Hu, X., Yin, H.: 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). pp. 29–36 (2024). https://doi.org/10.1109/DSN58291.2024.00018
50. Zahan, N., Acar, Y., Cukier, M., Enck, W., Kastner, C., Kapravelos, A., Wermke, D., Williams, L.: S3C2 summit 2023-11: Industry secure supply chain summit (Aug 2024)