IMT School for Advanced Studies, Lucca
Lucca, Italy

Transparent Dependencies: Improving Software Supply Chain Visibility at Build Time and Runtime

PhD Program in Cybersicurezza

Track in Software, System, and Infrastructure Security

XXXVIII Cycle

By

Serena Cofano

2026

The dissertation of Serena Cofano is approved.

PhD Program Coordinator: Marco Tribastone, IMT School for Advanced Studies Lucca

Advisor: Prof. Matteo Dell'Amico, Università degli Studi di Genova

The dissertation of Serena Cofano has been reviewed by:

Dr. Serena Elisa Ponta, SAP Research France

Prof. Wojciech Mazurczyk, Warsaw University of Technology

IMT School for Advanced Studies Lucca
2026

Contents

List of Figures

List of Tables

Abstract

The increasing complexity of modern software systems has intensified reliance on third-party and open-source components, expanding the Software Supply Chain (SSC) and its vulnerability to attacks. Ensuring transparency in the Software Supply Chain is essential to assessing security risks and responding to incidents. The Software Bill of Materials (SBOM) has emerged as a fundamental tool to list all components of a software product and support vulnerability assessment. However, current SBOM generation tools often produce incomplete or inaccurate outputs, especially in ecosystems like Python, where the lack of standardized metadata formats and dependency specifications hinders correctness and completeness. Moreover, most SBOM generation approaches operate statically, which leads to missing runtime dependency information. This information is fundamental to determining the actual usage of dependencies included in the software. This poses a limitation in security management tasks. It makes maintainers waste time assessing unreachable dependencies, delaying responses to real security issues.

This Thesis advances Software Supply Chain transparency in both static and dynamic contexts through two main contributions. First, it presents a systematic evaluation of SBOM generation tools in the Python ecosystem, identifying ecosystem- and tool-related issues that degrade SBOM quality and consequently affect vulnerability assessment. Building on these findings, it introduces PIP-SBOM, a solution based on the `pip` package installer that achieves higher precision than the best existing tool. Second, it proposes a novel approach for runtime dependency introspection in Java, Classport, which em-

beds dependency metadata into binaries and retrieves it at runtime with negligible performance overhead.

Evaluations on real-world projects demonstrate the feasibility, accuracy, and efficiency of the proposed approaches, providing actionable insights and novel techniques to enhance Software Supply Chain transparency and security in both build-time and runtime scenarios.

Chapter 1

Introduction

Software has become more complex in recent years. The demand for richer features makes it unfeasible to develop applications entirely from scratch, driving widespread use of third-party and open-source components [56]. The collection of all such components, actors, tools, and platforms involved in building and delivering a final product forms the Software Supply Chain [57]. While this approach accelerates development, it also increases the attack surface. In general, a Software Supply Chain attack is defined as an attack that occurs on a component and reflects on the final software, which can be an intentional or an unintentional target for the attacker. Many attacks on the Software Supply Chain have been registered in recent years. Notable Software Supply Chain incidents include the Log4Shell vulnerability [64], the Codecov compromise [54], and the xz catastrophe [42].

Governments and industry respond with regulatory and technical measures. A key emerging concept is *transparency*, i.e., the ability to identify and inspect all components in the Software Supply Chain. In this context, the Software Bill of Materials (SBOM) has become a cornerstone. The Software Bill of Materials (SBOM) is a document enumerating all components in a software product [10]. By correlating the SBOM with the Common Vulnerabilities and Exposures (CVE) database [29], i.e., a publicly available index of reported software vulnerabilities, secu-

rity teams can detect vulnerabilities in their code and take early action to patch or replace the infected components. SBOMs also help assess supply chain risks by making it clear which third-party or open-source elements are present, enabling better evaluation of their trustworthiness and security posture. During a security incident, such as the occurrence of a zero-day vulnerability, an SBOM enables rapid identification of whether and where the vulnerable component is used, which shortens response times and limits potential damage. When integrated into the development pipeline, SBOMs support a secure software development lifecycle by allowing teams to detect insecure or unapproved components before release. Also, even after deployment, SBOMs are useful for continuous monitoring, as they allow organizations to stay informed when new vulnerabilities are discovered in components they've already shipped. The U.S. Executive Order on Improving the Nation's Cybersecurity [27] (2021) and the EU Cyber Resilience Act [18] (2024) make SBOMs mandatory in specific contexts. Also, non-profit organizations such as OpenSSF are working toward the implementation of SBOM as a standard in the most used software ecosystems.[1]

However, to be useful, an SBOM has to be *complete*, i.e., to include all the components present in the Software Supply Chain, and *correct*, i.e., to include the correct information and not have false positives. These two properties are essential to avoid the occurrence of false negatives and false positives during security analysis, respectively. Existing tools that generate the SBOM often fail to meet these criteria. Studies report that false positives and negatives undermine practitioners' trust in SBOMs [60, 66, 69]. 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 [3] or Javascript [47]. 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:

[1] `https://openssf.org/`

Yu et al. [68] 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 the causes that lead to different tools producing different SBOMs for the same piece of software. 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.

Most of the approaches to SBOM operate statically, by looking for dependencies in metadata files, such as manifests or lockfiles, only at build time [47, 68]. This has two fundamental limitations. First, it does not reflect the actual dependency usage at runtime, i.e., whether a dependency is actually loaded or used in production [3]. Even if this does not present a direct risk for the application, it makes maintainers and developers waste a lot of time doing unnecessary work by assessing the security of unreachable dependencies, which slows down the response time to the actual security problems. Second, static analysis of dependencies does not support the maintainers or developers to make runtime decisions based on the actually used dependencies. Basically, the fundamental limitation of SBOMs is that all dependency information is lost at runtime.

Thus, achieving Software Supply Chain transparency requires addressing two complementary challenges. The first concerns the static scenario, where it is necessary to improve the completeness and correctness of SBOM generation so that build-time dependency visibility is reliable. The second concerns the dynamic scenario, which requires enabling accurate introspection of the dependencies that are actually used during runtime.

In summary, the Software Supply Chain is growing very fast, which increases the risk of introducing vulnerable components. It is necessary to know which components are present in the software to determine their security and to react to possible attacks. Securing each component means securing the entire supply chain. SBOM presents a valuable tool to enhance transparency of the Supply Chain. However, this tool is not yet mature and it lacks research studies on it. Moreover, it is not suitable to support runtime scenarios, leading to the misidentification of the components in this situation.

1.1 Thesis Contribution

This Thesis advances Software Supply Chain security by addressing transparency in both static and runtime contexts. The contribution can be summarised as:

1. *SBOM generation study and improvement*: an evaluation of SBOM generation tools in the Python ecosystem, assessing completeness, i.e., *all* the dependencies of the distributed artifact are included in the SBOM, and correctness, i.e., the components of the SBOM are correctly identified by name and version, identifying ecosystem- and tool-specific causes of deficiencies, and analyzing the impact of poor SBOM quality on vulnerability assessment. Based on these findings, we propose PIP-SBOM, a technique based on the `pip` Python package installer that overcomes major limitations of existing tools.

2. *Runtime dependency introspection*: the design and implementation of Classport, a novel Java-based method to embed dependency metadata into binaries and retrieve it at runtime with low performance overhead.

1.2 Thesis Structure

The Thesis is structured into two chapters, which reflect the chapters of the PhD. They are both structured with a background section, which provides the reader with context and a basic understanding to proceed with the reading. Then, the contribution is presented in a format strongly inspired by the papers we produced; for the list of included papers and their role in the Thesis, see Section 1.3. Each chapter includes a related work section, which positions our contribution with respect to the current state-of-the-art. Additionally, a summary is provided at the end of each chapter. Finally, the Thesis presents the overall conclusion.

1.3 Papers Included in the Thesis

The Thesis is inspired and driven by three scientific papers that are the output of our research.

The included papers are the following:

1. Serena Cofano, Giacomo Benedetti, Matteo Dell'Amico, "SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis", *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, `https://doi.org/10.1109/TrustCom63139.2024.00077`
   Summary: in this paper, we study the SBOM generation tools for Python, investigating their ability in identifying dependencies and the causes that lead to an incomplete and incorrect generation of the SBOM. This study reveals the immaturity of the SBOM generation tools analyzed and highlights the need for new approaches in SBOM generation.
   Contribution to the Thesis: this paper contributes to Chapter 2, in particular to Section 2.2.

2. Giacomo Benedetti, Serena Cofano,[2] Alessandro Brighente, Mauro Conti, "The Impact of SBOM Generators on Vulnerability Assess-

[2]The second and the first authors contributed equally to the paper.

ment in Python: A Comparison and a Novel Approach", *International Conference on Applied Cryptography and Network Security* https://doi.org/10.1007/978-3-031-95764-2_19

Summary: in this paper, we evaluate the impact an incomplete and incorrect generation of the SBOM has on the vulnerability assessment based on that SBOM. We find a large number of false positives and negatives in vulnerability detection, and this motivates us to develop a novel approach for Python to generate SBOM. By leveraging most of the logic which Python uses to solve dependencies, we develop a new tool, PIP-SBOM, and compare it with the state-of-the-art tools. This reveals that our implementation outperforms the existing ones, leading to a better vulnerability assessment.

Contribution to the Thesis: this paper contributes to Chapter 2, in particular to Section 2.3 and in particular to Section 2.4..

3. Serena Cofano, Daniel Williams, Aman Sharma, Martin Monperrus, "Classport: Designing Runtime Dependency Introspection for Java", the paper has been submitted to the Software Engineering in Practice (SEIP) track of the International Conference of Software Engineering (ICSE), and it is currently under revision.

   Summary: in this paper, we propose Classport, a tool that enables runtime identification of dependencies in Java. It brings static available information at runtime, allowing runtime dependency introspection.

   Contribution to the Thesis: this paper contributes to Chapter 3.

Chapter 2

Software Bill of Materials: Concepts, Limitations, and Improvements

The SBOM is a powerful artifact to enhance the security of the Software Supply Chain. Its ability to collect all the dependencies present in the software, comprising the transitive ones, promotes the transparency of the entire Software Supply Chain of a given software. This is fundamental for enabling many security practices, such as vulnerability assessment, which determines the security of each component, identifying known vulnerabilities.

However, the tools that generate the SBOM are not mature and they are perceived as not reliable by practitioners, resulting in a low adoption [17]. Moreover, a few studies have been done on the precision of SBOM generation tools [3, 47, 28] and a few measurements have been implemented.

In this Chapter, we describe our contribution in studying and facing the issues related to the SBOM.

In detail, the three main goals are: *(i)* determining the performance of the state-of-the-art SBOM generation tools for Python and investigating the causes that lead to an incomplete or incorrect SBOM generation,

(ii) studying the impact that the generation of the SBOM has on the vulnerability assessment, and *(iii)* proposing a novel technique to generate the SBOM and provide the implementation.

All the contributions of this chapter are related to the Python ecosystem. Focusing on a single ecosystem is fundamental in studies on the SBOM, because dependency management varies significantly depending on the ecosystem. The choice of Python is driven by the following reasons:

- its widespread use [59, 14, 15].
- 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.
- 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.

In the first contribution, we identify problems in SBOM generation and determine their root causes in both ecosystem and SBOM generation tools. Our study focuses on the elements that primarily affect SBOM creation in Python projects in terms of correctness and completeness. Specifically, we perform a study on the involvement of the Python ecosystem in 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. In this contribution, we focused on static generation methods, since they are relevant in

specific context where the generation of the SBOM should not affect the performances of the overall process, e.g., microservice context with limited memory and computing resources. Moreover, generating an SBOM from the source code enables developers and maintainers to have an artifact reflecting not a specific configuration in a defined environment, but the status of the package at source code level. For this goal, we create a set of Python projects with the same dependencies but using different package managers. We also focus on how the approach used by SBOM generation tools changes the final output of the SBOM. In detail, 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. By analyzing the causes of such differences, we obtain two main takeaways: the lack of standards in the Python ecosystem and the defects in SBOM generation tools. In particular, this analysis allowed us to provide two key recommendations: *(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.

In the second contribution, following the SBOM generation tools assessment, we study the impact of a bad generation on the vulnerability assessment conducted using the generated SBOM. In this case, we relaxed the generation method constraint, and considered both tools relying on static elements, such as metadata parsing, and dynamic ones, such as environment analysis. We evaluate how the representational capabilities of SBOM generation tools impact the identification of known vulnerabilities in the Software Supply Chain. 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. We use the SBOMs they produce as input to a vulnerability scanner. Then, we compare the vulnerabilities found starting from SBOMs generated by different SBOM generation tools and we compare them with the ground truth. Precision, recall, and Jaccard similarity computed between the ground truth and the generated vulnerability reports highlight poor reliability on vulnera-

bility assessment based on SBOMs.

In the third contribution, to solve the issues of SBOM generation tools and improve the security posture of the Software Supply Chain we propose PIP-SBOM, a novel `pip`-based solution. Our solution overcomes the most relevant issues of SBOM generation tools, 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 Software Supply Chain.

We summarize our contributions as follows.

- We investigate the completeness and correctness of SBOM generation tools operating with static approaches, and we analyze the causes and the consequences of the lack of these two properties.

- We evaluate the capabilities of SBOM generation tools (both static and dynamic) in helping increase the Software Supply Chain 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, with respect to a ground truth.

- 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.

The following sections first give an overview of the main background concept of the Chapter, i.e., notions about the SBOM, the bases of dependencies management in Python, and how the vulnerability assessment with SBOM works. Then, there are three sections about the three main contributions of the thesis related to SBOM. Finally, the work is positioned with related work, and a summary is given to the reader.

2.1 Background

This section provides the necessary concepts to dive into the current Chapter. In particular, it gives an overview of the SBOM, on what it is and its usage and history, then it focuses on the Python ecosystem, showing the basic concepts of dependency management. Finally, it is detailed how the SBOM is used in the context of vulnerability assessment.

2.1.1 Software Bill Of Material

The SBOM is an inventory of software components and dependencies, information about those components, and their hierarchical relationships [55]. There are multiple standards for SBOMs; the most used are CycloneDX [34] and SPDX [21]. 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 [5].

The CycloneDX standard supports JSON, XML, and Protobuf formats as output [35]. 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 in a universal and uniform way across programming languages, package managers, packaging conventions, tools, APIs, and databases [38].

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 rather on a simulated one, which may occur at a different time than 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.

2.1.2 Dependency Management in the Python Ecosystem

PyPI[1] 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 [7], 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.

[1] `https://pypi.org/`

The metadata files include dependencies specified with versions in various formats [11]. 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 range of possible versions.

Constrained dependencies can have different clauses: the compatible release clause "~=",[2] 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 with the core functionalities. Optional dependencies are not fundamental to the correct functioning of the project but are either used during the development phase (*development* dependencies), or are related to additional features that can be made available to the user. The final user of a Python project can decide whether to install any optional dependencies.

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. It's important to choose the back-end that aligns with the selected front-end, 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

[2] `https://packaging.python.org/en/latest/specifications/version-specifiers/#compatible-release`

other than PyPI.

2.1.3 Vulnerabilities Scanning with SBOM

A *vulnerability scanner* is a tool that analyzes a software artifact and provides a *security report* listing potential vulnerabilities affecting the scanned product. There exist vulnerability databases, such as NVD (National Vulnerability Database) [61] and OSV (Open Source Vulnerabilities) [24], which 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 [50], Grype [2], KubeClarity [31], and Bomber [16] — 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 component names and retrieve their security information from public databases (e.g., NVD).

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

We use Grype to obtain security reports from SBOMs. A Grype 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.

```
{
  "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" }
  }]
}
```

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

- *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 security report for a Python SBOM.

2.2 Analyzing the Causes of Incomplete and Incorrect SBOM Generation

In this section, we illustrate how we conduct our study on the static generation of SBOMs, by evaluating their completeness and correctness in the context of Python. We provide the results of our study and a discussion. Our goal is to identify the issues with the generation of the SBOM and determine their relation with the SBOM generation tool or with the ecosystem.

Table 1: Back-ends (BE) and front-ends (FE) selected for python projects generation

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

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.

2.2.1 Experimental Setup

This step consists of creating a dataset of Python projects and selecting SBOM generation tools.[3]

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;

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

(ii) do not have a way to create a trusted ground truth for the components of a large existing project due to the diversity of build tools and development practices [68].

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 managing Python projects.

Due to these criteria, we select *Setuptools* and *Poetry*, which are the most used tools [43]. Moreover, we select *Pdm*, *Flit*, *Pipenv*, and *Hatch*, which are less used, but are cited in articles about package managers for Python [44, 6, 45, 23]. We end up with 5 back-ends and 5 front-ends. Of the 25 possible combinations, 12 are compatible, as shown in Table 1. We analyze all of them.

Next, we select the dependencies to add to each project. The purpose is to study the behavior of SBOM generation tools 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

Table 2: Set of dependencies declared inside each project in the sample.

Dep.	Ver. Spec.	Type	Imp. in Code	Trans. Deps.
numpy	Unversioned	PyPI (default)	Yes	0
black	21.10b0 (Git reference)	Git repository	No	0
docopt	0.6.2 (direct URL)	PyPI tarball URL	No	0
seaborn	0.12.2 (pinned)	PyPI	No	6
matplotlib	>3.9.0 (constrained)	PyPI	No	12
urllib3	Unversioned	PyPI (default)	No	8

allows us to choose the dependencies from Table 2. Only one dependency, Numpy, is imported and used in the code. This helps us understand whether there is a difference 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.
2. Add the selected dependencies through the commands available with the selected front-end. If generated by the front-end, the lockfile is included in the dataset. At this point the project contains the metadata files reported in Table 3.
3. Add a main file with example code importing one of the dependencies, 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). Neither is included in the final dataset.

Table 3: Configuration files used by common Python package managers

Tool	Main Config File(s)	Lock File	Uses pyproject.toml
Pip	requirements.txt	None	Optional (build only)
Hatch	pyproject.toml	Optional	Yes
PDM	pyproject.toml	pdm.lock	Yes
Pipenv	Pipfile	Pipfile.lock	No
Poetry	pyproject.toml	poetry.lock	Yes

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,[4] we filter for those having:

- an open-source implementation. It is necessary to inspect the code of the tool;
- 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);
- as input the directory containing the source code. Thus, we do not consider tools that require installing the project or container images;
- support for recent CycloneDX versions (at least version 1.3).

After the filtering, we select a set of four tools: *Trivy*, *Syft*, *Cdxgen*, and *Ort*.[5]

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 2.2.4.

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 with parsing metadata files. To evaluate its static behavior, we focus only on the metadata parsing method.

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

[5] During interaction with the Reviewers of this Thesis, we discovered that an additional tool, cyclonedx-python, satisfies the above requirements. We discuss this in Section 2.2.5, and provide results in Appendix B.

Table 4: Selected SBOM generation tools and their Python package manager support

Tool	Tool ver.	CycloneDX ver.	Generation method	Python package manager support
Trivy	0.49.1	1.5	metadata parsing	pip, Pipenv, Poetry
Syft	1.0.1	1.5	metadata parsing	pip, Pipenv, Poetry, PDM
Cdxgen	10.2.2	1.5	metadata parsing	pip, Pipenv, Poetry, PDM, Hatch
Ort	17.0.0	1.5	metadata parsing, PyPI querying	pip, Pipenv, Poetry

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.

The main characteristics of the tools are summarized in Table 4.

2.2.2 SBOM Generation

We generate SBOMs for each project in our sample. We run the SBOM generation tools according to the documentation available online, and to our requirements, i.e., consuming a Python project from a directory in the filesystem. Syft and Trivy run on the host machine, while cdxgen and ORT run on a Docker container. We obtain a total of 48 SBOMs.

2.2.3 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., to how the projects are structured), or to the SBOM generation tools (because of bugs and/or root causes of the methodology used).

Table 5: 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	▲*	▲	▲●■	▲⧗●■	▲⧗●■
Implement searching of 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	-	-	-	⧗■	-

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. To perform this task, since for unpinned dependencies we cannot determine a correct version, we compare the version numbers output by different SBOMs.

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 2.2.4.

2.2.4 Results

We present identified issues and their causes, differentiating between ecosystem and SBOM generation tools origins. Table 5 gives an overview of our findings, showing how SBOM generation tools behave based on the package manager. Appendix A provides a table summarizing the ground truth and a table for each SBOM generation tool involved in our analysis, reporting the number of observed components, the number of expected ones, and the number of false positives and false negatives. In detail, the rows represent the characteristics of the SBOM generation tools that we collect from our experiments. The columns report the front-ends and the symbols represent the SBOM generation tools. For example, cdxgen can find direct dependencies across all projects, while Trivy fails on Hatch and Pdm. The "`NA`" and the "`-`" notations mean respectively that no evaluations can be done for any tools because the characteristic is not present for the given front-end, and that no tools handle the situation described. For example, the implementation of a parsing technique for the requirements.txt cannot be evaluated in cases other than Pip, because other package managers do not include this file. Also, the implementation of parsing the `pyproject.toml`, could be done for every front-end, but only Trivy and ORT do this for Poetry projects.

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 6 illustrates how issues within these categories impact the completeness and correctness of SBOMs across the ecosystem (E) and the SBOM generation tools (T).

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 2.1.2, 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 [68], 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.

Metadata Files Management Issues

Metadata files management includes identifying them, 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.

Table 6: 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.

Lockfiles Most SBOM generation tools rely on the lockfile to collect dependencies. As discussed in Section 2.1, a lockfile is supposed to be the most complete representation of a 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 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 lockfiles, 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`", as declared in the tool's documentation.[6] However, SBOM generation does not work for Pipenv due to a known implementation issue.[7] Specifically, when ORT tries to resolve a remote dependencies from the lockfile of Pipenv, the entire process crashes, and the generation of the SBOM fails. As seen in the following and in Table 5, the lockfile is the only file considered by ORT to generate an SBOM for Pipenv.

`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 SBOM generation tools 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 [7, 8], some tools still support only `setup.py`, resulting in incomplete SBOMs for new projects.

[6] https://oss-review-toolkit.org/ort/docs/tools/analyzer
[7] https://github.com/aboutcode-org/python-inspector/issues/11

`requirements.txt` All of the tools we analyze are able to 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.

Except for the lockfile, matadata files, including `requirements.txt`, contain only direct dependencies by default (E4). Consequently, tools relying on static parsing of metadata files miss 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 (1) wrapped by the URL or (2) listed by field, depending on the lockfile format. Considering these two cases: (1) When SBOM generation tools have to extract the remote dependency information out of the URL, they often fail to correctly parse the URL (T5), leading to missing versions and incomplete dependencies. (2) 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. 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.

2.2.5 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 2.2.4: (1) The lack of standards in the Python ecosystem, and (2) the approximation of dependency solving of SBOM generation tools. From them, we provide consequent recommendations.

As the Reviewers of Thesis highlighted, CycloneDX-python (that was not included in our analysis) is also compliant with the requirements of the study. For completeness, we included the results obtained with the same methodology used for the other SBOM generation tools on CycloneDX-python in Appendix B.

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

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).

2.3 Assessing the Impact of SBOM Quality on Vulnerability Detection

Our previous contribution reveals a lack of maturity in SBOM generation tools. Recall that an SBOM can be used for security assessment of software by feeding a vulnerability scanner with it. The security scanner

uses vulnerability databases to scan if the given SBOM lists components with a known vulnerability.

This leads us to investigate the impact that an incomplete or incorrect generation of an SBOM has on the vulnerability assessment that relies on that SBOM. The analysis focuses on two aspects. First, it examines how much the output of a specific vulnerability scanner differs from the ground truth when it relies on SBOMs produced by different state-of-the-art generation tools. This measurement can be quantified by computing the Jaccard similarity. Second, it explores the reasons for such variation, assessing how variations in completeness and correctness of SBOM affect the reliability of vulnerability detection. In particular, computing the false positives, false negatives, precision, and recall.

In this section, we detail how we conduct our study and the main achievements.

2.3.1 Experimental Setup

In order to carry out the study, we create the following experimental setup. First, we *collect Python projects* where to do the vulnerability assessment. Then, we *select the SBOM generation tools* that have to generate the SBOMs. Finally, we generate a *ground truth* of the vulnerabilities present in the collected projects.

Projects Collection

As discussed in Section 2.1.2, Python supports multiple package managers, each handling dependencies differently. Since SBOM generation tools rely on specific project metadata to generate the SBOM, it is important to understand how these tools behave under different package managers.

To identify the most commonly used package managers, we analyzed 1,351 randomly selected Python packages from the whole population list on `ecosystem.ms`. We excluded any packages whose repositories did not clearly indicate the package manager in use. The resulting distribution is shown in Table 7.

Table 7: 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 then constructed a second sample of 1,000 packages using the following steps:

(1) Randomly select a package from PyPI;

(2) Identify the package manager it uses;

(3) If the quota for that package manager has not been reached, include it in the sample; otherwise, discard it and repeat the process.

This approach ensures that the sample mirrors the real-world distribution of package managers. Based on standard sample size calculations, the final sample supports generalization to the entire Python ecosystem with a 3.04% margin of error at a 95% confidence level.

SBOM generation tools selection

To select the SBOM generation tools for our study, we follow this process:

(1) We scrape the CycloneDX Tool Center[8] and collect a list of 169 open-source tools.

(2) We filter this list to include only tools that *(i)* generate SBOMs, *(ii)* support Python, and *(iii)* offer a command-line interface. This reduces the list to 24 candidates.

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

(3) We manually test the 24 tools. We exclude those that fail to run, require unsupported technologies (e.g., build-root), or have not been maintained in the past year.

From the survivors of our filtering, we selected five tools: cdxgen, GH-sbom, ORT, Syft, and Trivy. In contrast to the previous study, we introduce a new tool, GH-sbom. This tool plays a role in the vulnerability assessment pipeline, but it is executed within the GitHub repository of the project being analyzed. In our earlier contribution, we apply the SBOM generation tools to a synthetic dataset, which means that the projects do not have a corresponding GitHub repository.

Table 8 summarizes these tools, including their SBOM generation methods and examples of vulnerability scanners that use them. We use the SBOMs produced by these tools to evaluate how suitable they are for dependency-based security assessments.

Table 8: 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 [25]
Syft	Metadata Based	Grype, KubeClarity
Trivy	Metadata Based	KubeClarity
ORT	Metadata Based	NA
GH-sbom	Dep. Graph Based	NA

Security Report Ground Truth

To check if SBOM generation tools can find the right vulnerabilities, we first need to know which vulnerabilities actually affect each project. We follow an automated process for each package in our sample to obtain the list of known vulnerabilities for each project.

(1) Retrieve the package's project from its code repository;

(2) Parse metadata files to identify 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) Audit the requirements.txt using `pip-audit`;

(6) Collect the resulting security report.

The reliance on pip-audit for establishing the ground truth introduces potential limitations, as this tool may not detect all relevant vulnerabilities, potentially impacting the baseline used for comparing other tools. pip-audit is largely used to detect vulnerabilities in the dependencies installed inside an environment. However, it is affected by the incompleteness of vulnerability databases, a limitation that also applies to vulnerability scanners relying on them. We mitigated this gap by conducting our measurements with the vulnerability scanner at the same time as the ground truth generation.

2.3.2 SBOMs and Security Reports Generation

To generate the SBOMs, we run our selected SBOM generation tools on the packages in our dataset. We then use `jq` [26] to parse the output and check that each SBOM has the correct format. We use Grype [2] to generate security reports from the SBOMs. We acknowledge that other tools are available, such as OWASP Depscan and the Advisor module included in ORT, which would have produced a slightly different outcome. However, for our purposes, we only need a tool that can read an SBOM and check it against known vulnerability databases. The usage of different vulnerability assessment tools would have influenced the vulnerability assessment outcome, making it difficult to evaluate the effects of different SBOM generation methodologies on the vulnerability assessment. Grype does this by querying multiple databases and cross-checking the results, making it well-suited for our needs.

Each security report from Grype lists the vulnerabilities it finds. We compare this list with the ground truth to evaluate how accurate each SBOM generation tool is.

2.3.3 Analysis

Our goal is to understand to what extent the SBOM impacts the security analysis tool output.

We evaluate the vulnerabilities identified from the SBOMs generated by each tool against a ground truth obtained with `pip-audit`. To quantify the overlap between the two sets of vulnerabilities, we use the Jaccard similarity index. For each tool and for each SBOM $\mathcal{S}$, we first collect the security reports produced from $\mathcal{S}$ and extract the matched vulnerabilities (`ToolVulns`); we then retrieve the vulnerabilities included in the ground truth for the corresponding project (`GrTrVulns`); finally, we compute the Jaccard similarity index between the two sets, as defined in Equation (2.1).

$$J(\text{ToolVulns}, \text{GrTrVulns}) = \frac{|\text{ToolVulns} \cap \text{GrTrVulns}|}{|\text{ToolVulns} \cup \text{GrTrVulns}|} \tag{2.1}$$

While this metric provides a useful indication of how well the vulnerabilities identified by a tool align with those of the ground truth, it does not explain the reasons behind the observed performance. To capture these details, we also compute false positives, false negatives, precision, and recall. Precision measures the proportion of correctly identified vulnerabilities with respect to all vulnerabilities reported by a tool, as shown in Equation (2.2), while recall quantifies the proportion of correctly identified vulnerabilities with respect to all actual vulnerabilities in the ground truth, as shown in Equation (2.3).

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Together, these metrics provide a more comprehensive view of each tool's performance. Precision reflects the trustworthiness of the vulnerabilities identified by the tool, whereas recall highlights the extent to which a tool enables effective security assessment by correctly detecting relevant vulnerabilities.

2.3.4 Results

Our analysis shows that the approach taken to generate an SBOM has a significant impact on the outcome of vulnerability scans. None of the evaluated tools enables the correct identification of all vulnerabilities in more than 20% of the cases, with the sole exception of cdxgen, which reaches nearly 40%. The distribution of Jaccard similarity values in Figure 2 highlights how strongly vulnerability detection depends on the quality of the generated SBOM, confirming that current tools hinder accurate security analysis.

Among the analyzed tools, cdxgen provides the most reliable results. This is due to its strategy of installing dependencies in a virtual environment and its broad support for multiple package managers. Both aspects prove critical for obtaining a faithful representation of the Software Supply Chain. Tools lacking one or both features show worse performance: for instance, ORT simulates installation by querying PyPI but supports only a limited set of package managers; Syft and Trivy rely exclusively on static metadata without installing dependencies, which leads to lower accuracy; and GH-sbom is strongly influenced by repository configuration and restricted to the last commit on the main branch, preventing reproducibility across different commits or tags.[9]

Figure 3 reports the precision and recall values, further clarifying the factors behind these results. All tools exhibit low averages, with cdxgen again performing best (0.17 precision and 0.21 recall). Figure 4 shows that the main limitation is the overwhelming number of false positives: for example, 99.5% of the misclassified vulnerabilities for cdxgen and 97.8% for Syft are false positives. Although overestimation is usually

[9] https://github.com/orgs/community/discussions/118612

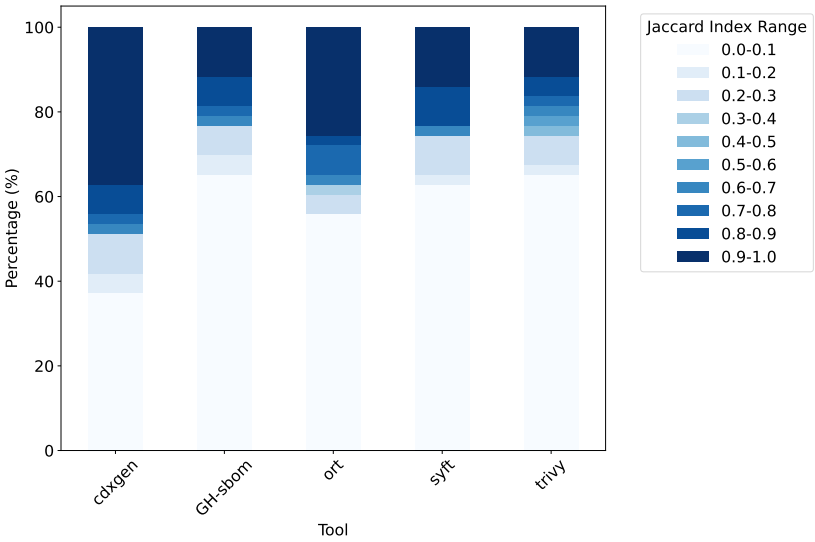

Figure 2: Jaccard Similarity Distributions. Each bar represents the percentage of SBOMs that lead to identification with a certain Jaccard index range.

considered preferable to missing vulnerabilities [36, 19, 49], the magnitude observed here makes the results impractical for security teams.

By sampling projects affected by false positives, we identified two main causes. First, SBOMs often include dependencies listed in metadata files but not actually collected during installation. On average, 75% of the dependencies in the generated SBOMs fall into this category, creating a systematic misalignment between declared and installed dependencies (see Section 2.1.2). In fact, projects may contain additional requirements.txt files listing dependencies that are actually involved in the packaging of the project. The high number of false positives is caused by vulnerabilities affecting dependencies contained in these additional files. Since SBOM generation tools considers them during SBOM generation, the vulnerability report ends up to contain many vulnerabilities that are not actually involved in the package. It is noteworthy that this behavior occurs also when the SBOM is generated through environment analy-

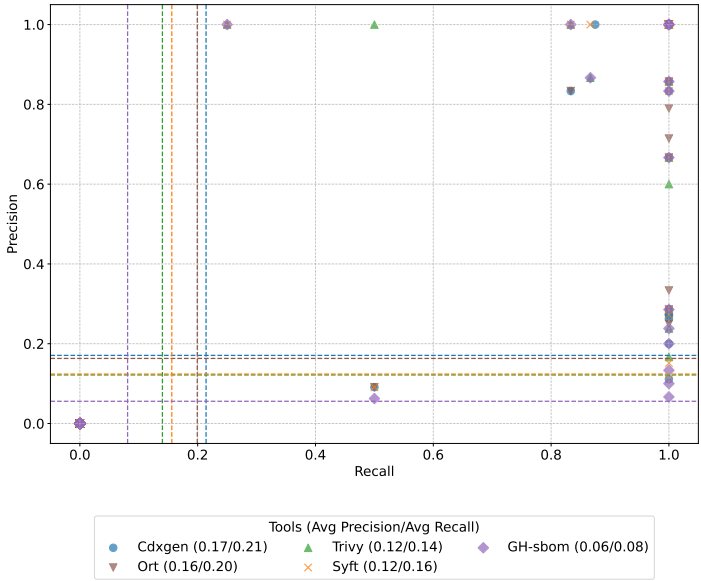

Figure 3: Precision and Recall for the vulnerability scans conducted through SBOMs generated by each of the selected SBOM generation tools.

sis, since the dependencies reported in *all* the requirements.txt files are installed in the environment. Second, in a small number of cases, mismatches between vulnerability identifiers in databases and those in the ground truth caused additional misclassifications. This issue is linked to limitations of vulnerability databases themselves, which may be outdated or incomplete, as shown for example by recent delays in CVE collection by NVD [37].

2.3.5 Discussion

These findings underline the critical importance of SBOM quality for effective vulnerability detection. None of the tools evaluated can correctly identify vulnerabilities in more than a small fraction of cases. Among them, cdxgen achieves the highest accuracy, owing to its installation-

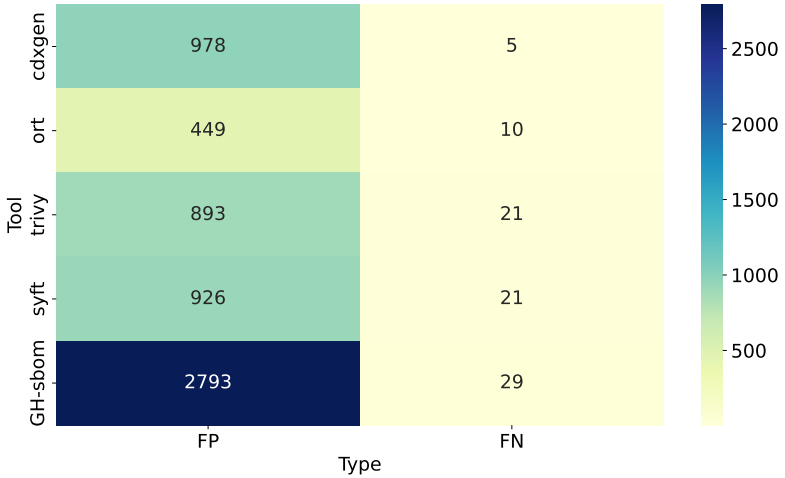

Figure 4: False Positives and False Negatives by Tool

based approach. Yet even in this best-case scenario, the overwhelming number of false positives, primarily caused by dependencies that are declared but never actually installed, makes the results unsuitable for real-world use.

These limitations reflect the findings of the previous contribution of the Thesis, Section 2.2. This underscores the immaturity of current SBOM generation tools and highlights the need to address the foundational weaknesses of SBOM generation to improve Software Supply Chain security.

2.4 Improving SBOM Generation with a Novel Approach

We proved that SBOM generation tools are not yet mature and they cannot properly support the vulnerability assessment, causing incomplete and incorrect vulnerability reports. We propose a new methodology to generate the SBOM and demonstrate that it outperforms existing SBOM

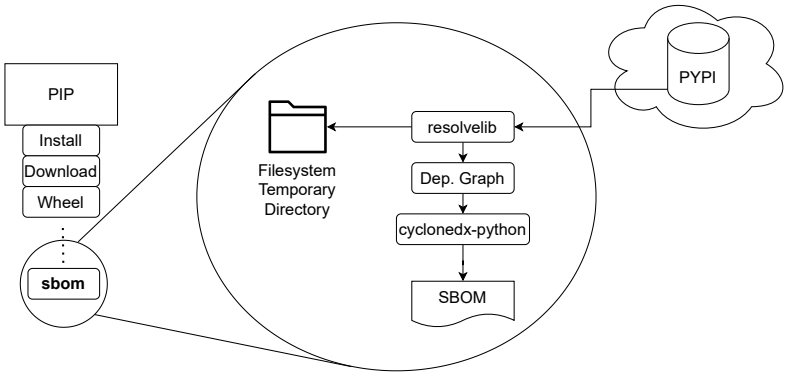

Figure 5: Design of PIP-SBOM. The implementation of PIP is extended to include SBOM generation in the build phase.

generation tools, producing SBOMs that function better as input to vulnerability scanner.

This Section describes the implementation of PIP-SBOM, the SBOM generation tool we propose, and its evaluation. We test it against the same set of applications used for the previous study, and we compute the same measurements. We finally present the results.

2.4.1 PIP-SBOM Implementation

This section presents PIP-SBOM, our pip-based approach for native generation. Its design is summarized in Figure 5.

PIP-SBOM is implemented as an extension of PIP, the official package manager of the PyPI ecosystem. We chose pip because it supports multiple front-ends and back-ends—allowing it to build projects managed by other tools such as Poetry—and because it is widely adopted across both expert and novice developers.

Internally, PIP is organized into modules corresponding to its CLI commands. We introduced a new module, SBOM, that contains the logic

required to generate an SBOM.[10]

This design enables developers to generate an SBOM for a Python project with the command: `pip sbom <project-path>`.

The approach is structured into two processes. The online process queries the PyPI registry to obtain the dependency network, while the offline process builds the dependency graph and generates the SBOM document for the target project.

Dependency Network Solving

Dependency resolution builds on PIP 's use of the `resolve-lib` package, which handles version constraints by optimizing the navigation of the dependency tree.[11] Our implementation reuses this logic to replicate the same algorithm employed during dependency retrieval at installation time.

This process behaves similarly to the `download` command: the project's dependencies are retrieved from PyPI and stored in a temporary directory, which is deleted upon completion. By automatically resolving version constraints, this procedure ensures that the generated network accurately reflects the dependencies selected during installation. Constraints that cannot be resolved due to incompatibility are discarded, consistent with pip's behavior during actual project builds.

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 modeled as a directed, unweighted graph $G = (V, E)$. Each node $n \in V$ represents either a direct or a transitive dependency of the input project $r \in V$, while each edge $(u, v) \in E$ denotes a dependency relationship. The relation is transitive: if $u \to v$ and $v \to w$, then $u \to w$. Thus, u is a transitive dependency of w.

[10] PIP-SBOM refers to the extended version of pip as a whole, while SBOM denotes the additional module. For simplicity, we use PIP-SBOM throughout to refer to both.

[11] `https://pypi.org/project/resolvelib/`

PIP-SBOM can output the dependency graph as a `.dot` file using the option `-g <file-name>`. Internally, the graph representation serves as the foundation for generating the final SBOM.

SBOM Generation

Once the graph is constructed, PIP-SBOM traverses it and creates an entry in the `components` field of the SBOM for each node. Each entry records the bom-ref, dependency name, version, and purl—information essential for vulnerability scanning tools (see Section 2.1.1). Although the implementation focuses on these fields, since they are needed to vulnerability scanner, the resulting document can be enriched to comply with the minimum data elements required by NTIA [30].

The graph's edges are used to populate the `dependencies` field of the SBOM, capturing the relationships between components. When traversal completes, PIP-SBOM outputs a CycloneDX-compliant SBOM.

2.4.2 PIP-SBOM Evaluation

To assess the effectiveness of PIP-SBOM, we perform the same evaluation used for comparing state-of-the-art SBOM generation tools, measuring the vulnerabilities detected from its generated SBOMs against those obtained from the ground truth. This ensures that PIP-SBOM is evaluated under identical conditions, allowing a direct comparison with existing tools. As in the previous analysis, we rely on the Jaccard similarity index, precision, recall, and the distribution of false positives and false negatives to characterize performance. These metrics provide both a high-level perspective on the accuracy of vulnerability detection and a detailed view of the factors influencing the results, enabling us to position PIP-SBOM within the landscape of SBOM generation approaches.

2.4.3 Results

Extending PIP with a dependency resolution algorithm native to the package manager enables a significant improvement in the accuracy of vulnerability assessments. As shown in Table 9, PIP-SBOM reaches a Jaccard

Table 9: 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**

similarity of 78.39%, with almost 80% of vulnerability reports matching the ground truth. No other tool considered in this study achieves comparable performance.

Compared to the best-performing baseline, PIP-SBOM improves precision by 64 percentage points and recall by 59 percentage points, reaching 80.95% and 80.26% respectively. The small gap between precision and recall indicates that PIP-SBOM provides a balanced and consistent detection capability, reducing the manual effort required to discard false positives. This is confirmed by the absolute numbers: only 47 false positives were recorded, a drastic reduction compared to other tools that exceed 900 in most cases. False negatives remain comparable to those of alternative approaches.

The significant reduction in false positives makes vulnerability reports more actionable for developers, lowering the operational burden and supporting the practical adoption of SBOMs for security purposes. Discrepancies between the vulnerability assessments made on the SBOM generated with PIP-SBOM and the ground truth were manually reviewed and found to be caused exclusively by mismatches in vulnerability identifiers, rather than by limitations of the dependency resolution process. In fact, as already noted for other SBOM generation tools false positives and false negatives may be caused by the vulnerability scanner that match a component to a vulnerability identifier (e.g, a CVE number) that is present in the ground truth with a different identifier (e.g., a GHSA number).

2.5 Related Work

This section positions our contribution about SBOM to other research. In particular, related to SBOM generation and consumption.

2.5.1 SBOM Generation

Despite the emergence of the SBOM technology and the well-known problems of its generation [60, 69, 66], 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. [28] 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. [68] 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. [3] focus on Java, Rabbi et al. [47] 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.

2.5.2 SBOM Consumption

Although the SBOM is widely recognized as a valuable instrument for enhancing Software Supply Chain transparency and supporting both functional and security testing, its adoption in practice remains limited. Recent work by Sharma et al. [53] demonstrates the potential of SBOMs for security, proposing a technique that leverages SBOM information to mitigate vulnerabilities in Java applications.

Enck et al. [17] highlight that practitioners debate the actual benefits of SBOMs for improving security. More recent evidence suggests that skepticism persists: Zahan et al. [69], for example, report that participants at the S3C2 Industry Summit [48] expressed doubts regarding the feasibility and usefulness of SBOM adoption. A recurring concern is the difficulty of integrating SBOM generation into continuous integration and deployment (CI/CD) pipelines. One proposed remedy is to embed SBOM generation directly into standardized build templates, making it a mandatory step in CI/CD processes. In this thesis, we show that such an approach can be adopted without disrupting the build process, at least in the case of the Python ecosystem.

2.6 Contribution Summary

This Thesis makes three contributions toward improving SBOM generation and its role in software supply chain security.

First, we conduct a systematic assessment of state-of-the-art SBOM generation tools, analyzing their completeness and correctness in representing Python projects' dependency networks. The results reveal significant inconsistencies across tools, with none achieving full accuracy in capturing the actual dependencies.

Second, we evaluate the impact of these shortcomings on vulnerability assessment workflows. Our findings show that inaccurate SBOMs

severely degrade the performance of security tools, leading to low similarity with ground truth and an excessive number of false positives, which burdens developers and reduces trust in SBOM-based security practices.

Finally, we propose and implement PIP-SBOM, a novel approach that integrates SBOM generation into the dependency resolution process of PIP. This design provides a precise view of the installed dependency graph, ensuring high fidelity with the ground truth. Our evaluation demonstrates that PIP-SBOM outperforms the other tools considered in this study by a wide margin, achieving near 80% precision and recall and drastically reducing false positives. This improvement highlights the feasibility and benefits of embedding SBOM generation in package managers as a step toward more reliable and actionable Software Supply Chain transparency.

Chapter 3

Runtime Identification of Dependencies

The SBOM is a powerful tool to enhance transparency in the Software Supply Chain. As we discussed in Chapter 2, it presents some limitations that prevent its widespread adoption and its full reliability. Researchers contribute to improving its generation, and we also prove that a better solution in Python is possible and easy to implement. However, the SBOM generated at build-time is not well-suited for identifying runtime dependencies because of its static nature.[1] Missing runtime dependencies information has two main limitations. First, static dependency analysis does not reflect the actual dependency usage at runtime—that is, whether a dependency is truly loaded or invoked in production [3]. Although this may not pose a direct risk to the application, it leads maintainers and developers to spend significant time assessing the security of unreachable dependencies, thereby slowing down their response to real security threats. Second, static analysis provides no support for maintainers or developers in making runtime decisions based on the dependencies that are actually used. In essence, the fundamental limitation of build-time SBOMs is that all dependency information is lost once the

[1] Throughout this chapter, even not specified otherwise, the term SBOM refers to the build-time SBOM.

application is running.

Runtime dependency introspection addresses these limitations. It enables software to observe which components are executed during actual runtime, including their names and versions. Such introspection forms the foundation for advanced security hardening. First, it allows the identification of dependencies that are bundled with the application but never used at runtime. This facilitates more effective vulnerability management by deprioritizing unused components and helps reduce the attack surface by eliminating them. Second, runtime dependency introspection is essential for enforcing runtime privilege restrictions, as it allows the mapping of accessed resources to their corresponding dependencies [1].

Also, in this case, depending on the ecosystem, there are different ways to manage runtime dependencies. Our contribution focuses on Java, since it represents one of the most widely used programming languages for critical software [22]. The Java runtime is known for its good support of object-oriented introspection (aka class, field, and method reflection). Yet, it is completely blind to dependencies: it has no native support for dependency introspection at runtime [67]. In Java, the metadata that uniquely identifies each dependency is available at build time but is not retained at runtime. Previous studies have attempted to infer dependency origins using heuristics—such as mapping package names to their potential sources [1]. However, these approaches face inherent limitations, as they rely on fragile naming conventions that must be consistently followed to work correctly. Consequently, achieving reliable runtime introspection of dependencies remains an open challenge. To the best of our knowledge, there are currently no principled blueprints for identifying which dependencies are actually executed at runtime in Java.

In this Thesis, we propose Classport, a novel technique for introspecting dependencies at runtime in Java. In essence, Classport brings dependency information to runtime: supplier, dependency name and version (group id, artifact id and version in Maven lingo). Classport operates in two phases by: *(i)* embedding dependency information into Java bi-

nary artifacts through build-time instrumentation, and *(ii)* retrieving this information at runtime through dynamic instrumentation.

We evaluate Classport on six real-world open-source Java projects. Specifically, we determine its ability to embed dependency information into the binary artifacts, assessing the time and disk space overhead caused by the added information. Then, we test Classport's ability to identify the correct set of runtime dependencies according to a meaningful workload. We also consider the impact on the running application, incl. the runtime overhead.

Our results demonstrate the applicability of Classport to real-world applications. Classport successfully embeds dependency information, accurately identifies runtime dependencies, preserves the functional behavior of the application, and introduces a low overhead.

In summary, the main contributions are:

- Classport, a novel approach to inspect Java dependencies at runtime, based on Java annotations. To the best of our knowledge, it is the first ever solution for runtime dependency introspection in Java.
- A publicly available prototype to embed information into binary artifacts and a runtime agent to retrieve dependency information during execution.[2]
- An evaluation of our novel technique on a set of six real-world applications, demonstrating the feasibility and applicability of our approach.[3]

3.1 Background

This section provides background on how dependency management works in the Java ecosystem, with a focus on Maven and its limitations regarding runtime visibility of dependencies.

[2] `https://github.com/chains-project/classport`
[3] `https://github.com/chains-project/classport-experiments`

Dependency management by package managers and build systems varies across ecosystems, as they handle resolution, versioning, and transitive dependencies differently. In the Java ecosystem, Maven is one of the available build systems [20]. Within Maven, developers declare dependencies in a file called pom.xml. Each of these dependencies is downloaded as a binary artifact called a JAR [62], and is uniquely identified by Group ID, Artifact ID, and Version [63], this triple being called a GAV coordinate. The Group ID represents the organization or project namespace, e.g., *org.apache.commons*. The Artifact ID specifies the module or library, e.g., *commons-lang3*. The Version distinguishes the same library across different releases, e.g., *3.12.0*.

At build time, the dependencies of pom.xml are resolved, in order to list the GAV coordinates of both direct and transitive dependencies. This allows Maven to download the exact version to use. In cases of conflicts between multiple versions of the same transitive dependency, Maven's dependency resolution mechanism selects the version closest to the root of the dependency tree [62]. The version visible in pom.xml is thus not necessarily the same one that will be used at runtime. In Maven, dependency metadata such as the GAV of dependencies is only available at build time; this information is not preserved at runtime at all [67]. The platform makes it impossible to identify the set of dependencies that are actually used at runtime, or to make any runtime decision depending on the dependencies. In other words, Java supports classes, methods, and fields introspection, but not dependency introspection.

In this thesis, we address this problem by proposing a blueprint architecture and a prototype for supporting dependency introspection in Java.

3.2 Classport

We propose Classport, a novel technique for runtime dependency introspection in Java. Recall that in Java, dependency metadata is available at build time, but it is absent during execution.

We define runtime dependencies as dependencies that: 1) are fully

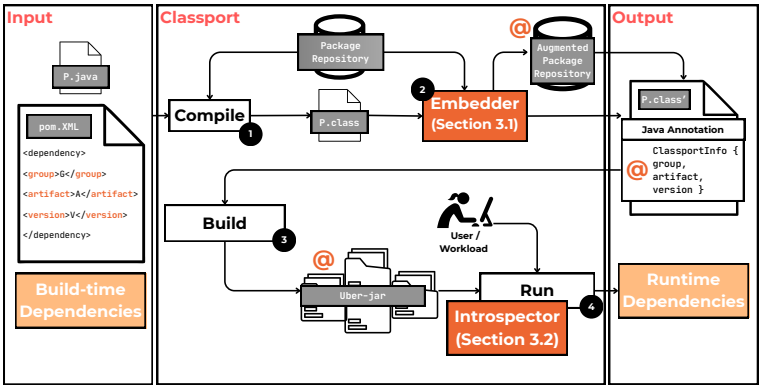

Figure 6: Overview of Classport, a novel system that enables runtime dependency introspection in Java.

specified within the binary to be executed, and 2) are available as variables from within the program.

We also provide an open-source implementation of Classport. The main goal of Classport is to embed Software Supply Chain information, i.e., the dependencies, in the final executable artifact of a Java application, and extract this information at runtime, during application execution. Classport achieves this goal through its two key components: the Embedder and the Introspector. The Embedder stores the information about dependencies in Java artifacts, while the Introspector extracts this embedded information with runtime introspection.

Figure 6 presents an overview of Classport and its integration within the Maven lifecycle. Classport processes a target Java Maven project, which contains the source code and the information about build-time dependencies, which are listed in the pom.xml file. The Embedder performs the *embed* action (step 2), following the Maven compilation phase (step 1). This produces the Java binary artifacts enriched with the dependency information, which are represented in the output box. After the Maven build phase (step 3), the uber-JAR, i.e., the executable archive comprising the project's dependencies, is run (step 4). During this phase,

the Introspector *introspects* the execution of the Java program and gives as output the runtime dependencies.

The following two subsections detail the technical aspects of the two components of Classport. Next, we present Classport with an end-to-end example in Section 3.2.3. Finally, we give details about the implementation of Classport, including tools and libraries used, in Section 3.2.4.

3.2.1 Embedder

The **Embedder** is the first key component of Classport, designed to bridge the gap between build-time and runtime visibility of dependency information. In Java Maven projects, dependency metadata is explicitly declared in pom.xml and fully resolved during the build phase, but this information is not preserved in the final application artifacts. The Embedder addresses this limitation by capturing build-time dependency metadata and embedding it directly into the compiled artifacts, making it accessible at runtime.

The Embedder takes as input the compiled artifacts of a target Maven project, i.e., the project's class files and the JARs of the project's dependencies. The compilation phase, step 1 in Figure 6, ensures that all dependencies are resolved and available locally. For each dependency JAR, the Embedder locates the local version that was downloaded and processes each file within the JAR. Depending on the resource it encounters, this processing involves different actions. For class files within a dependency JAR, i.e., compiled Java source files with `.class` extension, the Embedder embeds dependency metadata, i.e., its group, artifact, and version, using a bytecode transformation. The same process is applied to the application classes, such as *P.class* in Figure 6, annotated with the project module information. In Figure 6, the embedded information is represented by the @ symbol.

When the Embedder encounters manifest files with a dependency JAR, such as *MANIFEST.MF* files, it removes signature-related entries, e.g., digest attributes, and copies the remaining content. This is necessary because modifying the contents of a signed JAR, such as adding

metadata to class files, invalidates its signature. The Embedder ignores signature files and copies all other files without modification. We refrain from altering non-class files. If these other files are used, they must be referenced by other class files — which are themselves embedded. If they are not used, the Java Virtual Machine will not load them, and hence we don't need to get provenance later. After processing each dependency JAR, it repackages it into a new JAR. It saves it in a directory, which we refer to as the augmented package repository, as shown in Figure 6.

Finally, as shown in step 3 of Figure 6, the augmented package repository, i.e., the one containing the annotated dependencies, is used to build the final uber-JAR with embedded software supply chain information.

We highlight that the removal of signatures can be mitigated by signing the Java class file content, including the annotations, at build time and checking the signature at class loading time, for instance, using the built-in tool `jarsigner`. This would ensure that the content of each class file is verified at runtime. Also, we acknowledge that relying on uber-JARs can be a potential limitation in specific build models that, e.g., require the collection of JARs at runtime, and that other limitations, such as the tracking of dependencies specified as `provided` in the pom.xml may happen in real-world scenarios. Both scenarios would require further investigation and are out of the scope of this particular contribution.

The rest of this subsection describes how the Embedder is implemented.

Embedder as a Maven plugin

We use a Maven plugin to implement the Embedder.[4] A Maven plugin is a reusable and highly configurable component that adds specific tasks or goals to Maven's build process. In particular, it is possible to decide in which build phase to execute it by configuring a Maven plugin goal.

Within Classport, we create the custom *embed* goal. We configure it to fulfill the following needs of the plugin: *(i)* to have project's class files available locally. This is fundamental because they have to be embedded.

[4]https://maven.apache.org/guides/introduction/introduction-to-plugins.html

Listing 3.1: The Classport Java annotation for sorting dependency information within the code. The annotation is available at runtime through introspection.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassportInfo {
    String group(); // G
    String artefact(); // A
    String version(); // V
}
```

(ii) To have dependencies' class files. Dependencies must be resolved to also have both direct and transitive ones.

The first point is achieved by compiling the target project before executing the plugin. It guarantees that the class files for the project are available for processing. The second point is fulfilled by ensuring that the resolution of dependencies happens before the embedding process.[5]

Information embedded as Java Annotations

The primary objective of the tool is to introspect dependency information during execution, so it is crucial to embed data in a manner that allows for retrieval at runtime. We use Java Annotations to add such information into the binary class files. We use bytecode transformation for that task. In each class, we create a custom annotation *ClassportInfo*, from the annotation type shown in Listing 3.1.

The annotation contains the GAV coordinate, i.e., group, artifact, and version of the dependency to which the class belongs. The Retention annotation [33], `@Retention(RetentionPolicy.RUNTIME)`, is used to make the `ClassportInfo` annotation visible at runtime. After the injection of Software Supply Chain information by the Embedder, the target application is packaged as a uber-JAR.

[5] `https://maven.apache.org/plugin-tools/apidocs/org/apache/maven/plugins/annotations/ResolutionScope.html`

3.2.2 Introspector

Once dependency metadata has been embedded, it has to be retrieved during execution. The **Introspector** component of Classport enables this by dynamically extracting the embedded dependency metadata at runtime. As illustrated in phase 4 of Figure 6, the Introspector introspects the running application, reading embedded annotations from running classes to recover dependency metadata. This process allows Classport to determine which dependencies are actually used during execution, providing a precise view of the runtime dependency set.

Introspector as Java agent using the Instrumentation API

The Introspector is a Java agent attached to the JVM. The goal of the Introspector is to add code that extracts the dependency annotations. To achieve this, we instrument the methods of the application under study using code transformation. Each time an instrumented method is executed, the extraction logic is triggered, storing the current dependency GAV in a set. To maintain a low overhead, not every method is instrumented. Specifically, since our objective is to have a list of used dependencies, once a dependency GAV is identified and stored in the set, the subsequent methods belonging to that dependency are no longer instrumented. This allows us to drastically reduce the overhead, without modifying the output of Classport. Finally, after execution of the workload, Introspector writes the resulting set of dependency GAVs into a CSV file.

3.2.3 End-to-end Example

Listing 3.2 shows the changes Classport makes on the original bytecode of the class `ExtractText` in the `PDFBox` project.[6] In particular, it shows the resulting bytecode modifications after running the Embedder. Classport adds an attribute in the attribute section (lines 12 to 17), which is the representation of the annotation. The keyword `RuntimeVisibleAnnotations` (line 12) indicates the retention policy that has been set, and that the an-

[6] https://github.com/apache/pdfbox

Listing 3.2: Example of resulting Java bytecode after embedding process.

```
1  Classfile /.../ExtractText.class
2        ...
3     Constant pool:
4        ...
5       #89 = Utf8               SourceFile
6       #90 = Utf8               ExtractText.java
7  +    #91 = Utf8               Lio/github/.../ClassportInfo;
8  +    #92 = Utf8               group
9  +    #93 = Utf8               org.apache.pdfbox
10       ...
11    Attributes:
12 +    RuntimeVisibleAnnotations:
13 +      @ClassportInfo(
14 +         group = "org.apache.pdfbox",
15 +         version = "3.0.4",
16 +         artefact = "pdfbox-tools",
17 +      )
```

notation is visible at runtime. The remaining part of the section represents the key-value pairs of the annotation, i.e., the Group, Version, and Artifact. Finally, the constant pool section (lines 4-9) is consequently updated with the added values.

Listing 3.3 shows the CSV output content of the Introspector. It reports the list of dependencies used during the execution of the application. Specifically, the Group, Artifact, and Version are reported for each dependency.

3.2.4 Implementation

Classport is implemented in Java. It is made of two modules: the Embedder uses the ASM[7] library to manipulate the bytecode in the class files, while the Introspector is a Java agent that uses the Instrumentation API [32]. The implementation is open-source and available on GitHub.[8]

[7] https://asm.ow2.io/

[8] `https://github.com/chains-project/classport`

Listing 3.3: Example of Introspector's output for PDFBox workload execution. Each row is a triple of Group, Artifact, and Version.

```
org.apache.pdfbox,pdfbox,3.0.4
org.apache.pdfbox,fontbox,3.0.4
org.apache.pdfbox,pdfbox-io,3.0.4
commons-logging,commons-logging,1.3.4
info.picocli,picocli,4.7.6
org.apache.pdfbox,pdfbox-tools,3.0.4
org.apache.pdfbox,pdfbox-debugger,3.0.4
```

3.3 Experimental Methodology

This section outlines the methodology for our study evaluating the implementation of Classport. First, it presents the research questions that guide our investigation and information about the dataset. Then, it provides a detailed explanation of how we address these research questions. Lastly, it shows the overall experimental setup.

3.3.1 Research Questions

We evaluate Classport to demonstrate its effectiveness in introspecting runtime dependencies in real-world applications. First, we assess its capability to embed dependency information into Java binary artifacts. Then, we test its behaviour during runtime execution and its ability to retrieve the actual set of running dependencies. Specifically, we investigate the following research questions:

RQ1: To what extent can Classport effectively embed dependencies into Java binary artifacts?

RQ2: To what extent does Classport support runtime inspection of dependencies?

3.3.2 Dataset

We aim at evaluating Classport with a dataset of real-world, maintained, open-source Java projects, which use Maven and span a range of ap-

Table 10: Study subjects considered in our experiments.

Project	Version	Deps.	Workload
PDFBox-app	3.0.4	12	Extract text from a PDF file
Certificate-ripper	2.4.1	5	Print the certificate of a website
mcs	0.7.3	4	Lookup dependency coordinates in Maven Central
batik	1.17	6	Convert an SVG to PNG
checkstyle	10.23.0	34	Lint a Java file
zxing	3.5.3	4	Decode 4 QR codes

plication domains, dependency sizes, and usage scenarios. Our dataset includes six projects, with projects varying in complexity, and includes both lightweight tools and large frameworks such as checkstyle. The most active repository is zxing, which has over 30K stars and around 10K forks on GitHub. Table 10 lists each project along with its version, number of dependencies, and the workload we use to exercise the project. The number of dependencies is calculated directly by Maven. The workload is decided by considering a regular use case scenario for each application. Note that we do not assume a minimum coverage. We do not aim to detect all the reachable dependencies, the goal is rather to demonstrate feasibility, that Classport is able to bring dependencies information, i.e., the GAV, at runtime, for a typical use case scenario. We compile all projects for RQ1 and analyze the resulting artifact. For RQ2, we execute all projects to test the Introspector component.

3.3.3 Methodology of RQ1

The first research question aims to evaluate the Embedder. We assess the dependency completeness, class completeness, and the performance overhead of embedding dependency metadata into Java binary artifacts using Classport. This is done by statically analyzing the JAR file produced by Classport.

Dependency Completeness This property ensures that Classport embeds the uber-JAR with dependency metadata of all dependencies. To assess this, we first run the Embedder on the dataset, generating the annotated uber-JARs. We only consider dependencies that are needed for

the application's execution, excluding test-only dependencies, as they do not end up in the uber-JAR. Then, we analyze the uber-JARs to read the annotations inside the binary artifacts. We consider the embedding process complete in terms of dependency metadata if the dependency metadata, i.e., each Group ID, Artifact ID, and Version, matches the ground truth retrieved by running the Maven command `mvn dependency:list` on the project under test.

Class Completeness We assess completeness by verifying that the annotation is embedded in each class file of every dependency in the project. Although the dependency can have files other than class files, such as build manifests and various configuration files [52], we only consider class files for annotation as they are the only ones that can be executed. To verify completeness, we first run the Embedder on the study subjects. Then, we analyse the uber-JAR, looking inside each class file. If all Java class files are annotated with the Classport annotation, the embedding process is complete in terms of class metadata.

Build Time Overhead Build time overhead is measured by comparing the time taken to build the uber-JAR with and without the Embedder. In detail, we run the package phase of the Maven build process with and without the Embedder and measure the time taken for each run using the Unix `time` command. We compute the time difference between the two runs and report the overhead as absolute time in seconds.

Disk Overhead Disk overhead is the additional space required to store the embedded dependency metadata. To measure the disk overhead, we compare the sizes of the uber-JAR before and after embedding. Specifically, we run the same Maven command as in 'Build Time Overhead' to build the uber-JAR with and without the Embedder and measure the disk usage for each run using the Unix `stat` command. We compute the difference in size of the JAR between the two runs and report the percentage of overhead.

3.3.4 Methodology of RQ2

The second research question evaluates the behavior of Classport at runtime, in particular, its capability to introspect runtime dependencies. The objective is to show that 1) Classport can correctly identify which dependencies are currently running and 2) the target application's runtime behavior is not affected by Classport.

To evaluate the Introspector, it is necessary to first run the Embedder, to ensure that the embedding process is completed and also that all the dependency metadata is embedded into the uber-JAR. This means that we assume a correct Embedder, as validated by RQ1, and we now assess the Introspector, which is the component dedicated to the runtime introspection.

Dynamic Correctness Dynamic correctness is satisfied when the runtime-detected dependencies correspond to the ones actually used during the execution of the workload, i.e., the number of false negatives and false positives is equal to zero. First, we define a workload for each application, as shown in Table 10. The workload reflects a regular use case of the selected application. Then, we run the workload on the embedded project, which produces a list of runtime dependencies observed during the execution of the workload. We compare this list with the ground truth, i.e., the list of dependencies resulting from the `mvn dependency:list` command. If all the dependencies match, this means that Classport can correctly identify dependencies for this workload at runtime. In the case of missing dependencies from the list detected by Classport, this means that either the workload does not trigger the dependencies or that it is a false negative. To assess this, we modify the pom.xml of the project to include only dependencies that are detected by Classport. Specifically, we moved the dependencies not detected by Classport out of the runtime classpath by changing the scopes of dependencies. Finally, we repackage the application and rerun it with the same workload. If the application executes without errors and produces the same output as without Classport, this demonstrates the correctness of Classport in identifying all runtime dependencies that are necessary for

the workload.

Overhead To measure the runtime impact of Classport, we use Java Microbenchmarking Harness (JMH) to benchmark the workloads with and without the Classport agent.[9] We prefer microbenchmarking over a simple time difference between executions because microbenchmarking helps record the time of execution when the JIT compiler has fully optimized a Java application. Microbenchmarking ignores optimization time, which can lead to misleading performance results [13]. By accounting for Java Virtual Machine (JVM) optimizations such as JIT compilation and warm-up, JMH provides a more accurate assessment of the runtime overhead introduced by Classport.

Functional correctness For validating that Classport does not break correctness, we run the test suite of the target application and monitor for runtime errors or test failures caused by the agent.

3.3.5 General Setup

We run the experiments on a server with an Intel i9-10980XE CPU (36 cores), 125 GB RAM, using Java 17.0.15 and Maven 3.8.7. The dataset and experiment scripts are publicly available on GitHub.[10]

3.4 Experimental Results

This section presents the results of the experiments as detailed in section 3.3.

3.4.1 Results for RQ1

The results of RQ1 are summarized in Table 11. It presents the effectiveness of Classport in embedding dependency metadata into Java class files across the open-source projects tested. Each row of the table represents the results of a project on our dataset. The first column reports the completeness of the embedding process considering two aspects. The

[9] https://github.com/openjdk/jmh

[10] `https://github.com/chains-project/classport-experiments`

Table 11: RQ1 – Static Completeness and Overhead of Embedding Process

Project	Completeness		Overhead	
	Dependencies	Classes	Time	Space
PDFBox-app	12/12	7,914/7,914	+18.86s	12%
Certificate-ripper	4/5	426/426	+2.68s	14%
mcs	4/4	557/557	+4.43s	10%
batik	6/6	3,679/3,679	+1.15s	29%
checkstyle	34/34	10,691/10,691	+6.18s	17%
zxing	4/4	717/717	+0.51s	11%

first one is the number of dependencies embedded compared to the total number of dependencies in the project, and the second one is the number of class files containing the Classport annotation, compared to the total number of class files in the uber-JAR. Then, the table reports the overhead introduced during the build phase by the Embedder. In detail, it reports the absolute time overhead, i.e., the time added by the embedding process to the build, and the space overhead, i.e., the percentage increase in the size of the uber-JAR after embedding.

Consider the first row of the table, which corresponds to the PDFBox project. Classport successfully embeds all 12 dependencies of PDFBox-app, which is the total number of dependencies required by the application. The output uber-JAR contains 7,914 class files, and all of them are annotated with the dependency metadata. The embedding process introduces an absolute time overhead of 18.86 seconds and a space overhead of 12% in the final uber-JAR.

Classport successfully embeds all dependencies and class files in the projects we tested. Thus, 5 out of 6 projects are completely embedded, while Certificate-ripper has one dependency that is not embedded. This is because the dependency is only required for compilation and not for runtime, which is a characteristic known at build time by Maven, and hence it is not included in the uber-JAR. Moreover, classes of uber-JARs of all 6 projects, including the partially embedded project, are annotated with the dependency metadata. This confirms that the embedding process is complete in terms of adding dependency metadata.

The embedding process introduces a moderate performance cost during the build phase. The build time overhead ranges from 0.51 to 18.86 seconds of added time. The value is computed as the time required for the embedding plugin to be run. Considering the entire build process, this time can be negligible in the context of longer testing or packaging steps [12]. The space overhead also varies from 10% to 29% which is considered acceptable given the introduction of a novel feature - visibility into runtime dependencies. The space overhead depends upon the number of dependencies and identifier names in the core binary content. Certificate-ripper and mcs have about the same number of classes and the exact same number of dependencies embedded, yet the overhead is larger for Certificate-ripper due to longer identifier names. Moreover, the dimension of the class files has a role in the space overhead: since it is computed as a percentage, applications with small class files present a higher percentage overhead than those with big class files, which is the case of batik.

Answer to RQ1: Our experiments show that Classport completely embeds dependency metadata into Java class files. The evidence is supported by 1) six diverse Maven projects 2) which reflect real-world usage and complexity. Build time and space overhead are an acceptable trade-off for getting a unique feature absent from the Java stack: visibility into runtime dependencies.

3.4.2 Results for RQ2

We now assess to what extent we are able to obtain the dependency information during execution. Table 12 summarizes the effectiveness of Classport in detecting dependencies at runtime across the evaluated applications. The table is divided into two parts: the first reports the results regarding the dynamic correctness, while the second shows the runtime impact. Regarding the correctness, the introspection process is defined as correct if the set of runtime dependencies corresponds to those actually used during the execution of the workload, as discussed in Section 3.3.4. Specifically, the first column reports the number of true positives (TP),

Table 12: RQ2 – Dynamic Correctness and Overhead of Introspector

Project	Dynamic correctness				Runtime Impact	
	TP	FP	TN	FN	Overhead	Functional Correctness
PDFBox-app	7	0	5	0	0.74%	✓
Certificate-ripper	4	0	0	0	4.27%	✓
mcs	3	0	1	0	1.22%	✓
batik	2	0	4	0	1.24 %	✓
checkstyle	5	0	29	0	0.81%	✓
zxing	3	0	0	0	4.02%	✓

which is the number of dependencies detected by Classport, that are also present in the ground truth, i.e., the dependencies resulting from the `mvn dependency:list` command. The second column reports the number of false positives (FP), which represents the dependencies that Classport finds but are not present in the ground truth. The third one is about true negatives (TN), i.e., the dependencies that are present in the ground truth but not detected by our tool because the workload does not trigger them. Finally, the fourth column reports the number of false negatives (FN), i.e., the number of dependencies missed by Classport. The part of the table related to the runtime impact is divided into two columns: one reports the percentage of overhead, and the other indicates whether the functional correctness of the target project is preserved after running Classport. For instance, in the case of mcs, Classport successfully detects three dependencies, which are true positives. The one that it does not detect is a true negative because the workload does not trigger it. Classport introduces an overhead of 1.22% and preserves the functional correctness.

Detected Runtime Dependencies The correctness of the identified runtime dependencies is confirmed for all the tested applications. For certificate-ripper and zxing, the workload covers the entire set of dependencies. For the other applications, not all static dependencies are detected at runtime. To investigate whether this is a false negative, we remove from the runtime classpath the dependencies that are not detected by the

workload, as explained in subsection 3.3.4. For PDFBox and checkstyle, the workloads succeed, confirming that Classport correctly identifies the dependencies covered by the workload.

For mcs and batik, the workload fails because one class of removed dependencies is loaded. Manual investigation of the code reveals that these classes are loaded but not executed. This is due to the internal logic of the JVM that requires loading classes even if they are not used.[11] This behavior is due to different reasons; in our case, this is related to the following two cases. The JVM loads classes when they are referenced in the code, such as when an object is assigned to a specific type, requiring the JVM to verify that type for safety, even if no actual methods are invoked. Similarly, classes for declared exceptions are loaded, regardless of whether those exceptions ever occur during execution. It would be possible to modify Classport to also intercept the class loading event, but we aim at only detecting executed code. Overall, Classport correctly detects the dependencies used at runtime.

Overhead. The overhead during the execution of the workload is negligible or low for all the study subjects. It varies from 0.74% to 4.27% which shows that reporting the runtime dependencies has a limited impact on the performance of the application. The overhead is due to the fact that Classport needs to instrument methods invocation and retrieve the dependency metadata from the class files.

Functional correctness We check that Classport does not break any functionality by running the test suite. For all the applications in the dataset, no functional divergence is observed. This means that the bytecode instrumentation done by Classport has no negative effects on the target application.

[11] https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-5.htm

Answer to RQ2: Classport correctly introspects runtime dependencies. It does not affect application behavior. The execution overhead is low for all the study subjects. To our knowledge, Classport is the first approach to embed and retrieve dependencies at runtime in the Java ecosystem.

3.5 Use cases of Classport

This section describes the possible benefit of having dependency information available at runtime in Java.

3.5.1 Runtime Permissions per Dependency

Limiting what resources (e.g., network, filesystem) software components can access at runtime is relevant for security. In the context of Software Supply Chain security, a promising strategy is to assign permissions per dependency, ensuring that each third-party library is granted only the privileges it strictly requires. This permission enforcement per dependency requires precise identification of dependencies at runtime [1, 9].

Classport facilitates this approach in Java by retrieving dependency metadata (Group ID, Artifact ID, and Version) at runtime, enabling permission managers to make security decisions based on the dependency's inherent characteristics.

3.5.2 Vulnerability Detection at Runtime

Vulnerability detection tools aim to identify known vulnerabilities in third-party libraries. Identifying dependencies that are actually used at runtime is fundamental to avoiding false positives or false negatives in vulnerability assessment [65].

Classport addresses this gap by making it possible to observe exactly which dependencies are executed during a program's run. By embedding GAV metadata into class files and retrieving it dynamically, Classport enables runtime-aware vulnerability detection tools that focus only

on actually used components.

3.6 Related Work

In this section, we position our contribution with respect to related work on embedding dependency metadata into executables and identifying runtime dependencies in Java.

3.6.1 Embedding Information into Executables and Binaries

Boucher et al. [4] propose a novel approach to identifying software dependencies, introducing ABOM, Automatic Bill Of Material. The main idea of ABOM is to query an executable for its dependencies to facilitate vulnerability detection. ABOM automatically embeds hashes of dependencies into compiled binaries during the build process. These hashes are stored efficiently in a probabilistic data structure called Compressed Bloom Filters. It allows for rapid querying to detect the presence of a specific dependency and hence to identify the vulnerabilities associated with it. However, since the data structure is probabilistic in nature, it can report the presence of a dependency even if it is not actually present. Our approach cannot generate false positives because it directly relies on Maven, and a Maven build success implies that all dependencies for the project are resolved.

Seshadi et al. [51] present OmniBOR, which embeds a content-based identifier called gitoid into executables, including Java class files, to link them to an external Artifact Dependency Graph built during compilation. This approach requires build-time instrumentation and external storage. In contrast, our method embeds Maven GAV metadata directly into class files, allowing immediate runtime access via a Java agent without relying on external resolution. Our technique is dedicated to Java and uniquely supports lightweight, runtime introspection of runtime dependencies.

Quatch et al. [46] propose a solution that combines static and dynamic

analysis to identify and remove the unused code during the program loading process. Their framework generates function-level dependency graphs and embeds this information into an optional section of the ELF binary. This embedded dependency information is then used by a piece-wise loader at runtime to identify and remove unused code from the program's memory. Our approach is more general and is not limited to use cases like debloating.

3.6.2 Runtime Identification of Dependencies

In this section, we report works related to the runtime identification of dependencies.

Ponta et al. [41] present a novel code-centric and usage-based method for detecting, assessing, and mitigating vulnerabilities in open-source software dependencies. This work builds upon their previous approaches [39, 40], and is implemented in the open-source tool Eclipse Steady.[12] They use a dynamic call graph of dependencies to check the reachability of the vulnerable code. They instrument the target Java application and search if, during the execution, it reaches the vulnerable code, i.e., if it uses vulnerable methods. The difference with our approach is that they focus on the vulnerable dependencies and not on an overall identification of the actually executed dependencies.

Soto-Valero et al. [58] propose a tool to detect bloated code in Java and to remove it by transforming the bytecode of the compiled project. They use four code coverage tools along with probes to identify methods that are not executed during runtime. If no methods in any class are executed within a dependency, the dependency is considered bloated. They only work at testing time. However, we embed dependency information directly in the class file in JARs, making this information available at production time as well.

Amusuo et al. [1] implement a tool that is able to create a policy file with a mapping of required permissions for every dependency. The policies are then enforced at runtime. To create a policy file, they infer de-

[12] `https://github.com/eclipse-steady/steady`

pendency namespaces by assuming that all classes within a JAR share a common root directory path, which is then mapped to the Maven Group and Artifact ID. In contrast, we directly retrieve dependency information at runtime from metadata embedded in the class files, avoiding fragile assumptions based on package name heuristics.

3.7 Contribution Summary

This Thesis contributes to the runtime identification of dependencies. In particular, we address the issue of the lack of dependency information at runtime in Java by moving dependency information from build time to runtime. We present Classport, a novel approach for runtime introspection of dependencies in Java. Classport embeds dependency information into Java binary artifacts and makes it accessible during execution. We evaluate its effectiveness on six real-world, open-source Maven-based Java projects. For all analyzed projects, Classport successfully embeds complete dependency information and accurately identifies the dependencies exercised at runtime, introducing only low overhead.

Chapter 4

Conclusion

Software Supply Chain security is a critical challenge due to the complexity of modern ecosystems and the lack of accurate visibility over dependencies. This Thesis addressed the issue of improving transparency in the Software Supply Chain by studying and enhancing the accuracy of dependency identification. We approached two sides of the problem: statically, by focusing on the SBOM, and dynamically, concentrating on runtime introspection of dependencies.

From the static perspective, we evaluated existing SBOM generation tools, revealing immaturity and inconsistency in their outputs. We further demonstrated that these shortcomings directly affect vulnerability assessments, leading to incomplete or misleading reports. To address this, we developed a novel SBOM generation methodology, implemented in a proof-of-concept tool, PIP-SBOM. Our evaluation showed substantial gains in both precision and recall over existing solutions.

From the runtime perspective, we propose Classport, a novel technique to introspect dependencies at runtime. We demonstrated its effectiveness by introspecting six real-world, open-source projects. For all of them, it is possible to correctly identify the executed dependencies at runtime, with low overhead.

In summary, the contributions of the Thesis are:

- **Evaluation of SBOM generation tools** – A systematic assessment

of widely used SBOM generation tools, revealing immaturity, inconsistencies, and lack of completeness and correctness in generating SBOMs though static approaches, relevant under resource-critical and isolation constraints.

- **Studying the impact of poor SBOM generation on vulnerability reporting** - A study on the effects that bad SBOMs, i.e., incomplete and incorrect SBOMs, have on the detected vulnerabilities. This reveals that inaccurate SBOMs severely degrade the performance of security tools.

- **Improved SBOM generation methodology** – A new dependency identification approach that directly uses the ecosystem logic to solve the dependencies, achieving higher precision and recall than the other considered tools.

- **Runtime dependency introspection** – Design and implementation of Classport, a lightweight runtime technique that brings build-time dependencies information at runtime, in Java, with low performance overhead.

From an industrial perspective, the proposed techniques can be naturally integrated into modern CI/CD (DevSecOps) pipelines. PIP-SBOM can be executed as part of the build stage, generating a deterministic SBOM directly from the package manager and enabling automated vulnerability scanning before artifact publication. This allows organizations to enforce policy gates (e.g., blocking releases affected by critical vulnerabilities) while improving the reliability of dependency-based security analysis. Classport complements this by extending supply chain transparency beyond build time: dependency metadata embedded during the build phase can be retrieved at runtime to monitor actual dependency usage in production. This enables runtime-aware vulnerability prioritization, reduces noise from unused dependencies, and supports continuous security monitoring. Together, these contributions provide an end-to-end approach to supply chain visibility across development, build,

deployment, and runtime stages, aligning with DevSecOps principles of continuous security integration.

While our findings do not yet achieve complete transparency in the Software Supply Chain, they contribute to raising awareness on the lack of maturity of SBOM as a tool to promote transparency, and on the impact that it can have on security. The two proposed tools provide concrete steps toward enhancing dependency visibility and lay the groundwork for further innovation in this field.

Although this thesis focuses on the Python ecosystem for build-time SBOM generation and on Java for runtime dependency introspection, the underlying challenges and proposed solutions are not ecosystem-specific. The issues identified — including metadata fragmentation, dynamic dependency resolution, and the loss of dependency identity at runtime — are structural characteristics of modern software package ecosystems. Consequently, the methodological framework for assessing SBOM quality, the integration of SBOM generation with native package manager logic, and the approach of embedding dependency metadata into artifacts to enable runtime introspection can be adapted to other ecosystems such as npm/Node.js, Go modules, and similar dependency management environments.

For these reasons, future research could extend this work by exploring other ecosystems, extending our implementation to cover other programming languages. This is important because different programming languages present diverse challenges in identifying dependencies. Also, a prominent continuation of the runtime dependency introspection study is to apply Classport to particular use cases and compare the benefit it brings with respect to the state-of-the-art tools. For instance, it would be interesting to assess its capability to detect dependencies that are declared but not used, known as bloated dependencies, and compare it with existing debloating tools. This would give insights and direction on runtime dependency introspection. Moreover, another promising direction is combining static and dynamic approaches into a unified framework, enabling continuous dependency monitoring from build to production. Also, relaxing the use of uber-JAR may facilitate the adoption

of Classport in broader contexts.

By addressing both static and dynamic aspects of dependency transparency, this Thesis takes an important step toward building a more secure and trustworthy Software Supply Chain, offering both methodological and practical contributions for researchers and practitioners.

AI Usage Disclosure

The usage of AI tools was limited to the grammar checking. The content generated by AI has been reviewed and edited by a human.

Appendix A

Numerical Results from the Comparison of SBOMs

Tables 14 – 17 report the numbers of observed dependencies (Obs), the number of expected ones (Exp), and the number of false positives (FP) and false negatives (FN). Table 18 provides the command line configurations used to generate the results.

Table 13: Ground Truth Dependency Counts for Each Project

Category	Count
Direct dependencies	6
Distinct transitive dependencies	17
Total expected SBOM components	23
Remote dependencies (Git or URL)	2
Unversioned dependencies	2
Constrained dependencies	1
Pinned exact dependencies	2
Imported and used in code	1
Declared but unused	5

Table 14: SBOM vs. ground truth comparison (cdxgen)

Project	Obs	Exp	FP	FN
hatch-hatchling	28	23	5	0
hatch-pdm	28	23	5	0
hatch-setuptools	23	23	5	5
pdm-flit	33	23	10	0
pdm-hatchling	33	23	10	0
pdm-pdm	33	23	10	0
pdm-setuptools	33	23	10	0
pip-hatchling	35	23	12	0
pip-pdm	35	23	12	0
pip-setuptools	35	23	12	0
pipenv-pdm	31	23	10	2
poetry-poetry	33	23	10	0

Table 15: SBOM vs. ground truth comparison (ort)

Project	Obs	Exp	FP	FN
hatch-hatchling	0	23	0	23
hatch-pdm	0	23	0	23
hatch-setuptools	0	23	0	23
pdm-flit	0	23	0	23
pdm-hatchling	0	23	0	23
pdm-pdm	0	23	0	23
pdm-setuptools	0	23	0	23
pip-hatchling	24	23	6	5
pip-pdm	24	23	6	5
pip-setuptools	24	23	6	5
pipenv-pdm	0	23	0	23
poetry-poetry	29	23	8	2

Table 16: SBOM vs. ground truth comparison (syft)

Project	Obs	Exp	FP	FN
hatch-hatchling	0	23	0	23
hatch-pdm	0	23	0	23
hatch-setuptools	1	23	1	23
pdm-flit	0	23	0	23
pdm-hatchling	0	23	0	23
pdm-pdm	0	23	0	23
pdm-setuptools	1	23	1	23
pip-hatchling	1	23	0	22
pip-pdm	1	23	0	22
pip-setuptools	1	23	0	22
pipenv-pdm	25	23	2	0
poetry-poetry	33	23	10	0

Table 17: SBOM vs. ground truth comparison (trivy)

Project	Obs	Exp	FP	FN
hatch-hatchling	0	23	0	23
hatch-pdm	0	23	0	23
hatch-setuptools	0	23	0	23
pdm-flit	0	23	0	23
pdm-hatchling	0	23	0	23
pdm-pdm	0	23	0	23
pdm-setuptools	0	23	0	23
pip-hatchling	2	23	1	22
pip-pdm	2	23	1	22
pip-setuptools	2	23	1	22
pipenv-pdm	26	23	3	0
poetry-poetry	34	23	11	0

Table 18: Configuration and execution parameters of evaluated SBOM generators.

Tool	Execution Environment	Command Invocation
Cdxgen	Docker container ghcr.io/cyclonedx/cdxgen	`cdxgen -r /app/$folder -o ... -t python -deep`
Ort	Docker container ort analyze	`ort analyze -i /app/$folder -o /app/ort-result -f JSON`
Syft	Local CLI execution	`syft scan dir:<project> -o cyclonedx-json=<file>`
Trivy	Local CLI execution	`trivy fs -format cyclonedx -include-dev-deps -list-all-pkgs -output <file> <project>`

Appendix B

Analysis of CycloneDX-python

Despite CycloneDX-python was outside from our original analysis of SBOM generation in the context of the Python ecosystem, we report here the result obtained on that tool by applying the same analysis methodology.

We run CycloneDX-python (version 7.2.2) with the following commands from inside the project directory:

- `cyclonedx-python requirements -i ...` for PIP,
- `cyclonedx-python poetry` for Poetry,
- `cyclonedx-python pipenv` for Pipenv.

The tool parses the metadata files of project, with specific support for Pipenv, Poetry, and PIP.

In detail, for PIP, CycloneDX-python only finds direct dependencies, even if it fails to provide name and version tuples for remote dependencies. In fact, it does not correctly parse the URL. For this reason, the SBOM contains the component associated to the remote dependency with the `unknown` identifies for the name of the remote dependency, as shown by the false positive reported in Table 19.

Table 19: SBOM vs. ground truth comparison (cyclonedx-python)

Project	Obs	Exp	FP	FN
pip-hatchling	6	23	1	18
pip-pdm	6	23	1	18
pip-setuptools	6	23	1	18
pipenv-pdm	25	23	2	0
poetry-poetry	30	23	7	0

For Pipenv, It produces an SBOM that is complete and correct. It implements the parsing logic for the `pyproject.toml` file, and it is also able to find optional and remote dependencies. On the other hand, considering Poetry, CycloneDX-python fails to discover optional dependencies. However, this lack of completeness is due to an implementation feature that requires to flag the command line execution of the tool, explicitly requiring for the inclusion of optional dependencies (i.e., `-with optional`).

Overall, we can conclude that CycloneDX-python does not produce more complete or correct SBOMs than the SBOM generation tools considered in our original analysis.

Bibliography

[1] Paschal C. Amusuo et al. *ZTD$_{JAVA}$: Mitigating Software Supply Chain Vulnerabilities via Zero-Trust Dependencies*. 2024. arXiv: 2310.14117 [cs.CR]. URL: https://arxiv.org/abs/2310.14117.

[2] Anchore. *Grype*. URL: https://github.com/anchore/grype/.

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

[4] Nicholas Boucher and Ross Anderson. *Automatic Bill of Materials*. 2023. arXiv: 2310.09742 [cs.CR]. URL: https://arxiv.org/abs/2310.09742.

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

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

[7] 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/.

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

[9] Carmine Cesarano, Martin Monperrus, and Roberto Natella. *GoLeash: Mitigating Golang Software Supply Chain Attacks with Runtime Policy Enforcement*. 2025. arXiv: 2505.11016 [cs.CR]. URL: https://arxiv.org/abs/2505.11016.

[10] CISA. *Software Bill Of Materials*. 2024. URL: https://www.cisa.gov/sbom.

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

[12] Oscar Cornejo et al. "A family of experiments about how developers perceive delayed system response time". In: *Software Quality Journal* 32.2 (2024), pp. 567–605. DOI: https://doi.org/10.1007/s11219-024-09660-w.

[13] Diego Costa et al. "What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks". In: *IEEE Transactions on Software Engineering* 47.7 (2021), pp. 1452–1467. DOI: 10.1109/TSE.2019.2925345.

[14] 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/.

[15] 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.

[16] DevOps Kung Fu Mafia. *bomber: Scans Software Bill of Materials (SBOMs) for security vulnerabilities*. https://devops-kung-fu.github.io/bomber/. Accessed: 2025-10-13. 2025.

[17] William Enck and Laurie Williams. "Top Five Challenges in Software Supply Chain Security: Observations from 30 Industry and Government Organizations". In: *IEEE Security and Privacy* 20 (2 2022). ISSN: 15584046. DOI: 10.1109/MSEC.2022.3142338.

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

[19] *False Positives and False Negatives in Information Security*. en. https://www.guardrails.io/blog/false-positives-and-false-negatives-in-information-security/. Accessed: 2024-9-5. Aug. 2022.

[20] The Apache Software Foundation. *Maven*. 2025. URL: https://maven.apache.org/.

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

[22] Brian Fox. *Java at 30: From Portable Promise to Critical Infrastructure*. Sonatype Blog. https://www.sonatype.com/blog/java-at-30-from-portable-promise-to-critical-infrastructure. May 2025.

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

[24] Google. *OSV: Open Source Vulnerabilities*. https://osv.dev/. Accessed: 2025-09-11. 2025.

[25] Behnaz Hassanshahi et al. "Macaron: A Logic-based Framework for Software Supply Chain Security Assurance". en. In: *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. Copenhagen Denmark: ACM, Nov. 2023, pp. 29–37. ISBN: 9798400702631. DOI: 10.1145/3605770.3625213. URL: https://dl.acm.org/doi/10.1145/3605770.3625213 (visited on 05/02/2024).

[26] Jqlang. *jqlang/jq: Command-line JSON processor*. en. https://github.com/jqlang/jq.

[27] Joseph R. Biden JR. *Executive Order on Improving the Nation's Cybersecurity*. 2021. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/ (visited on 08/22/2024).

[28] 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.

[29] MITRE Corporation. *CVE: Common Vulnerabilities and Exposures*. https://www.cve.org/. Accessed: 2025-09-09. 1999-2025.

[30] NTIA. *The Minimum Elements For a Software Bill of Materials (SBOM)*.

[31] Openclarity. *kubeclarity*. URL: %5Curl%7Bhttps://github.com/openclarity/kubeclarity%7D.

[32] Oracle. *Instrumentation API*. 2023. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/Instrumentation.html.

[33] Oracle. *Retention*. 2023. URL: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/annotation/Retention.html.

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

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

[36] OWASP Foundation. *Intrusion Detection*. https://owasp.org/www-community/controls/Intrusion_Detection. Accessed: 2025-10-13. 2025.

[37] Serkan Ozkan. *NVD leaves thousands of vulnerabilities without analysis data*. 2024. URL: https://securityscorecard.com/blog/national-vulnerability-database-nvd-leaves-thousands-of-vulnerabilities-without-analysis-data/ (visited on 08/28/2024).

[38] package-url. *purl-spec*. URL: https://github.com/package-url/purl-spec.

[39] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. "Impact assessment for vulnerabilities in open-source software libraries". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 411–420. DOI: 10.1109/ICSM.2015.7332492.

[40] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "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)*. 2018, pp. 449–460. DOI: 10.1109/ICSME.2018.00054.

[41] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "Detection, assessment and mitigation of vulnerabilities in open source dependencies". In: *Empirical Software Engineering* 25 (5 Sept. 2020), pp. 3175–3215. ISSN: 15737616. DOI: 10.1007/s10664-020-09830-x.

[42] Piotr Przymus and Thomas Durieux. "Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack". In: *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE. 2025, pp. 91–102. DOI: 10.1109/MSR66628.2025.00026.

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

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

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

[46] Anh Quach, Aravind Prakash, and Lok Yan. "Debloating software through {Piece-Wise} compilation and loading". In: *27th USENIX security symposium (USENIX Security 18)*. 2018, pp. 869–886. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/quach.

[47] 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.

[48] *Secure Software Supply Chain Center*. URL: https://s3c2.org/.

[49] Contrast Security. *False Negative*. en. https://www.contrastsecurity.com/glossary/false-negative. Accessed: 2024-9-5.

[50] Shift Left Security. *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*. en. URL: %5Curl%7Bhttps://github.com/ShiftLeftSecurity/sast-scan%7D.

[51] Bharathi Seshadri et al. *OmniBOR: A System for Automatic, Verifiable Artifact Resolution across Software Supply Chains*. 2024. arXiv: 2402.08980 [cs.SE]. URL: https://arxiv.org/abs/2402.08980.

[52] Aman Sharma, Benoit Baudry, and Martin Monperrus. *Canonicalization for Unreproducible Builds in Java*. Apr. 2025. DOI: 10.48550/arXiv.2504.21679. arXiv: 2504.21679 [cs]. URL: http://arxiv.org/abs/2504.21679.

[53] Aman Sharma et al. "SBOM.EXE: Countering dynamic code injection based on software bill of materials in Java". In: *arXiv [cs.CR]* (June 2024).

[54] Ax Sharma. *What You Need to Know About the Codecov Incident: A Supply Chain Attack Gone Undetected for 2 Months*. 2021. URL: https://www.sonatype.com/blog/what-you-need-to-know-about-the-codecov-incident-a-supply-chain-attack-gone-undetected-for-2-months.

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

[56] Sonatype. *10th Annual State of the Software Supply Chain*. https://www.sonatype.com/state-of-the-software-supply-chain/introduction. 2024.

[57] Sonatype. *What is a Software Supply Chain?* 2024. URL: https://www.sonatype.com/resources/articles/what-is-software-supply-chain.

[58] César Soto-Valero et al. "Coverage-Based Debloating for Java Bytecode". In: *ACM Trans. Softw. Eng. Methodol.* 32.2 (Apr. 2023). ISSN: 1049-331X. DOI: 10.1145/3546948. URL: https://doi.org/10.1145/3546948.

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

[60] 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.

[61] National Institute of Standards and Technology. *National Vulnerability Database*. https://nvd.nist.gov/. Accessed: 2025-09-11. 2025.

[62] The Apache Software Foundation. *Introduction to the Dependency Mechanism*. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. 2024.

[63] The Apache Software Foundation. *Introduction to the POM*. https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. 2024.

[64] Ilkka Turunen. *Log4Shell by the numbers: Why did CVE-2021-44228 set the internet on fire?* 2021. URL: https://www.sonatype.com/blog/why-did-log4shell-set-the-internet-on-fire.

[65] Laurie Williams et al. "Research Directions in Software Supply Chain Security". In: *ACM Trans. Softw. Eng. Methodol.* 34.5 (May 2025). ISSN: 1049-331X. DOI: 10.1145/3714464. URL: https://doi.org/10.1145/3714464.

[66] 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.

[67] Yue Xiao et al. "JBomAudit: Assessing the Landscape, Compliance, and Security Implications of Java SBOMs". In: *ISOC Network and Distributed System Security Symposium*. 2025. DOI: https://dx.doi.org/10.14722/ndss.2025.240322.

[68] 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.

[69] 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.