JBomAudit: Assessing the Landscape, Compliance, and Security Implications of Java SBOMs

Yue Xiao*†, Dhilung Kirat*, Douglas Lee Schales*, Jiyong Jang*, Luyi Xing†, and Xiaojing Liao†
*IBM Research
Email: xiaoyue@ibm.com, dkirat@us.ibm.com, schales@us.ibm.com, jjang@us.ibm.com
†Indiana University Bloomington
Email: xiaoyue@iu.edu, luyixing@iu.edu, xliao@iu.edu

***Abstract*—A Software Bill of Materials (SBOM) is a detailed inventory that lists the dependencies that make up a software product. Accurate, complete, and up-to-date SBOMs are essential for vulnerability management, reducing license compliance risks, and maintaining high software integrity. The US National Institute of Standards and Technology (NTIA) has established minimum requirements for SBOMs to comply with, especially the correctness and completeness of listed dependencies in SBOMs. However, these requirements remain unexamined in practice. This paper presents the first systematic study on the landscape of SBOMs, including their prevalence, release trends, and characteristics in the Java ecosystem. We developed an end-to-end tool to evaluate the completeness and accuracy of dependencies in SBOMs. Our tool analyzed 25,882 SBOMs and associated JAR files, identifying that 7,907 SBOMs failed to disclose direct dependencies, highlighting the prevalence and severity of SBOM noncompliance issues. Furthermore, 4.97% of these omitted dependencies were vulnerable, leaving software susceptible to potential exploits. Through detailed measurement studies and analysis of root causes, this research uncovers significant security implications of non-compliant SBOMs, especially concerning vulnerability management. These findings, crucial for enhancing SBOM compliance assurance, are being responsibly reported to relevant stakeholders.**

I. Introduction

Software supply chain attacks exploit vulnerabilities in software dependencies, posing a significant threat to organizations worldwide [59, 54, 51]. The recent *XZ Utils backdoor* incident [33], where malicious code was injected into the widely used *XZ Utils*, highlights the severity of these risks. These dependencies are often maintained by small teams yet are widely used, making them attractive targets for attackers. Empirical studies [34, 75, 64] highlight the challenges in managing these dependencies, which exacerbate security vulnerabilities within the supply chain. In response, Executive Order 14028 on Improving the Nation's Cybersecurity, issued by President Biden on May 12, 2021, mandates that software developers provide a Software Bill of Materials (SBOM). An SBOM is a nested inventory, a list of dependencies that make up software components. The US National Telecommunications and Information Administration (NTIA) [21] subsequently released minimum requirements for SBOMs to comply with (see § II-B). SBOMs are crucial for government agencies to enhance software security by identifying and managing supply chain risks [29]. However, without accurate and complete information, an SBOM's illumination and transparency may become sources of consumer confusion and harm. Incorrect or incomplete SBOMs can undermine the effectiveness of vulnerability management tools, mislead security analysts, waste maintenance resources, leave vulnerabilities overlooked, and ultimately exaggerate security risks in the software supply chain.

Extensive research [37, 39, 40, 80, 82, 38, 45, 8, 70, 58, 35, 58] has been conducted to understand the current SBOM practices, including key topics, readiness, benefits, challenges, and solutions. However, there has been little study on the prevalence and characteristics of SBOMs distributed by developers on a large scale over time. Although several tools [27, 28] have been developed to evaluate SBOM quality, they focus primarily on SBOM format (e.g., whether an SBOM includes certain fields like version or license). These tools do not thoroughly analyze the internal quality of SBOMs, such as the correctness and comprehensiveness of the `dependencies` section, which has direct security implications for vulnerability management tools that rely on it. Balliu et al. [37] tried to address this issue to some extent by manually cross-checking the results of six third-party SBOM generators on 26 Java projects. Their findings showed significant variations across the `dependencies` sections in SBOMs. However, the manual examination of a limited number of SBOMs does not provide a holistic and in-depth insight into the quality of SBOM dependencies. In our study, we systematically investigate official SBOMs that are released by developers and distributed through Maven Central [18], a widely used repository for Java packages. By analyzing these official SBOMs, rather than those generated by third-party

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240322
www.ndss-symposium.org

vendors and hosted in third-party databases [32, 23, 25], we aim to provide more realistic insights into the current state of Java SBOMs.

Research Questions. We thoroughly examine the following unaddressed research questions:

RQ1: *What are the current landscape and characteristics of SBOMs distributed by developers in the Java ecosystem?*

RQ2: *How complete and correct are the dependency relationships disclosed in SBOMs and to what extent do they comply with NTIA requirements?*

RQ3: *What are the security implications of non-compliant SBOMs for vulnerability management?*

Methodology. To answer (**RQ1**), we collected all artifacts released by developers from Maven Central. From there, we collected 47,042 SBOMs which we analyzed to understand the trends, scope, and characteristics of SBOMs (§ V). For (**RQ2**), we collected the corresponding distributed code (i.e., JAR files) associated with these SBOMs and employed *JBomAudit* to automatically assess dependency inconsistencies between SBOMs and the distributed code (§ VI). We began by defining a consistency model through a qualitative analysis of 496 SBOM-related discussions, concentrating on the challenges, risks and concerns developers encounter regarding SBOM dependency issues. This analysis informed our adoption of the NTIA's minimum requirements for dependencies [2] and helped us establish an inconsistency model for evaluating SBOM dependency compliance (§ III). Based on our new inconsistency model that formalizes six inconsistencies in SBOM dependencies compliance, we designed and implemented *JBomAudit*, an automated, end-to-end system by utilizing program analysis techniques to unpack the dependencies tree in distributed code and then compare with that listed in SBOMs to output six types of inconsistencies (§ IV). Our thorough evaluation shows that *JBomAudit* can detect SBOM dependency non-compliance effectively and efficiently (§ IV-E). For (**RQ3**), we assessed the security implications of such non-compliant SBOMs by examining their impact on subsequent vulnerability management (§ VII).

Measurement and findings. Our analysis using *JBomAudit* revealed widespread SBOM non-compliance within the Java ecosystem, with significant security implications for vulnerability management across the software supply chain. Specifically, 7,907 SBOMs omitted direct dependencies, averaging 6.18 missing dependencies per SBOM. Additionally, 19,404 SBOMs inaccurately listed an average of 4.45 dependencies. Altogether, 13,394 SBOMs exhibited at least four types of inconsistencies, with each SBOM containing an average of 3.56 inconsistencies (§ VI). These inconsistencies in SBOMs can lead to vulnerabilities being overlooked, leaving software susceptible to potential exploits, or result in the misallocation of resources by addressing non-existent issues. On average, 4.97% of missing dependencies were vulnerable, leading to 507 vulnerabilities being overlooked by SBOM-based vulnerability management tools. Furthermore, 0.28% of incorrectly listed dependencies were vulnerable, resulting in the wasteful allocation of resources to address 105 vulnerabilities in non-existent dependencies (§ VII). We also identified a set of root causes contributing to these issues, including design flaws, misuse of SBOM generators, and incomplete or erroneous metadata (§ VI-B).

Responsible disclosure. We reported our findings to 13 project maintainers regarding 52 missing and 26 incorrect dependencies. So far, 7 maintainers have acknowledged the issues, and 6 have already addressed them in their latest SBOMs. Additionally, we are collaborating with `CycloneDX-Maven-plugin` team to resolve non-compliance issues caused by improper handling of certain `pom.xml` configurations.

Contributions. Our key contributions are summarized below:

- We conducted the first large-scale study of the SBOM landscape in the Java ecosystem, highlighting the prevalence, release trends, and characteristics of official SBOMs.
- We designed and deployed *JBomAudit*, an end-to-end implementation that automatically assesses the correctness and completeness of `dependencies` in SBOMs. *JBomAudit* is based on a formally defined inconsistency model derived from a qualitative analysis of GitHub issue discussions. It can help SBOM practitioners to meet compliance objectives and serve as a resource for auditing purposes. We open source the code at `https://github.com/code-genome/jbomaudit`.
- We conducted comprehensive measurement studies on SBOM non-compliance issues, investigating root causes, and analyzing their security implications. These insights can improve SBOM quality and compliance requirements, helping SBOM providers, consumers, and policymakers better achieve their security and accountability goals.

II. Background

A. Software Bill of Materials

A Software Bill of Materials (SBOM) is a detailed, machine-readable document that lists the attributes, dependencies, and licensing information of software components. Presently, CycloneDX [24], Software Package Data Exchange (SPDX) [30], and Software Identification (SWID) [22] are the three universally recognized SBOM formats, with CycloneDX and SPDX being the most popular [14]. Listing 1 provides a CycloneDX `json` format snippet of an SBOM for the *org.apache.flink* project's `flink-json` component. This SBOM is structured into three main sections: metadata, components, and dependencies. The *metadata* section contains essential details such as the timestamp, publisher, version, and licenses. The *components* section enumerates the nodes, while the *dependencies* section outlines the edges, representing the dependency relationship graph. For example, `flink-json`'s reliance on `flink-table-common`, which in turn depends on the `icu4j` library, highlights the constructed dependency tree.

SBOM Distribution. SBOMs are required to be decoupled from the software and distributed separately [4]. In the Java ecosystem, SBOMs are typically released separately from the

Listing 1: SBOM Example of `flink-json`

```
{
  "bomFormat" : "CycloneDX",
  "specVersion" : "1.4",
  "serialNumber" : "urn:uuid:3db22509-0440-45f0-9ebd...",
  "version" : 1,
  "metadata" : {
    "timestamp" : "2024-03-06T14:55:56Z", ... }
  "components": :[{
      "group" : "com.ibm.icu",
      "name" : "icu4j",
      "version" : "73.2",
      "purl":"pkg:maven/com.ibm.icu/icu4j@73.2?type=jar
           "...]
  "dependencies": [
    {
      "ref" : "pkg:maven/org.apache.flink/flink-json@1
           .19.0?type=jar",
      "dependsOn" : [
        "pkg:maven/org.apache.flink/flink-table-common@1
             .19.0?type=jar", ...]
    }, {
      "ref": "pkg:maven/org.apache.flink/flink-table-
           common@1.19.0?type=jar",
      "dependsOn": [
        "pkg:maven/com.ibm.icu/icu4j@67.1?type=jar", ... ]
    }, ... ]
}
```

JAR file but are located in the same project directory, rather than being packed within the JAR file. In our study, we collected SBOMs from Maven Central [18], a major repository for Java projects, where developers release both the SBOM and JAR in the project directory.

SBOM Usage Scenarios. SBOMs are crucial for enhancing security and transparency throughout the software supply chain. Vulnerability management is the primary use case for SBOMs, as rated by 412 organizations [26]. For instance, software developers use SBOMs to identify vulnerable software dependencies promptly by matching software dependencies against vulnerability databases and issuing warnings when a vulnerable dependency is found. However, any incorrect or missing dependencies in SBOMs can significantly affect the effectiveness and efficiency of vulnerability management. Our research systematically assesses the comprehensiveness and correctness of SBOM dependencies (§ VI) and evaluates the security risks associated with non-compliant SBOMs, as detailed in Section § VII.

SBOM Quality. Several existing tools aim to evaluate SBOM quality, such as `sbom-scorecard` [27] and `sbomqs` [28]. However, these tools only assess format compliance with standards like CycloneDX, SPDX, and SWID, ensuring the inclusion of essential fields such as identifiers, licenses, and versions, but they do not evaluate the accuracy or completeness of the SBOM content. In this work, we go beyond format-level compliance assessments by systematically investigating the internal quality of SBOMs. Guided by the NTIA's minimum requirements for SBOM dependencies, we formally defined a consistency model (§ III) that includes six minimum, atomic, essential inconsistency types between the dependency tree extracted from the code and the one disclosed in the SBOM.

B. Regulatory Compliance and Standards for SBOM

SBOMs are increasingly mandatory in various industries and are considered a critical component of any organization's regulatory compliance strategy. A series of governing bodies and regulators globally [10, 1, 6, 11, 7] have either mandated or recommended organizations provide and maintain accurate and up-to-date SBOMs and use them as a strategy to bolster cybersecurity infrastructure.

NTIA Minimum Requirements. The US National Telecommunications and Information Administration (NTIA) [2] defines a set of minimum requirements that SBOM should comply with. These requirements are divided into two main categories: format compliance and quality compliance. Format compliance specifies that SBOMs must include all necessary data fields, such as the package name, version, unique identifier, licensing information, and dependencies for each component. Additionally, the chosen data formats, like CycloneDX, should facilitate automatic generation while ensuring machine readability and interoperability. We assess these format requirements on our large-scale dataset in § V. Quality compliance requires that the content listed in the SBOM be accurate, complete, and up-to-date. In our study, we focus on the quality of the dependencies section in SBOMs, specifically assessing their correctness and completeness. If any incorrect or missing dependencies are found in an SBOM, we consider it non-compliant with NTIA requirements and flag the SBOM as non-compliant. We systematically assess SBOM quality compliance in § VI and its security implications in § VII.

C. Scope of Study

SBOM Auditing Tools. Our study introduces *JBomAudit*, the first tool designed to audit SBOM compliance issues specifically in the Java ecosystem. Currently, there are no off-the-shelf SBOM auditing tools available for other ecosystems, as SBOM adoption in other programming languages is still in its early stages. Specifically, after manually examining over 1,000 projects from mature dependency management systems (e.g., Cargo for Rust, Go Modules for Go, pip for Python, Conan for C/C++, npm for JavaScript), we found no instances of developer-released SBOMs. Moreover, at the time of this study, only four SBOMs had been released by developers across 377,000 C/C++ projects on GitHub. The lack of developer-released SBOMs for other languages limits such study.

III. Inconsistency Model

NTIA defines *Dependency Relationship* (denoted as $\mathcal{D}$) as characterizing the inclusion of an upstream component $\mathcal{X}$ in software $\mathcal{Y}$. This relationship is typically represented as a hierarchical tree comprised of components that may have sub-components. The `dependencies` section in an SBOM discloses these *Dependency Relationships* within the software. NTIA mandates that *"an SBOM should contain all primary (top-level) components, along with all their transitive dependencies (second-level)."* [21] Therefore, the $\mathcal{D}$ disclosed in an

SBOM ($\mathcal{T}_s$) should align with the actual $\mathcal{D}$ in the distributed software (executable code, denoted as $\mathcal{T}_c$), including both top-level and second-level dependencies in the tree. Discrepancies between $\mathcal{T}_s$ and $\mathcal{T}_c$ at these levels indicate non-compliance with NTIA's disclosure requirements. However, categorizing these inconsistencies, which reflect various quality issues and security risks, remains unclear and is a concern among SBOM practitioners. Our research question focuses on identifying the types of inconsistencies that violate NTIA requirements and are significant to practitioners.

To this end, we conducted a qualitative study by analyzing discussions related to SBOM dependency issues in public GitHub projects to gain insights into the concerns, challenges, and errors encountered in creating accurate and complete dependency relationships within SBOMs. We identified two general types of dependency issues causing non-compliance and six specific types of dependency inconsistencies that do not satisfy NTIA requirements, each reflecting a certain granularity of discrepancy between the declared dependency relationships in SBOMs and the actual dependency relationships in code.

A. Study Procedure

This section outlines our approach to collecting and analyzing SBOM-related discussions to develop an inconsistency model covering various types of inconsistencies.

Stage ❶: Data collection. We utilized a tag-based filtering approach to identify GitHub repositories tagged with SBOM", ensuring relevance in our data sources. Our primary focus was on discussions within GitHub issues, extracting relevant Titles, Questions, and Answers using keywords like SBOM", dependency issues", and quality." These selected discussions were centered around two main themes: (1) *Internal SBOM quality issues:* This included reports of missing or incorrect information within SBOMs, highlighting the potential negative impacts of such inaccuracies on downstream users who rely on SBOMs for security and compliance assessments. (2) *Challenges in tracking third-party components:* Many contributors discussed the difficulties associated with keeping an accurate and up-to-date record of third-party components. This method yielded 51 projects and 496 SBOM-related discussions specifically focusing on dependency issues.

Stage ❷: Qualitative Analysis. Our qualitative analysis followed the Grounded Theory methodology [72], suitable for developing conceptual frameworks in new areas of study [44]. This consisted of three iterative steps:

• *Open Coding:* Conducted by the first two authors, this step involved segmenting the collected discussions into manageable concepts.

• *Axial Coding:* This phase refined the concepts into categories, identifying the main topics and subtopics, reviewed by an additional author for comprehensiveness.

• *Verification:* The coding results were compared using Krippendorff's alpha coefficient, achieving a reliability score of 0.87, higher than the reliability threshold in social science [49].

B. Inconsistency in Dependency Complication

In general, we found developers raise concerns regarding the accuracy and completeness of third-party dependencies listed in SBOMs, given that software systems are typically maintained by multiple developers and undergo a significant volume of code and components changes on a daily basis.

Two types of dependency issues causing non-compliance: We identified two general categories of dependency issues causing non-compliance: missing dependencies and incorrect dependencies, each posing distinct security implications.

• *Missing dependencies* are components used in the distributed code but not listed in the SBOM, leading to potential security oversights. If missing dependencies contain vulnerabilities that are not tracked due to their absence in the SBOM, the resulting security gaps can remain unpatched, leaving the software open for exploitation.

• *Incorrect dependencies* are listed in the SBOM but not used in the distributed code. Resources may be wasted while trying to investigate and fix vulnerabilities for non-existent components. This not only wastes time and resources but also diverts attention and efforts away from real threats and potentially delays the mitigation of critical security risks.

Six types of inconsistencies in dependency complication: We identified three specific types of inconsistencies for each of the two broad dependency issues, examining discrepancies at both the node and edge levels within dependency trees. Missing dependencies are categorized into three distinct types: (1) missing direct dependency ($\mathcal{M}_1$), (2) missing transitive dependency ($\mathcal{M}_2$), and (3) missing transitive relationship ($\mathcal{M}_3$). $\mathcal{M}_1$ identifies direct dependencies omitted from the top layer of $\mathcal{T}_c$, while $\mathcal{M}_2$ refers to transitive dependencies that are absent from the second layer. $\mathcal{M}_3$, defined at the edge level, denotes the absence of a connection between dependencies within $\mathcal{T}_c$. Similarly, incorrect dependencies are divided into three types: (1) incorrect direct dependency ($\mathcal{N}_1$),(2) incorrect transitive dependency ($\mathcal{N}_2$), and (3) incorrect transitive relationship ($\mathcal{N}_3$). Each type reflects a different granularity of discrepancy between the SBOM's documented dependency tree and the actual dependency relationships found in the distributed code. Detailed descriptions and visualization of these SBOM inconsistency types are listed in the accompanying Table I.

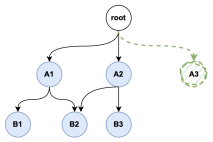
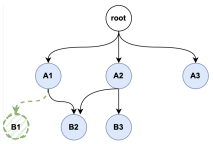
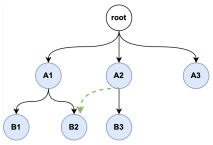
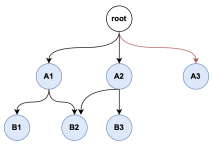
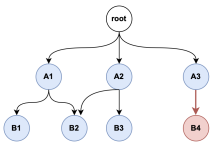
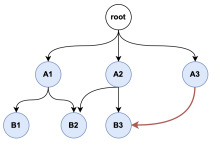
Discussion. Our thorough qualitative analysis ensures that all fundamental issues developers encounter with dependencies in SBOMs are covered. The categorization of six inconsistency types captures the minimum, atomic, and essential aspects of differences between two dependency trees. This consistency model comprehensively covers all complex inconsistencies through combinations of these six basic types and is extensible to handle deeper layer inconsistencies (see Appendix § A).

IV. *JBomAudit*: Design and Implementation

A. Overview

As outlined in Figure 1, the design of *JBomAudit* includes three major components: *Data Collection (§ IV-B)*, *Dependency Tree Extraction* (*DTE*, § IV-C), and *Compliance check*

TABLE I: Summary of Six Inconsistency Types in Dependency Complication with Visual Representations and Formal Definitions. $\mathcal{L}_1$ means the top layer in dependency tree and $\mathcal{L}_2$ represents the second layer in dependency tree; $u \equiv v$ means their gid and aid are congruent, i.e., $gid_u = gid_v$ and $aid_u = aid_v$ where gid is `groupId` and aid is `artifactId`.

Type	Description	Figure	Formal Definition	Security Implication
$\mathcal{M}_1$	$\mathcal{M}_1$ occurs when a node v in $\mathcal{L}_1$ of $\mathcal{T}_c$ lacks a corresponding node in $\mathcal{L}_1$ of $\mathcal{T}_s$. This inconsistency indicates that a component, such as `A3`, used directly in the code is not documented as a direct dependency in the SBOM.		**Inconsistency 1** ($\not\models_{\bigcirc}$ Missing Direct Dependency)**.** $\exists v \in \mathcal{L}_1(\mathcal{T}_c) : \nexists u \in \mathcal{L}_1(\mathcal{T}_s) : u \equiv v \Rightarrow \mathcal{T}_s \not\models_{\bigcirc} v$	The omission of direct dependencies, such as `A3`, from the SBOM may result in overlooking vulnerabilities in `A3` (if present) by security analysis tools that depend on accurate SBOM data.
$\mathcal{M}_2$	$\mathcal{M}_2$ is identified when a transitive dependency v, which is a node in $\mathcal{L}_2$ of $\mathcal{T}_c$, does not have a corresponding node in $\mathcal{L}_2$ of $\mathcal{T}_s$. For example, a library `B1` required by another library `A1` is missing in the SBOM's detailed dependency tree.		**Inconsistency 2** ($\not\models_{\bigcirc}$ Missing Transitive Dependency)**.** $\exists v \in \mathcal{L}_2(\mathcal{T}_c) : \nexists u \in \mathcal{L}_2(\mathcal{T}_s) : u \equiv v \Rightarrow \mathcal{T}_s \not\models_{\bigcirc} v$	The omission of transitive dependencies like `B1` from the SBOM can also expose systems to security risks by leaving vulnerabilities in transitive components unaddressed.
$\mathcal{M}_3$	$\mathcal{M}_3$ is detected when a transitive dependency has multiple parents, but not fully disclosed all dependent relationship in SBOM. For example, the software has direct dependencies `A1` and `A2` and `A1` and `A2` both have `B2` as a transitive dependency, then SBOM dependency tree will describe `B2` as only being a dependency of `A1` and will ignore it also being a dependency of `A2`.		**Inconsistency 3** ($\not\models_{\bigcirc}$ Missing Transitive Relationship)**.** $\exists v \in \mathcal{L}_2(\mathcal{T}_c), u \in \mathcal{L}_2(\mathcal{T}_s) : (u \equiv v) \wedge (P(v) > P(u)) \Rightarrow \mathcal{T}_s \not\models_{\bigcirc} v$	This cause a problem when `B2` has a vulnerability, and `A1` has an available update that includes a fixed version of `B2`, while `A2` does not. The dependency graph might misleadingly suggest that updating `A1` alone is sufficient to address the vulnerability. However, this is not the case. Only after the upgrade is complete and a new SBOM is generated, the dependency graph accurately reflects `A2`'s reliance on `B2`.
$\mathcal{N}_1$	$\mathcal{N}_1$ denotes a node u in $\mathcal{L}_1$ of $\mathcal{T}_s$ that does not correspond to any actual use in $\mathcal{L}_1$ of $\mathcal{T}_c$, often arising from outdated or incorrect documentation. For example, the listed component `A3` is not utilized in the distributed code.		**Inconsistency 4** ($\not\models_{\triangle}$ Incorrect Direct Dependency)**.** $\exists u \in \mathcal{L}_1(\mathcal{T}_s) : \nexists v \in \mathcal{L}_1(\mathcal{T}_c) : u \equiv v \Rightarrow \mathcal{T}_s \not\models_{\triangle} u$	Listing an incorrect direct dependency such as u can cause security tools to expend efforts on non-existent vulnerabilities, misdirecting focus from actual security risks within the code.
$\mathcal{N}_2$	$\mathcal{N}_2$ denotes an incorrectly documented transitive dependency u within $\mathcal{L}_2$ of $\mathcal{T}_s$ that has no counterpart in $\mathcal{L}_2$ of $\mathcal{T}_c$. For instance, the SBOM erroneously lists `B4` as a transitive dependency of `A3`, despite there being no such dependency in the distributed code.		**Inconsistency 5** ($\not\models_{\triangle}$ Incorrect Transitive Dependency)**.** $\exists u \in \mathcal{L}_2(\mathcal{T}_s) : \nexists v \in \mathcal{L}_2(\mathcal{T}_c) : u \equiv v \Rightarrow \mathcal{T}_s \not\models_{\triangle} u$	The inclusion of non-existent transitive dependencies can also lead to a misallocation of security efforts, neglecting actual threats in the software and impairing the integrity of vulnerability management processes.
$\mathcal{N}_3$	$\mathcal{N}_3$ identifies when a transitive dependency u in $\mathcal{L}_2$ of $\mathcal{T}_s$ is inaccurately documented with an dependency relationship in the SBOM compared to its distributed code in $\mathcal{L}_2$ of $\mathcal{T}_c$. For instance, although `A3` does not depend on `B3` in the actual code, the SBOM erroneously documents such a dependency.		**Inconsistency 6** ($\not\models_{\triangle}$ Incorrect Transitive Relationship)**.** $\exists u \in \mathcal{L}_2(\mathcal{T}_s), v \in \mathcal{L}_2(\mathcal{T}_c) : (u \equiv v) \wedge (P(u) > P(v)) \Rightarrow \mathcal{T}_s \not\models_{\triangle} u$	Over-claiming transitive relationships can lead to insufficient vulnerability management and patch application. Security analysts might expend unnecessary efforts seeking compatible versions of `B3` for `A2` and `A3`, despite only needing to address `A2`.

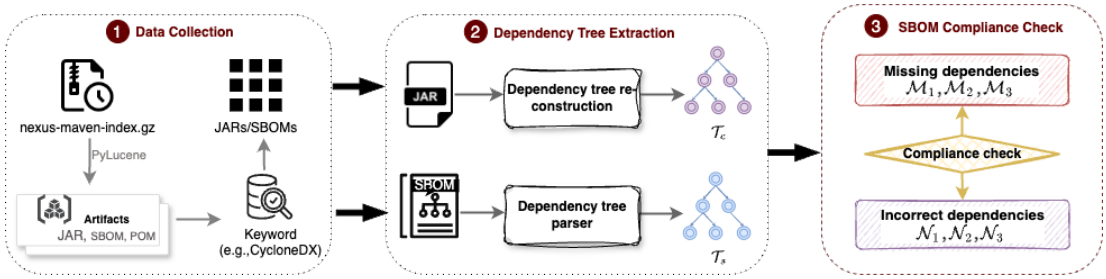


Fig. 1: Overview of *JBomAudit*.

(*SCC*, § IV-D). Particularly, in the *Data Collection* component, *JBomAudit* first collects 25,882 SBOMs and corresponding binaries (i.e., JARs) from Maven, spanning the period from June 2023 to April 2024. After that, *DTE* extracts the depen-

TABLE II: Summary of datasets and corpora (MC is short for Maven Central)

Name	Source	Size	Timestamp	Usage
S_{total}	MC	47,042 SBOMs	2023-2024	Detection
S_i	MC	25,882 SBOMs	2023-2024	Detection
J_i	MC	25,882 JARs	2023-2024	Detection
M_j	MC	606,840 Maven Packages	2023-2024	Detection
P_j	MC	25,882 POMs	2023-2024	Measurement
G_t	MC	3,704 target profiles	202404	Measurement
G_d	MC	1,315 dependency profiles	202404	Measurement
N_v	NVD	719 vulnerabilities	202404	Measurement

dency tree ($\mathcal{T}_c$) directly from the JAR files, using *JarPkgTags* (a binary analysis tool detailed in § IV-C). Additionally, *DTE* parses the corresponding SBOM to extract the claimed dependency tree, represented as $\mathcal{T}_s$. In the final analysis, *SCC* performs a comparative evaluation between $\mathcal{T}_c$ and $\mathcal{T}_s$, identifying any inconsistencies as specified in Section III.

Example. To demonstrate the functionality of *JBomAudit*, we analyze JAR file: `env-var-0.0.2.jar`. Initially, we download the JAR file and its corresponding SBOM (`env-var-0.0.2-cyclonedx.json`) from the Maven repository. *DTE* then extracts the dependency tree from the binary code, resulting in a tree that includes 1 direct node in the first layer and 2 transitive nodes in the second layer. Concurrently, *DTE* processes the `dependencies` section in the SBOM, producing a dependency tree with 3 direct nodes in the first layer and 17 transitive nodes in the second layer. Lastly, *SCC* takes these two trees as input and applies a Breadth-First Search (BFS) Layer Comparison to identify discrepancies. The analysis outputs two incorrect direct dependencies ($\mathcal{N}2$) and one missing transitive dependency ($\mathcal{M}3$).

B. Dataset collection

We summarize the dataset produced and consumed by each stage of our detection pipeline as below. Table II shows the datasets used in our study. In total, we collected 47,042 SBOMs (S_{total}) provided by developers from Maven Central [18], a major repository for Java projects. More specifically, we downloaded the latest "nexus-maven-repository-index.gz" from Maven, which indexes all artifacts within the repository. Using `index-cli-6.2.0`, we unpacked this index into a raw Lucene format and utilized `PyLucene's IndexReader` and `IndexSearcher` to efficiently query for SBOMs. Our search employed keywords such as "CycloneDX", "SPDX", "sbom", "cdx", focusing on artifact names containing these terms along with either a `.json` or `.xml` extension. The total SBOMs (S_{total}) were used to measure their prevalence and characteristics within the SBOM ecosystem (see measurement results in § V).

The JAR file, which the SBOM describes, is typically released in the same project directory on Maven Central. However, we observed that not all SBOMs (S_{total}) had a corresponding JAR file released by the developer on Maven. We found that only 55% of the SBOMs (S_i) had corresponding JARs, resulting in 25,882 JARs (J_i). The S_i and J_i further served as inputs for dependency tree extraction (§ IV-C) and SBOM compliance check (§ IV-D) to detect inconsistencies (see measurement results in § VI). Furthermore, we collected 606,840 Maven packages (M_j) hosted on Maven to construct the *Package Name to Dependency Mapping* database (§ IV-C).

C. Dependency Tree Extraction

The dependency tree $\mathcal{T}s$ is constructed from an SBOM by traversing the *dependencies* section, which lists all edges. This section focuses on Dependency Tree Extraction from the distributed code. We developed *JarPkgTags*, which processes `JAR` files to extract top-layer direct and second-layer transitive dependencies, denoted as $\mathcal{L}1(\mathcal{T}c)$ and $\mathcal{L}2(\mathcal{T}c)$ respectively. The tool then compares these layers with the SBOM to identify inconsistencies as outlined in Section III.

1) Utilized Java Classes Extraction: A Java application or library is encapsulated within a `JAR` file (Java ARchive), which employs the standardized ZIP file format. The contents of a `JAR` file typically include: (1) Metadata files such as `pom.xml` and `MANIFEST.MF`, which provide critical information about the application, including dependencies. (2) Compiled Java code, consisting of class files that execute the application's functionality. These metadata files are manually curated by developers. Due to the frequency of code updates and the extensive reliance on third-party libraries, these files are susceptible to containing outdated, incomplete or incorrect information, which is not a reliable source to extract dependencies from. Conversely, the Java class files, generated by the Java compiler, serve as a definitive source of "ground truth." They accurately reflect the actual code utilization within the application, thereby providing a reliable basis for extracting the actual dependencies. Our approach analyzes Java bytecode to capture reference from external classes, methods, annotations and also covers those introduced by dynamic features. This process involves scanning the bytecode to identify references that indicate the use of external Java classes.

Source ❶: External classes. The primary category of utilized Java classes that our analysis targets encompasses those originating from external packages. These classes are identified through their references within the constant pool of the Java bytecode. Specifically, the constant pool stores these references as fully qualified names, for example, `java.util.List`, which uniquely identify each external class. The Java class file constant pool consists of a table of variable length tagged value entries. We scan the constant pool looking for entries tagged as `ClassReference`. These entries contain an index back into the constant pool to an entry tagged as `UTF8String`, which is the fully qualified class name.

Source ❷: Method parameters. Another source of utilized external Java classes can be identified through method parameters. Specifically, if these parameters are never used except as arguments to methods in classes that are external to the jar file, then the constant pool will not have a class reference. Thus,

we also scan all of the parameters for all methods defined in the class to look for additional utilized classes.

Source ❸: Annotations. Annotations represent a critical static source for class references within Java applications. Java annotations are defined using Java classes and are handled at compile time and are stored as attributes associated with classes, fields, and methods. Because they are normally handled at compile time, they are not normally a run-time dependency. However, because annotations can generate code at compile time, it is necessary that they be included in the SBOM. At the class file level, examples of these annotation attributes are `RuntimeVisibleAnnotations`, `RuntimeVisibleTypeAnnotations`, and `RuntimeVisibleParameterAnnotations` as well as the `Invisible` counterparts. Our method examines all attributes to identify and extract these annotations. We then decode each annotation to obtain the fully qualified class name that defined the annotation.

Source ❹: Dynamic features. The final step in our analysis involves extracting classes accessed via Java's dynamic class loading API, which allows applications to load external classes at runtime using Java strings to specify class names. While these dynamically referenced classes do not appear in the constant pool as external references, non-computed string constants specifying class names do. Although scanning the constant pool for string constants formatted as fully qualified class names can lead to false positives, a more reliable approach involves analyzing the usage of the Java reflection API. Primarily, we focus on the dynamic loading APIs, such as `java.lang.Class.forName()` method, which requires a string containing the class name as its argument. Our method considers 33 class loaders, catalogued in [20], detailing their usage and parameter positions.

For each class file, we scan the constant pool for references to these dynamic loading APIs. If found, we then examine the Java bytecode of the class file, searching for invocations of any dynamic loading APIs. Given that the JVM operates on a stack-based mechanism, we simulate bytecode execution to ascertain inputs to targeted methods. We assume that the input value is pushed onto the stack shortly before the method call, typically via the `ldc` or `ldc_w` instructions which transfer constants from the constant pool to the stack. For calls to dynamic loading APIs, we monitor the stack for these inputs, discarding other non-essential data. When a target method call is identified, and if preceded by an `ldc` or `ldc_w` instruction, we extract the class name from the stack for further processing.

To improve the resolution rate, we also integrated state-of-the-art dynamic feature resolution tools such as Soot [73] and WALA [31]. The combination of Soot in CHA mode and WALA in CFA mode has been validated against the most challenging dynamic feature test cases as detailed in the study by Reif et al. [66].

2) Dependency Tree Construction: After retrieving the utilized class files, our next goal is to identify the dependencies that provided these files. Initially, we determine the package containing the class file. If this package originates from an external dependency, we classify it as a directly used dependency, constructing the node in the first layer of the dependency tree ($\mathcal{L}_1(\mathcal{T}_c)$)). Subsequently, we recursively construct the second layer ($\mathcal{L}_2(\mathcal{T}_c)$).

Step ❶: Class Name to Package Name Transformation. After identifying the utilized class name, we convert it to its corresponding package name, which is crucial for pinpointing any external dependency that provides it. Class file names consist of a series of labels separated by dots ("."), with the class name typically being the last label in the sequence. To isolate the package name, we remove this last label, thereby retaining the sequence that represents the package name.

Step ❷: Internal vs. External Package Determination. We then ascertain whether the package name is internally provided by the current JAR or derived from external dependency. This involves cataloging all packages contained within the JAR under investigation. If the utilized class file does not belong to any internally provided package, we conclude that it corresponds to an external source.

Step ❸: Package Name to Dependency Mapping. To identify which external dependency provides a utilized package, we maintain a comprehensive Maven database mapping package names to dependencies, denoted as $M : K \rightarrow V$, where K represents a dependency identified by `PURL` and V denotes the set of package names it provides. This is necessary because at the code level, we only extract package names, which cannot be directly compared with SBOM dependencies represented by `PURL`, a standardized URL format used to identify and locate software packages in an SBOM. To bridge this gap, we established a detailed mapping from package names to `PURL`. We achieved this by collecting all Maven packages currently hosted on Maven, each identifiable by `PURL`. Using the *JarPkgTags*, we extracted package name details from these artifacts. This process allowed us to establish a correlation between package names and the `PURL` of the external dependencies.

Step ❹: Dependency tree construction. Finally, we can obtain all utilized direct dependencies (i.e., $\mathcal{L}_1(\mathcal{T}_c)$). *JarPkgTags* further analyzes these direct dependencies to identify second-layer transitive dependencies (i.e., $\mathcal{L}_2(\mathcal{T}_c)$), which are served as inputs to the compliance check module.

D. SBOM Compliance Check

The `dependencies` section in an SBOM enumerates edges, each representing a dependency relationship between two components. We analyze this section to construct a declared dependency tree for the first two layers. To identify inconsistencies as defined in Section III, we employ a Breadth-First Search (BFS) Layer Comparison approach between $\mathcal{T}c$ and $\mathcal{T}s$. Initially, we compare the first layer of both trees. We identify missing direct dependencies ($\mathcal{M}1$), which are present in $\mathcal{T}c$ but absent in $\mathcal{T}s$. Conversely, incorrect direct dependencies ($\mathcal{N}1$) are noted if they appear in $\mathcal{T}s$ but are missing from $\mathcal{T}c$. Next, we analyze the second layer by extracting and comparing the subtrees, $\mathcal{T}'c$ and $\mathcal{T}'s$, which share the same parent node. We traverse the nodes within

these two subtrees to identify discrepancies at both the node and edge levels. (1) Node-Level Discrepancies: We check for missing or incorrectly formed nodes in $\mathcal{T}'s$. If discrepancies are identified, we categorize them as missing transitive dependencies ($\mathcal{M}2$) or incorrect transitive dependencies ($\mathcal{N}_2$), based on whether they are absent or malformed, respectively. (2) Edge-Level Discrepancies: For each node, we first retrieve all inbound edges from the original trees. These edges are then compared to identify any mismatches: missing edges are classified as missing transitive dependencies ($\mathcal{M}2$), and incorrect edges as incorrect transitive dependencies ($\mathcal{N}2$). It is important to note that if any node in $\mathcal{T}s$ is identified as an incorrect dependency, its subtree is not compared against $\mathcal{T}c$. This is because there is no corresponding subtree in $\mathcal{T}_c$, given that the root node is already marked as an incorrect dependency.

E. Evaluation of the JBomAudit System

Experiment settings. We deployed *JBomAudit* on a server running Ubuntu 20.04.6 LTS. The server was equipped with 20 CPU cores, 251 GB of RAM, and a storage capacity of 50 TB. This experiment settings are used to evaluate the effectiveness and performance of *JBomAudit*.

1) Ground truth dataset.: To evaluate *JBomAudit*'s efficacy in detecting various inconsistency types, we established two ground truth datasets: one featuring real-world Java projects and another comprising specialized test cases designed to challenge *JBomAudit* across different Java development practices, such as dynamic features, custom build scripts, and manual dependency inclusion.

• *Real-world java projects.* We randomly selected 30 Java projects in the wild. A team of three Java experts, boasting development experience spanning three, five, and twenty-five years, invested 100+ hours in total to construct a reliable ground truth for dependencies. This comprehensive process involved: (1) Decompiling .class files into Java source code using a tool named `jadx` [12]; (2) Manually collating all utilized classes from direct imports located in the header of the class file, implicit reference or dynamically loaded classes embedded in the code. (3) Manually mapping these classes to the project's dependency tree via the Maven Dependency Plugin [19] or, if unmatched, searching the Maven repository for any library providing those classes; (4) Searching for manually included or custom-built JARs, WARs, and RARs within the project; (5) Contrasting the identified dependency graph against SBOMs to pinpoint inconsistencies. Overall, the team annotated 259 direct dependencies and, through recursive analysis, 432 transitive dependencies.

• *Test cases.* Our test suite consists of 21 test cases, each crafted to assess *JBomAudit*'s capability in dealing with Java development practices and language features. We segmented our test cases into three categories: (1) *Multiple-build Systems (ST00-ST03)*: We constructed four test cases that can be built under different build systems (e.g., OSGI, Maven) to investigate whether *JBomAudit* is capable of detecting all dependencies included by these various build systems; (2) *Customized Build Settings (ST04-ST08)*: We created five cases to test the tool's proficiency in recognizing custom build configurations, including manually included dependencies, multi-module dependencies, and varied Maven plugin applications. (3) *Dynamic Features (TR9-CFNE1)*: We built 13 cases that utilized Java's dynamic capabilities to load classes, like Classloading, Dynamic Proxies, and reflection techniques, to verify if *JBomAudit* can effectively identify dependencies used dynamically. This diverse test suite aims to evaluation the capability or potential insufficiency of *JBomAudit* in specific Java development contexts. Details for each test case are provided in [20].

TABLE III: Overall effectiveness of *JBomAudit*: S_g and D_g are the manually identified non-compliant SBOMs and dependencies, respectively. S_t and D_t are the outputs of *JBomAudit*. R_a is the average recall, and P_a is the average precision.

Type	S_g	D_g	S_t	D_t	**Recall**	**Precision**	R_a	P_a
$\mathcal{M}_1$	9	60	9	59	91.66%	93.22%		
$\mathcal{M}_2$	19	132	22	132	91.66%	91.66%	92.30%	91.74%
$\mathcal{M}_3$	17	140	18	145	93.57%	90.34%		
$\mathcal{N}_1$	14	58	15	57	79.31%	80.70%		
$\mathcal{N}_2$	8	20	9	21	80.00%	76.19%	81.31%	78.49%
$\mathcal{N}_3$	6	52	5	56	83.61%	78.57%		

2) Evaluation results.: Running *JBomAudit* on 30 real-world Java projects, the tool demonstrated an average precision of 91.74% and a recall of 92.30% in detecting missing dependencies and 78.49% and a recall of 81.31% in detecting incorrect dependencies. Our tool shows strong performance in detecting missing cases, which have significant security implications (see § VII). Additionally, when tested against 21 specific cases, *JBomAudit* successfully passed 16 of them, showcasing its capability to handle the complicated dependencies.

Real-world Java projects evaluation. In this analysis, *JBomAudit* identified 59, 132, and 145 missing dependencies categorized as $\mathcal{M}1$, $\mathcal{M}2$, and $\mathcal{M}3$ respectively; and 57, 21, and 56 incorrect dependencies categorized as $\mathcal{N}1$, $\mathcal{N}2$, and $\mathcal{N}3$ respectively. The precision and recall for each type of inconsistency are detailed in Table III. Notably, *JBomAudit* demonstrated high performance in detecting missing dependencies. However, it exhibited lower effectiveness in identifying incorrect dependencies. This challenge arises from the difficulty in confirming that classes listed by incorrect dependencies are truly not used in anywhere of the code, unlike the more straightforward verification of used classes when identifying missing dependencies.

• *Falsely detected inconsistency.* We observed 57 instances of falsely detected inconsistencies. First, 28 of these were instances where *JBomAudit* incorrectly flagged dependencies as non-dependent, even though they were actually utilized in the code. This is mainly because the challenges in resolving dynamic features. Specifically, 17 dependencies used reflection techniques across five Java projects for purposes like extensibility, testing, and legacy issues. These dynamic features, often context-sensitive, necessitate a complicated inter-procedural control-/data-flow analysis for accurate resolution. Other errors arose from the failure to resolve annotations or methods

referenced in the constant pool. Second, in left 29 instances, the tool incorrectly identified dependencies as missing from the SBOM, although they were accurately disclosed. This was due to those dependencies being non-public or unavailable, which led to incomplete mappings in the *Package Name to Dependency Mapping* process (step ❸ in § IV-C2). Consequently, the utilized classes could not be correctly matched to the dependencies that provided them.

• *False negatives.* We observed 49 instances of false negatives, among which 25 were missing dependencies and 24 are incorrect dependencies. False negatives of missing dependencies detection means that although these dependencies were actually utilized in the code, *JBomAudit* failed to detect them. False negatives of incorrect dependencies detection means that although these dependencies were incorrectly disclosed in SBOM, *JBomAudit* failed to detect them. To investigate the reasons, we found that inefficiencies in dynamic feature resolution also led to missed detections. Similarly, failures to correlate detected class names with dependencies in our *Package Name to Dependency Mapping* contributed to these false negatives.

• *Discussion.* To address the potential false positives and negatives caused by unresolved dynamic features, *JBomAudit* flags detection results as `undetermined` and labels the corresponding class files that contain unresolved dynamic elements, allowing developers to investigate them deeper. Additionally, to handle errors resulting from the incompleteness of the *Package Name to Dependency Mapping* database, *JBomAudit* similarly flags the results as `undetermined` and identifies the absent libraries, giving developers the option to manually upload the absent libraries for further investigation. Overall, *JBomAudit* flagged 95,835 results as `undetermined`, and these cases were excluded from subsequent measurement studies (§ V, § VI and § VII).

Test cases evaluation. *JBomAudit*'s performance on 21 test cases revealed its capability to detect dependencies introduced by multiple build systems and customized settings, as well as partially resolve dynamic features. As indicated in Table IV, *JBomAudit* fell short in five test cases involving complex, context-sensitive reflection or dynamic loading scenarios that require advanced analysis or runtime data retrieval.

3) Comparison with other SCA tools.: To date, no publicly available tools offer verification of the accuracy and completeness of a dependency tree within an SBOM. We benchmarked *JBomAudit* against three leading software composition analysis (SCA) tools, referenced in our study as [3, 5, 9]. Notably, *JBomAudit* specializes in the analysis of official SBOMs during the Release & Deliver stage, utilizing binary-level scrutiny. For a fair comparison, we ensured that the other three tools also conducted binary scans. These scans encompass a range of techniques, such as parsing manifest files, comparing cryptographic hashes, and matching file names, to pinpoint dependencies in the same stage of release and delivery.

• *Effectiveness.* To assess effectiveness, we utilized the three SCA tools on a carefully curated ground-truth dataset to reconstruct dependency trees from binaries. We then applied the inconsistency definitions outlined in § III to identify six types of inconsistencies. To provide a holistic view of effectiveness, we aggregated the results across six inconsistencies and calculated the average precision and recall for detecting non-compliant dependencies. The comparative results are detailed in Table V. At the non-compliant SBOM level, all three tools successfully identified the majority of non-compliant SBOMs, each containing at least one non-compliant dependency. However, at the non-compliant dependency level, all tools demonstrated lower recall, though they maintained comparable precision. Among the three, Steady was the most effective, leveraging a code-centric and usage-based detection method, whereas the others primarily relied on parsing manifest files, which are prone to errors and can become outdated.

• *Performance Overhead.* We evaluated the performance overhead of *JBomAudit* against three other SCA tools. The comparative results, detailed in Table V, show average processing times of 5.36 s for Dep-check, 8.79 s for Steady, 2.26 s for T1, and 2.03 s for *JBomAudit*. These findings indicate that all tools maintain a relatively low performance overhead, with *JBomAudit* achieving the fastest processing time. To demonstrate the scalability of *JBomAudit*, we ran *JBomAudit*

TABLE IV: Comparison of tools in test cases

Tests	*JBomAudit*	Dep-check	Steady	T1
ST00	✓	✗	✗	✗
ST01	✓	✗	✓	✗
ST02	✓	✗	✓	✗
ST03	✓	✓	✓	✗
ST04	✓	✗	✗	✗
ST05	✓	✓	✗	✓
ST06	✓	✓	✓	✓
ST07	✓	✗	✓	✗
ST08	✓	✓	✗	✓
TR9	✓	✗	✗	✗
LRR1	✓	✗	✗	✗
LRR2	✗	✗	✗	✗
CSR1	✓	✗	✗	✗
CSR2	✗	✗	✗	✗
CSR3	✗	✗	✗	✗
CSR4	✗	✗	✗	✗
CL1	✓	✗	✗	✗
CL2	✓	✗	✗	✗
CL3	✓	✗	✗	✗
CL4	✗	✗	✗	✗
CFNE1	✓	✗	✗	✗

TABLE V: Comparison effectiveness with three SCA tools. S_g are the number of non-complaint SBOMs and D_g are non-complaint SBOMs dependency.

Tools	S_g	D_g	Precision	Recall	F1	Time
Dep-check	9	114	74.56%	24.67%	78.8%	5.36 s
Steady	17	286	81.81%	61.90%	70.48%	8.79 s
T1	13	232	75.86%	50.21%	60.43%	2.26 s
JBomAudit	22	470	87.87%	89.39%	88.62%	2.03 s

across 300 select real-world Java projects of varying sizes and complexities. Figure 6 illustrates the performance trend as tree size increases, showing that the overhead grows linearly with the number of nodes. This indicates that *JBomAudit* can handle even complex projects in practice (see details in Appendix § B).

V. Ecosystem Overview of SBOM

In this section, we summarize the insights from our examination of official SBOMs for open-source Java projects, discussing their prevalence and characteristics.

A. Pervasiveness of SBOM

We aim to investigate the scope and magnitude of current SBOM implementation and adoption, and to provide insights into trends or patterns in the release of SBOMs that could inform future policy and practice.

Scope and magnitude. A CISA-backed study [71] advises that SBOMs should be accessible in central repositories. We assessed SBOM distribution on Maven, a key hub for Java projects. As detailed in § IV-B, our approach identified 47,042 SBOMs, with 25,882 having corresponding JAR files.

Release timeline. Our investigation of the SBOM release timeline seeks to understand the industry's response to the cybersecurity executive order issued by the Biden Administration on May 12, 2021 and subsequent NTIA guidelines on minimum SBOM requirements. As depicted in Figure 2, SBOM release activity before February 2022 was notably sparse, signifying a considerable delay in the industry's reaction to the May 2021 executive order. However, March 2022 saw a marked uptick in activity corresponding with the introduction of the CycloneDX 1.4 format. The trend continued to show periodic expansion, culminating in a notable peak in late 2023. This was subsequent to the OWASP Foundation's collaboration with Ecma International, emphasizing a global drive toward software transparency and standardization. This increase suggests a collective industry effort to comply with the new cybersecurity standards. Another significant increase occurred in March 2024, following CISA's *SBOM-a-Rama* event, which aided the broader software and security community in understanding and adopting SBOM practices. Overall, the trend shows an increasing trend with some fluctuations, reflecting the evolving landscape of SBOM release practices and showing that SBOM practices are becoming increasingly common and established.

B. Characteristics of Official SBOMs

Earlier studies [80, 78, 70] have engaged in qualitative research to gauge the benefits, challenges, and discussions surrounding SBOMs from developer's perspective. Nevertheless, a void exists in the literature where quantitative analyses are required to scrutinize the characteristics of SBOMs in depth.

Minimum SBOM requirements. The NTIA has delineated the essential constituents [2] of an SBOM in its publication "Minimum Elements for an SBOM", formulated to fulfill the objectives stipulated in Executive Order 14028. Below we evaluate the extent to which current SBOMs align with these minimum mandated requirements.

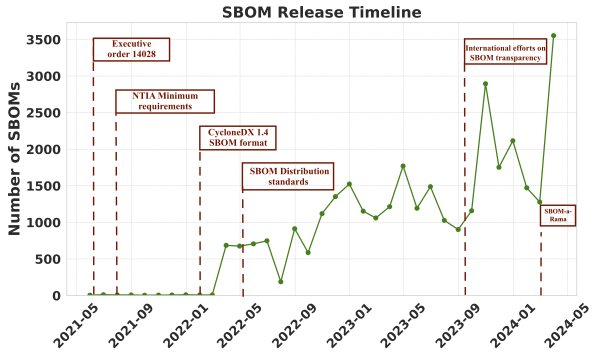


Fig. 2: SBOM release timeline. The y-axis represents the new added SBOM monthly.

• *Data fields*. The NTIA specifies seven required fields as foundational for SBOMs: *Supplier Name, Component Name, Component Version, Unique Identifiers, Dependency Relationships, SBOM Data Author, and Timestamp*. These fields constitute the core information for tracking each software component. Our analysis checks for the presence of these mandatory fields in the SBOMs collected, thereby gauging compliance with the minimum standards. Our analysis reveals compliance rates for key fields: *Supplier Name* at 91.24%, *Timestamp* at 86.91%, and *Dependency Relationship* at 93.46%. All other required fields met 100% compliance in the SBOMs analyzed, as detailed in [20].

• *Automation Support*. The NTIA requires SBOMs to be machine-readable, using formats such as SPDX, CycloneDX, and SWID tags to support automation. Our evaluation verified the adoption of these formats. All SBOMs in our study were machine-readable, with 98.19% in CycloneDX format and 1.81% in SPDX.

SBOM characteristics. Dependency relationships and licensing information are critical in SBOMs, facilitating effective vulnerability management and minimizing licensing risks.

• *Dependency Graph*. Understanding the structure of dependencies in SBOMs is essential for evaluating their complexity and effective management. We analyzed SBOMs by constructing dependency graphs, measuring the size and levels of transitive dependencies. The average number of dependencies is 70.93 (SD = 115.64), and the average depth of dependency chains is 9.46 (SD = 7.52).

• *License Usage*. License analysis within SBOMs sheds light on the legal compliance of software usage. We collected variety of licenses reported in the SBOMs and identified 222 distinct types of licenses. The five most common licenses observed were Apache-2.0, EPL-1.0, EPL-2.0, MIT, and GPL-2.0.

Answer to RQ1: Our study identified 47,042 SBOMs in Maven, showing an increasing trend in SBOM releases and reflecting the evolving landscape of SBOM practices. These SBOMs mostly comply with NTIA's minimum format requirements. The dependency graphs vary significantly in size, and 222 distinct licenses are used, highlighting the diversity of SBOMs.

VI. Inconsistency Analysis and Results

In this section, we analyze the landscape and characteristics of the identified inconsistencies within SBOMs (§ VI-A) and subsequently investigate their underlying root causes (§ VI-B).

A. Landscape

Scope and magnitude. Utilizing *JBomAudit*, we analyzed 25,882 JARs alongside their corresponding SBOMs, uncovering widespread inconsistencies defined in § III. Table VI summarizes our findings: 7,907 SBOMs failed to include direct dependencies, averaging 6.18 dependencies per SBOM, while 19,404 SBOMs inaccurately reported an average of 4.45 dependencies. The prevalence of errors in direct dependencies is particularly concerning, as it directly compromises SBOM usability and affects the performance of dependent downstream tools. We also uncovered numerous inaccuracies within transitive dependencies; for instance, 23,362 SBOMs were found to have missing at least one transitive dependency. In total, 13,394 SBOMs exhibited at least four types of inconsistencies, with an average SBOM containing 3.56 inconsistencies, and a standard deviation of 1.26. Remarkably, 918 SBOMs displayed every category of inconsistency identified in our study. The widespread nature of these issues underscores the critical need for rigorous compliance checks and quality audits of SBOMs to ensure their reliability. To understand the potential impact of projects releasing non-compliant SBOMs, we analyzed their profiles on the Maven repository [13], considering factors such as rankings, categories/tags, and license usage (see [20] for details).

Missing Dependencies in SBOMs. Inconsistencies such as types $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ highlight critical dependencies that are utilized in the code but not listed in the SBOMs. Such omissions pose significant security risks, potentially allowing vulnerabilities to go undetected during SBOM-based security assessments. Our analysis reveals that 88.57% of JARs utilize these missed dependencies in multiple locations within their codebase, with an average reference count of 20.65 times per dependency. This frequent usage underscores a substantial attack surface, which could be exposed to security threats if vulnerabilities associated with these dependencies are not recognized and addressed. Figure 3 illustrates the top 15 missing dependencies, ranked by their omission frequency across various Java projects. Notably, `com.google.guava:guava` emerges as the most frequently omitted direct dependency. Remarkably, every version of this component released before June 09, 2023, is known to contain at least one vulnerability, posing significant security risks if overlooked. Additionally, `com.google.code.findbugs:annotations`, `org.eclipse.jdt:org.eclipse.jdt.annotation`, and `org.slf4j :slf4j-api` also show the high number of missing transitive dependencies, suggesting a trend of complexity or oversight in managing dependencies that are not directly connected to the primary codebase. Detailed analysis of these security implications will be presented in § VII.

TABLE VI: Overall results of *JBomAudit*

Type	#SBOMs	#Dependencies	Average	Std
$\mathcal{M}_1$	7,907	48,931	6.18	10.95
$\mathcal{M}_2$	23,362	309,286	13.23	20.65
$\mathcal{M}_3$	21,665	365,897	16.88	36.33
$\mathcal{N}_1$	19,404	86,483	4.45	3.81
$\mathcal{N}_2$	6,140	14,830	2.41	2.02
$\mathcal{N}_3$	11,168	101,745	9.11	18.25

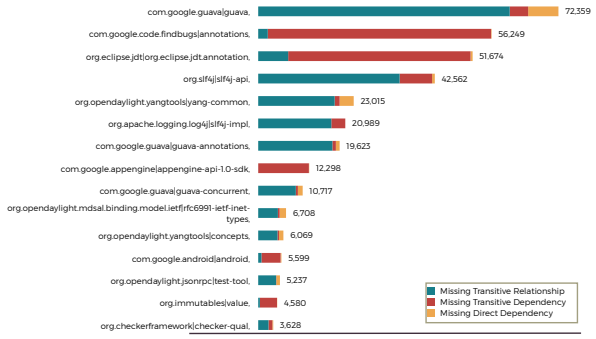


Fig. 3: Top 15 missing dependencies

Incorrect Dependencies in SBOMs. Types $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ denote incorrect dependencies listed in the SBOMs but not actually used within the codebase. Addressing these discrepancies is essential for effective vulnerability management, patching processes, and license compliance check, as they can lead to the misallocation of resources, unnecessary security efforts, and inaccurate compliance results. Figure 4 displays the top 15 incorrectly listed dependencies, identified based on the frequency of errors across Java projects. These dependencies are commonly associated with categories such as Annotation Libraries, Logging Frameworks, and Utilities. Often regarded as auxiliary components that do not affect core functionality, these categories are prone to mismanagement.

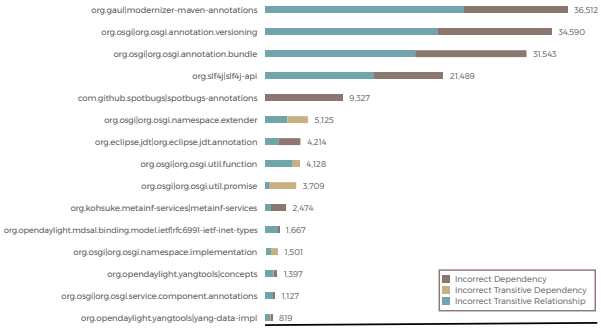


Fig. 4: Top 15 incorrect dependencies

B. Root Cause Analysis

In this section, we investigate the underlying reasons for the non-compliance observed in SBOMs.

Insufficient Design in SBOM Generators. Among the 47,042 SBOMs analyzed, we found that 99.9% were produced by *CycloneDX Maven plugin* and *cyclonedx-gradle-plugin*, with only 43 SBOMs generated by other tools. A significant finding is that these tools rely exclusively on a single specific metadata file to generate SBOMs. The *CycloneDX Maven plugin* utilizes `pom.xml`, while *cyclonedx-gradle-plugin* depends on `build.gradle`. This design is well-suited for projects solely utilizing a single building framework. However, it proves inadequate for complex projects that may declare dependencies across multiple metadata files, such as OSGi manifests. In fact, our study identified that 7.29% of the missing dependencies were OSGi bundles specified in OSGi manifest files, which are overlooked by these tools.

Incomplete and Erroneous Metadata. Compounding the challenges associated with reliance on a single metadata source, the situation is further aggravated by the incompleteness and errors found within the singular metadata file (e.g., `pom.xml`). Any inaccuracies or omissions in the metadata files inevitably affect the quality of the resulting SBOMs. To assess this impact, we examined the correlation between identified missing or incorrect dependencies and their presence in the POM file.

• *Incomplete Metadata*. The incomplete specification of dependency relationships in `pom.xml` files directly leads to omissions in SBOMs generated by metadata-dependent tools. Our analysis revealed that 77.25% of direct dependencies, 39.86% of transitive dependencies, and 97.94% of transitive relationships absent from SBOMs were also missing from the corresponding POM files. These findings highlight a significant deficiency in metadata provisioning, which substantially contributes to the gaps observed in SBOM completeness.

• *Erroneous Metadata*. Further, our analysis identified that 85.34% of incorrect dependencies, 97.46% of incorrect transitive dependencies, and 98.04% of erroneous transitive relationships reported in SBOMs were also inaccurately listed in the corresponding POM files. This strong correlation underscores the significant impact of inaccurate metadata on SBOM accuracy, highlighting the importance of careful maintenance of the `pom.xml` file to ensure its correctness.

Incorrect Usage of SBOM generators. Improper use of SBOM tools by developers can also result in non-compliant SBOMs. Typically, developers are expected to configure tools like the *CycloneDX Maven plugin* as `plugins` in the main `pom.xml` that governs the entire project. However, our analysis identified instances where developers mistakenly configured SBOM tools within the `pom.xml` of a sub-module, rather than the main project file. This results in SBOMs that only reflect the dependencies of that sub-module, omitting others and leading to incomplete SBOMs. To evaluate these discrepancies, we ran the SBOM generator on the main `pom.xml` within the JAR and compared the dependencies listed with those in the initially collected SBOM. Differences between these two SBOMs might suggest incorrect usage of the tool and our analysis identified 1,236 instances of such cases.

Misunderstanding SBOM Requirements during the Application Phase. We observed instances where developers erroneously included `Provided` and `Test` dependencies in SBOMs, which were not actually utilized in the final distributed JAR. Such dependencies, irrelevant to the final product, should not be featured in SBOMs at the application phase for several reasons: (1) `Provided` dependencies are supplied by the runtime environment of the project and therefore do not need to be included in the final artifact, and (2) `Test` dependencies are not incorporated into the final JAR as they are used solely during the testing phase. In our analysis, a substantial 82.69% of the incorrectly listed dependencies were `Provided` dependencies, while only 0.12% were associated with `Test` dependencies.

> **Answer to RQ2:** Our study reveals a significant number of SBOMs that misrepresent dependencies—both by omitting dependencies used in the code and by incorrectly listing dependencies that are never used. This highlights serious internal quality issues within the dependencies section of SBOMs, indicating non-compliance with NTIA's disclosure requirements. The root causes of these issues are diverse, including insufficient design in SBOM generators, incomplete and erroneous metadata, incorrect usage of SBOM generators, and misunderstanding of SBOM requirements.

VII. Security Implication

Non-compliant SBOMs can cause two types of security consequences. Firstly, SBOMs lacking complete dependency listings can lead to overlooked vulnerabilities, leaving software susceptible to potential exploits. Secondly, incorrect dependency listings may result in the misallocation of resources towards addressing non-existent issues. In this section, we assess the security implications of such non-compliant SBOMs by examining their impacts on vulnerability management.

A. Design

We selected Dependency-Track [16] for our evaluation, an OWASP-endorsed vulnerability management tool known for its comprehensive API and leadership in vulnerability intelligence. Our method begins by submitting non-compliant SBOMs to Dependency-Track to identify initial vulnerabilities, denoted as $\mathcal{V}_0$. To gauge the impact of missing dependencies, we update the SBOM to include these dependencies and rerun the analysis, producing a revised vulnerability list, $\mathcal{V}_1$. Vulnerabilities overlooked due to missing dependencies are quantified by $\Delta V_{\text{missed}} = \mathcal{V}_1 - \mathcal{V}_0$. Similarly, we eliminate incorrectly listed dependencies from the SBOM and conduct another analysis to derive $\mathcal{V}_2$. The vulnerabilities erroneously addressed, indicating resource misallocation, are captured by $\Delta V_{\text{false}} = \mathcal{V}_0 - \mathcal{V}_2$.

B. Case Study

This section presents two real-world case studies to illustrate the security implications of non-compliant SBOMs.

• $\Delta V_{\textbf{missed}}$. Consider the Apache project `flight-sql-jdbc-driver`, which released SBOM (`flight-sql-jdbc-driver-12.0.0-cyclonedx.json`) along with a corresponding JAR (`flight-sql-jdbc-driver-12.0.0.jar`). A critical missing direct dependency ($\mathcal{M}_1$), named `logback`, was identified within the JAR file. Specifically, the `flight-sql-jdbc-driver-12.0.0.jar` utilizes the class `org.slf4j.impl.StaticLoggerBinder` for binding to a logging framework. This functionality is provided by the `logback` dependency, identified by the purl: `pkg:maven/ch.qos.logback/logback-classic@1.2.3?type=jar`, however it is absent from the SBOM. The absence of `logback` is significant: it is associated with a direct vulnerability rated as "High" severity (CVSS score: 7.5), and includes 20 additional vulnerabilities from its dependencies, six of which are rated as "Critical". The omission of `logback` in the SBOM caused Dependency-Track to overlook these vulnerabilities, posing considerable security risk for vulnerability management.

• $\Delta V_{\textbf{false}}$. Consider `hbase-examples` project from `org.apache.hbase`, which released SBOM `hbase-examples- 2.5.3-cyclonedx.json` along with the JAR `hbase-examples-2.5.3.jar`. We identified an incorrect direct dependency ($\mathcal{N}_1$), named `zookeeper`, with the purl `pkg:maven/org.apache.zookeeper:zookeeper@3.5.7?type=jar`. None of the classes from `zookeeper` were used in `hbase-examples-2.5.3.jar`, yet it was incorrectly listed in the SBOM. This `zookeeper` dependency contains a critical vulnerability with a CVSS score of 9.1. Including this non-existent vulnerable dependency can waste maintenance resources and reduce the efficiency of vulnerability management.

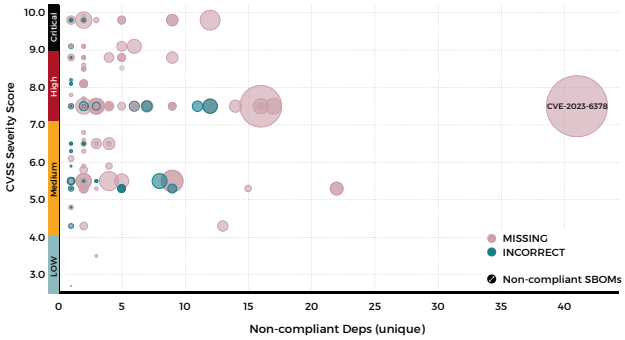


Fig. 5: The distribution of vulnerabilities severity within non-compliant dependencies/SBOMs.

C. Measurement

We conducted a comprehensive analysis to assess the extent of security risks associated with non-compliant SBOMs from multiple perspectives:

• **SBOM-level Analysis**. We categorized the non-compliance issues into two groups: *MISSING* for missing dependencies ($\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$) and *INCORRECT* for erroneously listed dependencies ($\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$), which allows us to analyze the prevalence and impact of each group to SBOM holistically. Our result indicates that 14.75% of SBOMs involves missing dependencies that contains vulnerabilities, whereas only 2.19% of SBOMs list incorrect dependencies with vulnerabilities. Our findings indicate that missing dependencies pose more substantial security risks than incorrect ones, underscoring the need for SBOM practitioners to prioritize the completeness of dependency listings to enhance overall security.

• **Dependency-level Analysis**. Breaking down the security risks by type of non-compliance at dependency level, we find that missing transitive relationships (i.e., 13.88%) ($\mathcal{M}3$) are most likely to involve security risks. These omissions can lead to a false sense of security, especially when a transitive dependency is updated to a secure version in one direct dependency but remains vulnerable in another dependency due to their neglected dependent relationship in SBOM. In contrast, incorrectly listed dependencies ($\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$) show a lower incidence of associated vulnerabilities. More evaluation details are listed in [20].

• **Vulnerability-level Risk Assessment.** We utilized a bubble chart to illustrate the severity distribution of vulnerabilities associated with non-compliant dependencies and their impact on SBOMs. In this chart, each bubble represents a vulnerability; the y-axis shows the CVSS severity score, and the x-axis indicates the number of unique, non-compliant dependencies linked to each CVE. The size of the bubbles reflects the number of unique, non-compliant SBOMs affected by these vulnerabilities. As depicted in Figure 5, a significant majority of vulnerabilities (66.34%) are rated as "High" or "Critical," underscoring the severe security risks posed by non-compliant dependencies. Each vulnerability, on average, impacts 3.5 dependencies and affects 42.75 SBOMs. A notable example is

CVE-2023-6378, classified as "High" severity (CVSS score: 7.5), which influences 41 dependencies and exposes 1,944 SBOMs to potential security breaches.

Answer to RQ3: Non-compliant SBOMs pose significant security implications for vulnerability management by either overlooking vulnerabilities associated with missing dependencies (14.75%) or causing false alerts with vulnerabilities in incorrect dependencies (2.19%). Additionally, the majority of these vulnerabilities (66.34%) are severe, underscoring the substantial security risks posed by non-compliant dependencies.

VIII. Discussion

A. Limitations

Our approach, based on static analysis, faces inherent challenges due to obfuscation techniques [17] and dynamic language features [46, 52, 66] employed in Java code. First, to assess the potential impact of obfuscation techniques on our study, we ran `Deobfuscator` [15], a tool designed to detect obfuscation in `JAR` files, on our dataset. Our results indicated that none of the Java projects used obfuscation techniques. This could be attributed to our focus on open-source Java projects in Maven, where there is little motivation to apply such techniques compared to their use in the Android ecosystem [36, 77, 79] for intellectual property protection or malware to evade detection [60, 67, 41, 42]. Second, regarding dynamic feature resolution, although *JBomAudit* integrates state-of-the-art techniques to address dynamic language features, our evaluation revealed limitations in precisely identifying dynamically loaded classes whose names are dynamically computed (e.g., through string building), supplied as parameters, or require additional runtime external configurations. Third, the incompleteness of our *Package Name to Dependency Mapping* database, due to the absence of non-public or unavailable dependencies, can lead to false positives or negatives. To address these limitations, our tool flags detection results as `undetermined` when dynamic features or non-public dependencies remain unresolved, allowing developers to investigate further (see discussion in § IV-E). Notably, these undetermined results are excluded from the measurement studies to ensure the soundness of our findings.

Furthermore, it is challenging to identify the version of dependencies based solely on JAR binaries because the JVM class loader does not support versioning, meaning Java class files only contain package names without version information. Version details are dynamically determined by the loading environments of dependencies, which are managed by downstream projects. As we do not have access to these environments, this aspect is beyond the scope of our study. However, to tackle this as much as possible, we downloaded the specific-version dependency listed in the SBOM to ensure the detected incorrect dependencies are version-specific. For missing dependencies, we retrieved version information from manifest files (e.g., pom.xml). We tied 47.75% of cases (345,799) to specific versions, while the remaining 378,315 cases were compared only at package-level. These cases were excluded from security analysis in § VII to ensure reliability, making our results a lower bound of SBOM issues.

B. Recommendations for Stakeholders

Based on our findings, we suggest recommendations for stakeholders to enhance the quality and utility of SBOMs.

- **SBOM Generators and SCA Tools:** The primary SBOM generators in the Java ecosystem (e.g., CycloneDX Maven plugin and CycloneDX Gradle plugin) mainly rely on metadata to generate SBOMs, often neglecting code-level analysis. This can lead to low quality of SBOMs due to erroneous metadata. While existing SCA [3, 5, 9] tools analyze dependencies at the code level, they often fail to incorporate advanced techniques for resolving dynamic features. Therefore, it is desirable SCA tools enhance their technical capabilities that can be integrated into SBOM generators to produce higher-quality SBOMs.
- **Developers:** It is ideal all dependencies that are used during development, including dynamic ones and manually packaged classes, are accurately documented in the metadata. Additionally, it is recommended to thoroughly review the documentation of SBOM generators and follow the best practices to ensure proper usage and accurate SBOM generation. Regular audits and updates of SBOMs can ensure maintaining their accuracy over time.
- **Downstream users:** Incorrect and incomplete dependency data in SBOMs can cause confusion and potential harm for downstream users who rely on them for vulnerability management. Therefore, it is crucial to audit the quality of SBOMs before its usage. By integrating automated verification tools like *JBomAudit* into the SBOM usage workflow, organizations can validate and address low-quality SBOMs early. This ensures more reliable and transparent dependency information, enhancing the accuracy of vulnerability assessments.

IX. Related Work

SBOM practice, design and deployment. Extensive research has been conducted on the current state of SBOM practices [37, 39, 40, 82, 38, 45, 80, 8, 70, 58, 35, 58, 65], exploring key topics, readiness, benefits, challenges, and solutions through various methods. These include data mining of 4,786 SBOM-related discussions on GitHub [38], hosting security summits with 30 industry and government organizations [45], and conducting user studies with SBOM practitioners [80, 70, 8]. Recent studies have also focused on the design and generation of SBOMs. Martin et al. [53] proposed a minimalistic SBOM format and Shripad et al. [57] introduced a pipeline for micro-service application SBOMs before and after building. Additionally, the lifecycle management of SBOMs has been studied [55, 56], with a focus on integrating security measures such as access control and encryption into the CI/CD pipeline, and the application of SBOMs in risk assessment processes [48, 74].

Balliu et al. [37] examined the quality of SBOMs produced by six different Java SBOM producers on 26 Java projects. Yu

et al. [81] perform differential analysis of the correctness of four popular SBOM generators on projects written in Python, Ruby, PHP, Java, etc. Our work differs from theirs in key aspects, including (1) *Data Source*: [37, 81] studied third-party-generated SBOMs while we analyzed developer-released SBOMs; (2) *Methodology*: [37, 81] conducted differential analysis between SBOMs without complete ground truth, whereas we used program analysis to identify discrepancies between code and SBOMs; (3) *Analysis Depth*: [37, 81] combined all dependencies into a list for comparison, whereas we analyzed six types of dependency inconsistencies at the tree-level with formal definitions (§ III); (4) *Results*: [37, 81] revealed significant variations in SBOM generators, whereas we identified significant non-compliant SBOMs according to NTIA requirements. Besides, Our study uncovers the root causes of these inconsistencies (§ VI-B) and their security implications (§ VII).

Dependency and vulnerability management. Vulnerable dependencies can introduce serious problems in the software ecosystems [34, 75, 64, 61, 47, 83, 63]. Studies provide the evidence that developers often do not update software dependencies in Java ecosystem [50, 76]. Soto-Valero, César, et al. [69] conducted a detailed investigation into the prevalence of bloated dependencies in Maven artifacts, examining how unused dependencies evolve over time in numerous single-module Maven projects [68]. Chuang [43] introduced a decision-making framework to aid developers in determining whether to remove specific dependencies, based on results from static analysis tools. Ponta et al. [62] evaluated the ability of three debloating tools to distinguish which dependency classes are necessary for an application to function correctly from those that could be safely removed. Pashchenko et al. [61] conducted a qualitative study through semi-structured interviews with 25 developers to gain insights into their approaches to selecting, updating, and managing dependencies, and their strategies for mitigating vulnerable dependencies. Different from previous work, our study focuses on whether SBOMs accurately declare dependency relationships for software products and the extent of security implications brought by incomplete or incorrect SBOMs when they are involved in vulnerability management.

X. Conclusion

This paper presents the first systematic study to investigate whether official SBOMs in open-source Java projects comply with the NTIA's minimum requirements for dependencies and assess their quality in terms of accuracy and completeness. By collecting and analyzing 25,882 SBOMs and associated JARs from Maven, spanning from June 2023 to April 2024, we revealed that 13,394 SBOMs exhibited at least four inconsistencies. This indicates widespread non-compliance issues within these official SBOMs in open-source Java projects. Our findings describe the landscape and characteristics of the identified inconsistencies. We further uncover a set of root causes, including design errors, misuse of SBOM generators, and incomplete or erroneous metadata. Our study brings new insights into the security implications of non-complaint SBOMs, particularly vulnerability management, essential to enhance SBOM compliance assurance.

Acknowledgment

We thank the shepherd and anonymous reviewers for their valuable and constructive feedback. This work is supported by the National Science Foundation (CNS-2330265, 2339537) and the Luddy Faculty Fellowship.

References

[1] Improving the nation's cybersecurity: Nist's responsibilities under the may 2021 executive order. https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity, 2021.

[2] The minimum elements for a software bill of materials (sbom). https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf, 2021.

[3] Owasp dependency-check project - owasp. https://owasp.org/www-project-dependency-check/, 2021.

[4] 15 ways to make sbom distribution easy. https://www.datatrails.ai/sbom-distribution-manifesto/, 2022.

[5] Eclipse steady. https://github.com/eclipse/steady, 2022.

[6] Ncsc issues fresh guidance following recent rise in supply chain cyber attacks. https://www.ncsc.gov.uk/news/ncsc-issues-fresh-guidance-following-recent-rise-in-supply-chain-cyber-attacks, 2022.

[7] Recommendations to improve the resilience of canada's digital supply chain. https://ised-isde.canada.ca/site/spectrum-management-telecommunications/sites/default/files/attachments/2022/CFDIR-June2022-recommendations.pdf, 2022.

[8] The state of software bill of materials (sbom) and cybersecurity readiness. https://www.linuxfoundation.org/research/the-state-of-software-bill-of-materials-sbom-and-cybersecurity-readiness, 2022.

[9] Black duck software composition analysis (sca) - synopsys. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html, 2023.

[10] Cybersecurity and infrastructure security agency, software bill of materials, usa. https://www.cisa.gov/sbom, 2023.

[11] Guidelines for software development. https://www.cyber.gov.au/resources-business-and-government/essential-cyber-security/ism/cyber-security-guidelines/guidelines-software-development, 2023.

[12] jadx. https://github.com/skylot/jadx, 2023.

[13] Mvn repository. https://mvnrepository.com/artifact/, 2023.

[14] Comparing sbom standards: Spdx vs. cyclonedx. https://blog.sonatype.com/comparing-sbom-standards-spdx-vs.-cyclonedx-vs.-swid, 2024.

[15] Deobfuscator. https://github.com/java-deobfuscator/deobfuscator, 2024.

[16] Dependency track. https://dependencytrack.org/, 2024.

[17] Java obfuscator. https://www.javatpoint.com/java-obfuscator, 2024.
[18] Maven central. https://repo1.maven.org/maven2/, 2024.
[19] Maven dependency plugin. https://maven.apache.org/plugins/maven-dependency-plugin/, 2024.
[20] More details about paper appendix. https://zenodo.org/records/14278840, 2024.
[21] Ntia sbom minimum elements report. https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf, 2024.
[22] Nvd-swid-national institute of standards and technology. https://nvd.nist.gov/products/swid, 2024.
[23] The open-source curation database. https://www.osselot.org/, 2024.
[24] Owasp cyclonedx software bill of materials (sbom) standard. https://spdx.dev/, 2024.
[25] A public repository of sbom. https://sbombenchmark.dev/, 2024.
[26] Sbom and cybersecurity readiness. https://8112310.fs1.hubspotusercontent-na1.net/hubfs/8112310/LF%20Research/State%20of%20Software%20Bill%20of%20Materials%20-%20Report.pdf, 2024.
[27] sbom-scorecard. https://github.com/eBay/sbom-scorecard, 2024.
[28] sbomqs. https://github.com/interlynk-io/sbomqs, 2024.
[29] Software bill of materials (sbom). https://security.cms.gov/learn/software-bill-materials-sbom, 2024.
[30] The software package data exchange (spdx). https://cyclonedx.org/, 2024.
[31] Static analysis framework for java. https://github.com/wala/WALA, 2024.
[32] Third-party sbom databases. https://app.soos.io/research/packages/NuGet/-/python/, 2024.
[33] xz-utils-backdoor. https://www.csoonline.com/article/2077692/dangerous-xz-utils-backdoor-was-the-result-of-years-long-supply-chain-compromise-effort.html, 2024.
[34] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 28(3):59, 2023.
[35] Anvesh Ambala. Exploring the dynamics of software bill of materials (sboms) and security integration in open source projects, 2024.
[36] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–9, 2018.
[37] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, (01):2–13, aug 5555.
[38] Tingting Bi, Boming Xia, Zhenchang Xing, Qinghua Lu, and Liming Zhu. On the way to sboms: Investigating design issues and solutions in practice. *ACM Transactions on Software Engineering and Methodology*, 2023.
[39] Seth Carmody, Andrea Coravos, Ginny Fahs, Audra Hatch, Janine Medina, Beau Woods, and Joshua Corman. Building resilient medical technology supply chains with a software bill of materials. *npj Digital Medicine*, 4(1):34, 2021.
[40] Peter J Caven, Shakthidhar Reddy Gopavaram, and L Jean Camp. Integrating human intelligence to bypass information asymmetry in procurement decision-making. In *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*, pages 687–692. IEEE, 2022.
[41] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. {Obfuscation-Resilient} executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3451–3468, 2021.
[42] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.
[43] Ching-Chi Chuang, Luís Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. Removing dependencies from large software projects: are you really sure? In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 105–115. IEEE, 2022.
[44] Klaas Andries de Graaf, Peng Liang, Antony Tang, Willem Robert van Hage, and Hans van Vliet. An exploratory study on ontology engineering for software architecture documentation. *Computers in Industry*, 65(7):1053–1064, 2014.
[45] W. Enck and L. Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(02):96–100, mar 2022.
[46] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 209–220, 2018.
[47] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 172:110653, 2021.
[48] Sagar Gupta and Sailaja Vadlamudi. Open-source software security challenges and policies for cloud enterprises. In *2023 3rd International Conference on Intelligent Communication and Computational Techniques (ICCT)*, pages 1–6. IEEE, 2023.
[49] Andrew F Hayes and Klaus Krippendorff. Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89, 2007.
[50] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration.

Empirical Software Engineering, 23:384–417, 2018.

[51] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[52] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[53] Robert Alan Martin. Visibility & control: addressing supply chain challenges to trustworthy software-enabled things. In *2020 IEEE Systems Security Symposium (SSS)*, pages 1–4. IEEE, 2020.

[54] Jeferson Martínez and Javier M Durán. Software supply chain attacks, a threat to global cybersecurity: Solarwinds' case study. *International Journal of Safety and Security Engineering*, 11(5):537–545, 2021.

[55] Nabil M Mohammed, Mahmood Niazi, Mohammad Alshayeb, and Sajjad Mahmood. Exploring software security approaches in software development lifecycle: A systematic mapping study. *Computer Standards & Interfaces*, 50:107–115, 2017.

[56] Shripad Nadgowda. Engram: the one security platform for modern software supply chain risks. In *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds*, pages 7–12, 2022.

[57] Shripad Nadgowda and Laura Luan. tapiserí: Blueprint to modernize devsecops for real world. In *Proceedings of the Seventh International Workshop on Container Technologies and Container Clouds*, pages 13–18, 2021.

[58] Sabato Nocera, Simone Romano, Massimiliano Di Penta, Rita Francese, and Giuseppe Scanniello. Software bill of materials adoption: A mining study from github. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 39–49. IEEE, 2023.

[59] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[60] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[61] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1513–1531, 2020.

[62] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 555–558. IEEE, 2021.

[63] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

[64] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26:1–34, 2021.

[65] Md Fazle Rabbi, Arifa Islam Champa, Costain Nachuma, and Minhaz Fahim Zibran. Sbom generation tools under microscope: A focus on the npm ecosystem. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1233–1241, 2024.

[66] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–261, 2019.

[67] Monirul Islam Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[68] César Soto-Valero, Thomas Durieux, and Benoit Baudry. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1021–1031, 2021.

[69] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):45, 2021.

[70] Trevor Stalnaker, Nathan Wintersgill, Oscar Chaparro, Massimiliano Di Penta, Daniel M German, and Denys Poshyvanyk. Boms away! inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[71] Jeremiah Trent Stoddard, Michael Adam Cutshaw, Tyler Williams, Allan Friedman, and Justin Murphy. Software bill of materials (sbom) sharing lifecycle report. 4 2023.

[72] Anselm L Strauss and Juliet M Corbin. *Grounded theory in practice*. Sage, 1997.

[73] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[74] Hans-Martin von Stockhausen and Marc Rose. Continuous security patch delivery and risk management for medical devices. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 204–209. IEEE, 2020.

[75] Wentao Wang, Faryn Dumont, Nan Niu, and Glen Horton. Detecting software security vulnerabilities via requirements dependency analysis. *IEEE Transactions on Software Engineering*, 48(5):1665–1675, 2020.

[76] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[77] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th annual computer security applications conference*, pages 222–235, 2018.

[78] Dominik Wermke, Jan H Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. " always contribute back": A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1545–1560. IEEE, 2023.

[79] Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with {TIRO}. In *27th USENIX security symposium (USENIX security 18)*, pages 1247–1262, 2018.

[80] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642, 2023.

[81] Sheng Yu, Wei Song, Xunchao Hu, and Heng Yin. On the correctness of metadata-based sbom generation: A differential analysis approach. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 29–36. IEEE, 2024.

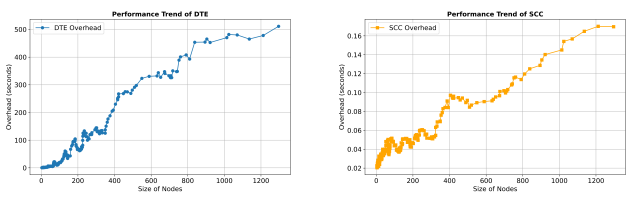
[82] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 21(2):82–88, 2023.

[83] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, 2019.

APPENDIX

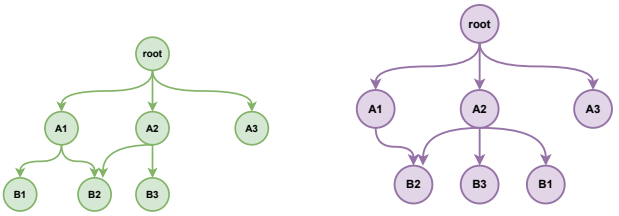
A. Comprehensiveness and Extensibility

Our six inconsistency types capture fundamental differences between dependency trees, with complex inconsistencies decomposable into these atomic types. For example, in Figure 7a, the correct tree shows `B1` as a dependency of `A1`, but in the SBOM tree (Figure 7b), it is incorrectly listed under `A2`. This inconsistency can be broken into *Incorrect Transitive Dependency* ($\mathcal{N}2$) in Figure 7c and *Missing Transitive Dependency* ($\mathcal{M}2$) in Figure 7d. The inconsistency model is extensible. Deeper-layer inconsistencies can be addressed by making types like $\mathcal{M}2$ and $\mathcal{N}2$ layer-sensitive, e.g., $\mathcal{M}2^i$ for Layer i, enabling coverage at any tree depth.

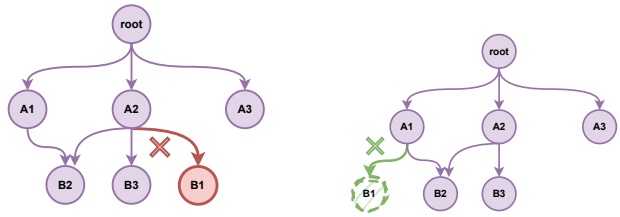


(a) Overhead trend for DTE (b) Overhead trend for SCC

Fig. 6: Performance overhead of *JBomAudit*



(a) Correct Dependency tree in JAR ($\mathcal{T}_c$) (b) Wrong Dependency tree in SBOM ($\mathcal{T}_s$)



(c) Incorrect Transitive Dependency ($\mathcal{N}_2$) (d) Missing Transitive Dependency ($\mathcal{M}_2$)

Fig. 7: An example of complex inconsistency

B. Scalability of JBomAudit

We evaluated *JBomAudit*'s scalability for complex Java projects through theoretical analysis and empirical testing. *JBomAudit* includes two components: Dependency Tree Extraction (DTE) and SBOM Compliance Check (SCC). DTE uses *JarPkgTags*to generate direct dependencies and construct the dependency tree $\mathcal{T}_c$ via Breadth-First Search (BFS) with time complexity $O(V_1)$, where V_1 is the number of nodes in $\mathcal{T}_c$. SCC performs BFS Layer Comparison between $\mathcal{T}_c$ and $\mathcal{T}_s$, with complexity $O(V_1 + V_2)$, where V_2 is the number of nodes in $\mathcal{T}_s$. Both components scale linearly with the number of nodes. Empirical testing on 300 real-world Java projects showed a linear increase in execution time with node count, as illustrated in Figure 6. For 47,042 SBOMs, the median node size of $\mathcal{T}_s$ is 30, taking 2.67 seconds to analyze. First and third quartile node sizes are 17 and 82, with analysis times of 1.56 seconds and 7.75 seconds, respectively, confirming manageable overhead for most projects.

APPENDIX A
ARTIFACT APPENDIX

Executive Order 14028 requires software developers to provide a Software Bill of Materials (SBOM)—a detailed inventory of a software product's dependencies-to support vulnerability management, reduce compliance risks, and enhance supply chain transparency. However, the accuracy and completeness of SBOMs are unclear, potentially undermining security tools, misleading analysts, and increasing risks. To assess the quality of these SBOMs, we developed ***JBomAudit***, a tool for evaluating the completeness and accuracy of Java SBOMs. It takes an SBOM and its corresponding JAR file, extracts the dependency tree from the binary, and compares it to the SBOM. Discrepancies fall into two categories: (1) *Missing dependencies*, further divided into $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$; and (2) *Incorrect dependencies*, categorized as $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$. The artifact includes ***JBomAudit***, environment setup instructions, sample SBOMs, JAR files for testing, and measurement results for validating the main claims of the paper. The workflow involves running the tool, detecting inconsistencies, and verifying the key measurement claims outlined in the paper.

A. Description & Requirements

This section provides all the necessary information to recreate the experimental setup for running ***JBomAudit***.

1) Accessing the Artifact: You can download the artifact, which includes the code, documentation, sample projects for testing, and large-scale measurement results, via this https://zenodo.org/records/14278840.

2) Hardware Requirements: The artifact can be tested on a Linux server or a MacBook. Please note that reproducing the results requires significant memory due to the size of the files involved. The compressed SBOM dataset (*sbom_-json_files.pkl*) is 6.94 GB, and intermediate results, such as *total_miss_deps.json*, are 18.22 GB. We recommend a minimum of 32 GB of RAM (64 GB preferred) to process these files efficiently, along with 30 GB of available disk space for temporary files. For systems with less memory, splitting large files into smaller chunks may help, but this could limit the ability to fully reproduce the results. The following experiments are running on MacBook Apple M3 Max 64 GB.

3) Software Requirements: Install the Python dependencies listed in the provided `functionality/code/jbomaudit/jarpkginfo/requirements.txt` file. We have tested compatibility with Python versions 3.12.3 and 3.10.14.

4) Benchmarks: None.

B. Artifact Installation & Configuration

Step ❶: Create a Python Virtual Environment:

```
# Install virtualenv (if needed)
pip3 install virtualenv
# Create & Activate environment
virtualenv venv
source venv/bin/activate
```

Step ❷: Install JarPkgTags & Dependencies

```
# Navigate to target directory
cd ./AE/functionality/code/jbomaudit/jarpkginfo
```

```
# Install jarpkginfo
pip install .
```

```
# Test jarpkginfo
jarpkgtags tests/my-app-1.0-SNAPSHOT.jar
```

C. Experiment Workflow

In this section, we describe how to test the functionality of ***JBomAudit***. The core functionality of ***JBomAudit*** is to compare an SBOM with its associated JAR binary to detect any inconsistencies between the actual dependencies used in the binary and those listed in the SBOM. These discrepancies are categorized as $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$.

We provide five projects under the directory `./AE/functionality/metaDB/maven_asset_sbom/`, each containing an SBOM file and a corresponding JAR file. Reviewers can specify the paths to the SBOM and JAR files to test the functionality on each project.

For example, the following is the command to run the tool for the *org.opendaylight.aaa* project. The detected inconsistencies are printed as a table in the command line and are also saved as a JSON file in `../../results/audit_results/org.opendaylight.aaa/aaa-cli-jar/0.15.2/compliance_result.json`.The expected runtime is 5s. Reviewers can substitute the paths to test other projects accordingly.

```
#navigate to target directory
cd ./AE/functionality/code/jbomaudit
```

```
python3 main.py --sbom_path \
../../metaDB/maven_asset_sbom/org.opendaylight.aaa
    /aaa-cli-jar/0.15.2/aaa-cli-jar-0.15.2-
    cyclonedx.json \
--jar_path \
../../metaDB/maven_asset_sbom/org.opendaylight.aaa
    /aaa-cli-jar/0.15.2/aaa-cli-jar-0.15.2.jar
```

D. Major Claims

Our study conducts a large-scale SBOM assessment by running ***JBomAudit***. Specifically, we perform three measurement studies to address three key research questions: (❶) the current landscape and characteristics of SBOMs (Section V); (❷) the inconsistency analysis of SBOM dependencies (Section VI); and (❸) the security implications of these inconsistencies (Section VII).

- (C1): We identified 47,042 SBOMs in Maven, revealing an upward trend in releases and the evolving SBOM landscape. Most SBOMs comply with NTIA's minimum requirements. Dependency graphs vary significantly in size, and 222 distinct licenses are used, highlighting the diversity of SBOMs. This is proven by **Experiment** (❶), which reproduces Figure 2, Table VII (see Appendix), and other statistics in Section V of the paper.
- (C2): Our study found a substantial number of SBOMs misrepresent dependencies—either by omitting dependencies used in the code or by incorrectly listing unused

dependencies. This is proven by **Experiment** (❷), which reproduces the results in Table VI, as well as the top 15 missing and incorrect dependencies in Figures 3 and 4, respectively, in Section VI.

- (C3): Our study demonstrate that non-compliant SBOMs pose significant security risks, either by overlooking vulnerabilities associated with missing dependencies (14.75%) or triggering false alerts due to vulnerabilities in incorrect dependencies (2.19%). Additionally, 66.34% of these vulnerabilities are severe. This is proven by **Experiment** (❸), which reproduces Table VIII in Appendix and Figure 5 in Section VII, showcasing security implications from SBOM-level, Dependency-level, and Vulnerability-level perspectives.

E. Evaluation

This section outlines the operational steps and experiments used to validate the claims (C1, C2, C3) through experiments (❶, ❷, ❸) to demonstrate the reproducibility of our study. We have organized and preserved the intermediate results in the corresponding data sections under the reproducibility folder for easy and quick validation

Experiment(❶): SBOM Ecosystem Overview: [0 human-minutes + 3-5 compute-min]: These experiments demonstrates(1) the scope and trend of SBOM releases by plotting the SBOM release timeline, as described in Section V.A of the paper; (2) evaluation of to what extent the SBOMs meet the NTIA minimum requirements and analysis of SBOM characteristics, as described in Section V.B of the paper.

• *[Preparation]* Navigate to target directory:

```
cd ./AE/reproducibility/Section_V/
```

• *[Execution]* [Approximate Running Time: 1s]

```
python3 get_sbom_timeline.py
```

• *[Results]* The script will generate a figure at `./Section_V/figure/timeline_per_sbom.png`, illustrating the SBOM release timeline and the number of SBOMs added monthly. This corresponds to Figure 2 in Section V of the paper.

• *[Execution]* [Approximate Running Time: 25s]

```
python3 check_min_requirements.py --task NTIA
```

• *[Results]* This will generate Table VII (see Appendix) for Section V.B from the paper, showing the number and percentage of SBOMs that contain the required fields (e.g., Supplier Name, Component Name, etc).

• *[Execution]* [Approximate Running Time: 45s]

```
python3 check_min_requirements.py --task Dep
```

• *[Results]* This will show statistics about the Dependency Graph from collected SBOMs, as detailed in Section V.B: Dependency Graph.

• *[Execution]* [Approximate Running Time: 25s]

```
python3 check_min_requirements.py --task License
```

• *[Results]* This will show statistics about the License Usage from collected SBOMs, as detailed in Section V.B: License Usage.

Experiment(❷): Inconsistency Analysis: [0 human-minutes + 1 compute-min]: This experiment evaluates SBOM dependency inconsistencies across 25,882 JARs and their corresponding SBOMs. It reproduces the results for Table VI, as well as Figures 3 and 4 from Section VI of the paper.

• *[Preparation]* Navigate to target directory:

```
cd ./AE/reproducibility/Section_VI/
```

• *[Execution]* [Approximate Running Time: 1s]

```
python3 stats_noncompliance.py --task landscape
```

• *[Results]* This command will reproduce the results shown in Table VI in Section VI, detailing the overall dependency inconsistencies detected by **JBomAudit**.

• *[Execution]* [Approximate Running Time: 1s]

```
python3 stats_noncompliance.py --task missing
```

• *[Results]* This will reproduce the results for the top 15 missing dependencies, as shown in Figure 3 of Section VI.

• *[Execution]* [Approximate Running Time: 1s]

```
python3 stats_noncompliance.py --task incorrect
```

• *[Results]* This will reproduce the results for the top 15 incorrect dependencies, as shown in Figure 4 of Section VI.

Experiment(❸): SBOM Security Implication : [0 human-minutes + 1 compute-min]: This experiment assesses the security risks associated with non-compliant SBOMs from three perspectives: SBOM-level, Dependency-level, and Vulnerability-level, as described in Section VII.C of the paper.

• *[Preparation]* Navigate to target directory:

```
cd ./AE/reproducibility/Section_VII/
```

• *[Execution]* [Approximate Running Time: 5s]

```
python3 analyze_security_risk.py --task SBOM_level
```

• *[Results]* This script will reproduce the SBOM-level analysis results from Section VII.C, showing the percentage of SBOMs that involve missing or incorrect dependencies containing vulnerabilities.

• *[Execution]* [Approximate Running Time: 4s]

```
python3 analyze_security_risk.py --task Deps_level
```

• *[Results]* This will reproduce the results for Table VIII (see Appendix) for Section VII.C, breaking down the security risks by type of non-compliance at the dependency level.

• *[Execution]* [Approximate Running Time: 4s]

```
python3 analyze_security_risk.py --task Vul_level
```

• *[Results]* This script will reproduce the raw data for Figure 5 in Section VII.C, showing the CVE IDs and severity scores of vulnerabilities associated with missing and incorrect dependencies, and their impact on SBOMs. Please note that the figure itself is not generated by the Python code, hence the raw data will be provided for validation.