On the Correctness of Metadata-based SBOM Generation: A Differential Analysis Approach

Sheng Yu*†, Wei Song†, Xunchao Hu†, Heng Yin*†
*University of California, Riverside
†Deepbits Technology Inc.
syu061@ucr.edu, wei@deepbits.com, xchu@deepbits.com, heng@cs.ucr.edu

***Abstract*—Amidst rising concerns of software supply chain attacks, the Software Bill of Materials (SBOM) has emerged as a pivotal tool, offering a detailed listing of software components to manage vulnerabilities, dependencies, and licensing. While many SBOM generation tools are extensively used in both commercial and open-source realms, the correctness of these tools remains largely unscrutinized. To date, there has not been a systematic study addressing the correctness of contemporary SBOM generation solutions. In this paper, we conduct a large-scale differential analysis of the correctness of four popular SBOM generators. Surprisingly, our evaluation reveals all four SBOM generators exhibit inconsistent SBOMs and dependency omissions, leading to incomplete and potentially inaccurate SBOMs. Moreover, we construct a parser confusion attack against these tools, introducing a new attack vector to conceal malicious, vulnerable, or illegal packages within the software supply chain. Drawing from our analysis, we propose best practices for SBOM generation and introduce a benchmark to steer the development of more robust SBOM generators.**

I. Introduction

Software Supply Chain Attacks (e.g., SolarWinds [18], PyTorch dependency confusion attack [9]) have increased by 742% between 2019 and 2022 [16]. In 2022 alone, 185,572 software packages were affected by these attacks [1]. The lack of visibility and transparency in the software supply chain makes defending against such attacks challenging. Recently, the Software Bill of Materials (SBOM) [10], a list of "ingredients" used to build software, has demonstrated its efficacy in protecting the software supply chain by enhancing visibility from software development to consumption. Driven by regulations, such as Biden's executive order [3] and the National Cybersecurity Implementation Plan [7], the industry is adopting SBOM-based solutions to safeguard the software supply chain.

An essential step in adopting SBOM is to generate accurate SBOMs. While SBOMs have the potential to enhance vulnerability detection and facilitate license compliance, these benefits can only be realized if the SBOMs themselves are precise and correct. Discrepancies or omissions in the SBOM can lead to false assurances of security or compliance, exposing systems to potential risks. Many SBOM generation tools [4], [6], [12], [13] are extensively used in both commercial and open-source realms. However, the correctness of these tools remains largely unscrutinized. To date, there has not been a systematic study addressing the correctness of contemporary SBOM generation solutions.

Given the diversity of programming languages, build tools, and development practices, constructing a ground truth for SBOM generation evaluation is inherently challenging. In this paper, we adopt a differential analysis approach: we analyze the discrepancies in SBOMs produced by different tools for the same software to assess both their correctness and weaknesses in SBOM generation. More specifically, we 1) select four popular SBOM generators: Trivy [13], Syft [12], Microsoft SBOM Tool [6], and GitHub Dependency Graph [4]; 2) collect 7,876 open-source projects written in Python, Ruby, PHP, Java, Swift, C#, Rust, Golang and JavaScript; 3) evaluate the correctness of the SBOMs by conducting a differential analysis on the outputs from these four tools.

Surprisingly, our evaluation reveals all four SBOM generators exhibit inconsistent SBOMs and dependency omissions, leading to incomplete and potentially inaccurate SBOMs. Moreover, we construct a parser confusion attack against these tools, introducing a new attack vector to conceal malicious, vulnerable, or illegal packages within the software supply chain. To assist in creating more effective SBOM generators, we have developed best practices for SBOM generation and a benchmark to facilitate their development based on our evaluation findings.

In summary, we make the following contributions in this paper:

- We are the first to conduct a large-scale differential analysis to examine the correctness of SBOM generation solutions.
- Our evaluation reveals significant deficiencies in current SBOM generators. We also conduct a comprehensive case study to uncover how each SBOM tool detects dependencies during the generation process.
- We construct a parser confusion attack against SBOM generators, introducing a new attack vector to inject malicious, vulnerable, or illegal software packages into the software supply chain.
- We develop best practices for developing SBOM generators and a benchmark to facilitate their development.

Acknowledgments: We thank the anonymous reviewers and our shepherd Yuchen (Dennis) Zhang for their valuable feedback. This work is supported, in part, by the Department of Homeland Security under OTA#70RSAT23T00000013. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00018

II. Background

A. Software Bill of Materials

An SBOM [10] is a formal, machine-readable inventory of software components and dependencies that includes information about those components and their hierarchical relationships. It can be shared and exchanged automatically among stakeholders (e.g., software vendors and consumers) to enhance software development, software supply chain management, vulnerability management, asset management, and procurement. This results in reduced costs, security risks, license risks, and compliance risks.

SBOM Types: Based on the stages of the software lifecycle at which SBOMs are generated, they can be categorized into six types [14]: Design, Source, Build, Analyzed, Deployed, and Runtime. Depending on what information is available in each stage, these types of SBOMs focus on different aspects. In this paper, we evaluate Source SBOM, a type of SBOM derived from the development environment. It mainly contains dependencies used for development and compilation, and is widely supported by SBOM tools. Also, our survey suggests that, owing to its simplicity and precision, metadata parsing is the industry's leading SBOM generation technique. Thus, this paper focuses on the Source SBOM generated using the metadata-based approach.

SBOM Applications: The increasing complexity and interdependence in software development have amplified the importance of SBOMs. These provide clarity by clearly listing software components, facilitating swift vulnerability tracking and identification for developers and security professionals. Their compatibility with Vulnerability Exploitability eXchange (VEX) [15], a structured database detailing product vulnerabilities, is noteworthy. Additionally, the comprehensive dependency information aids in license assessment, ensuring compliance and mitigating legal exposures. SBOMs enable quality assessment of closed-source software through component reputation checks, and their transparency fortifies the software supply chain by thwarting the introduction of potential backdoors and vulnerabilities via third-party components.

B. Metadata

At the heart of Source SBOM generation lies the metadata - an important element in modern software development. These files encapsulate parameters, settings, dependencies, and version constraints, all of which are indispensable for reproducibility and consistent and reliable deployment, and offer support for package management, version control, and even automated build processes. Nowadays, almost every programming language comes with at least one package manager, and each package manager defines its own metadata.

At high level, there are two kinds of metadata. One is "raw" metadata where only direct dependencies are specified and their versions are often given as a range or a constraint instead of a specific (pinned) one. Raw metadata, such as `requirements.txt` for Python and `package.json` for Node.js, are mainly for dependency declaration while ensuring a degree of flexibility and future compatibility. The other type is lockfile such as `package-lock.json` for Node.js. Lockfiles focus on providing a precise and deterministic snapshot of the exact dependency tree including transitive dependencies. Locking prevents unexpected updates or changes in the dependencies when installing the project across different environments, ensuring reproducibility and avoiding compatibility issues. Despite that lockfiles contain the richest information for SBOM generation, they are not always available. Library developers are not encouraged to share lockfiles which could otherwise lead to version conflicts. Some package managers lack a native locking mechanism. Without lockfiles, the missing transitive dependencies and pinned versions pose a great challenge to SBOM tools to generate accurate and complete SBOM files.

III. Methodology

Despite the growing significance and adoption of SBOMs, a notable gap exists in systematically assessing the quality of the SBOM files generated. The reliability of security-centric applications, including vulnerability detection and license compliance, highly depends on the correctness of SBOM data, which raises concerns regarding the trustworthiness of such information.

This work aims to investigate the correctness and completeness of the dependency information present in generated SBOMs. The objective is to not only measure the correctness but also to unravel the underlying factors contributing to high-quality SBOMs. Due to the lack of ground truth, we adopt a differential analysis approach to obtain insights into the performance of SBOM generators.

A. SBOM Generators

In this work, we evaluate four SBOM tools: `Trivy` 0.43.0, `Syft` 0.84.1, `Microsoft SBOM Tool` (sbom-tool) 1.1.6, and `GitHub Dependency Graph` (GitHub DG). Notably, the first three are popular open-source projects and offer cross-platform support for Linux, Windows, and Mac operating systems. Conversely, the GitHub Dependency Graph is intricately integrated with GitHub repositories. We choose Trivy and Syft because they are the de facto SBOM generators used by industries and open-source communities. We pick the Microsoft SBOM Tool because it is developed by the esteemed Microsoft. Similarly, the GitHub Dependency Graph is chosen because it is provided by the most widely used Git platform. All the evaluated SBOM tools implement metadata-based approaches, meaning they read metadata files and extract dependency information declared in the metadata files.

B. Setup

The evaluation was conducted by downloading popular GitHub repositories associated with each programming language onto the local file system and subsequently scanning the repository directories using the SBOM tools. Each tool will generate an SBOM report in either CycloneDX [8] or

SPDX [5] format depending on which format is supported by the tools. Dependencies in these reports are then extracted and compared against each other.

Dataset: GitHub repositories were sourced from the well-regarded `awesome-LANGUAGE` repositories, which are uniquely tailored to the respective programming languages. Our dataset contains 535 Python, 819 Ruby, 384 PHP, 398 Java, 1,019 Swift, 700 C#, 994 Rust, 2,367 Golang, and 660 JavaScript repositories. We do not evaluate C/C++ projects due to the absence of an "official" build toolset and extremely limited support provided by the SBOM tools. C/C++ projects can be configured and built via various tools such as Bazel, Makefile, CMake, Visual Studio project files, and more. Consequently, Trivy and Syft only analyze `conan.lock`, while GitHub Dependency Graph exclusively focuses on `*.vcxproj` files.

Metrics: For our large-scale evaluation, given the absence of ground truth, we adopt a differential analysis approach. First, we compare the number of dependencies reported by each SBOM tool. We then use Jaccard similarity to measure the reported dependency names. This tells us the degree of overlap and commonality among the dependencies reported by different tools. In addition, we identify duplicate packages reported by the SBOM tools. While these metrics may not provide a direct ranking, they do shed light on the performance of these tools.

IV. Large-scale SBOM Comparison

After analyzing 7,876 high-quality repositories, we made the following major findings. The reasons behind such discrepancies will be discussed in Section V.

A. Discrepancies in Package Counts within SBOM Reports Generated by Different Tools

The SBOM tools exhibited notable differences in the number of packages they identified. Figure 1 clearly depicts this variation. The x-axis is the repository ID sorted by the number of dependencies detected by the GitHub Dependency Graph. For Python, PHP, Ruby, and Rust programming languages, GitHub Dependency Graph discovers the most packages for these languages. For .Net repositories, Microsoft SBOM Tool excelled in identifying the most packages, which is unsurprising as it is tailored to Microsoft's own projects. For the Go and Swift languages, Trivy and Microsoft SBOM Tool proved to be the frontrunners, consistently identifying the most packages in the majority of cases. Syft excels in detecting the highest number of packages when it comes to JavaScript repositories. The disparities presented in this figure underscore that different tools possess varying capabilities and strategies in identifying dependency packages across different programming languages. It is important to note, however, that identifying more packages does not mean better because false positives may also be included.

B. Low Package Jaccard Similarities

To measure whether the SBOM tools detect similar dependencies for each repository, we compute a Jaccard similarity for each SBOM tool pair for each repository as Equation 1 shows. A and B are two sets of dependencies generated by two different SBOM tools. Each set contains dependency $(name, version)$ pairs.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

Our evaluation result is illustrated in Figure 2. The majority of these pairs show significant dissimilarity, with only a very small portion being similar. As shown in (a), the GitHub Dependency Graph and Syft have the most similarities among them, although the majority of SBOM reports still exhibit substantial differences.

C. Duplicate Packages in SBOMs

During our analysis of the generated SBOMs, we identified instances of duplicate packages: the same package appearing in different entries with varying or the same version requirements. To ensure accurate calculations, we excluded repositories in which tools could not find any packages.

In Table I, we have presented the rate of duplicate packages for various SBOM tools. This problem was found to be widespread across all four tools, suggesting a common occurrence. However, it is important to note that having duplicate packages is expected in some cases. For example, a repository may contain multiple independent projects and they happen to have a common subset of dependencies.

TABLE I: Rate of Duplicate Packages in SBOMs

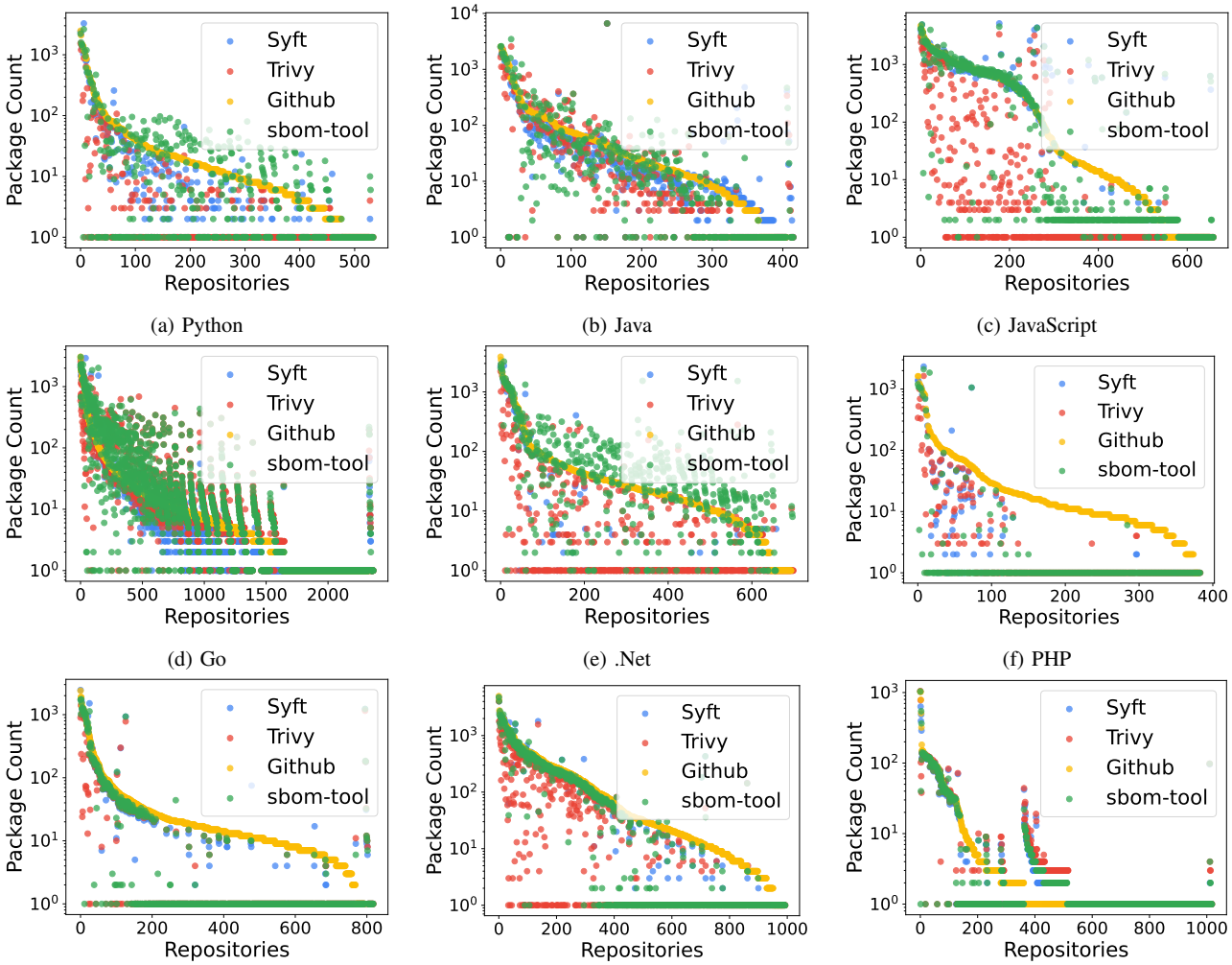
	Syft	**Trivy**	**GitHub DG**	**sbom-tool**
Python	14.05%	12.56%	13.54%	13.71%
Java	12.76%	15.01%	19.93%	18.89%
JavaScript	17.46%	17.34%	18.89%	19.42%
Go	9.97%	6.69%	11.03%	6.58%
.NET	17.38%	12.43%	18.01%	20.94%
PHP	13.76%	11.77%	14.53%	23.76%
Ruby	13.56%	9.1%	15.84%	12.39%
Rust	13.19%	11.37%	19.18%	13.83%
Swift	1.37%	2.28%	6.98%	3.39%

V. SBOM Generation Analysis

To uncover the root causes behind the large disparities in SBOM outputs, we conducted an in-depth analysis of the source code of the SBOM tools. Our examination revealed several critical issues in SBOM generation, which are summarized below.

A. Limited Support for Metadata

All the evaluated tools employ a metadata-based approach where they analyze metadata to identify the components used in the project. The supported metadata file types for each tool are detailed in Table II. It is important to note that the table indicates the tools' actual capability to extract dependencies from metadata, which may differ from their claims.



(a) Python (b) Java (c) JavaScript

(d) Go (e) .Net (f) PHP

(g) Ruby (h) Rust (i) Swift

Fig. 1: Comparison of Package Counts Across Languages Using Various SBOM Generators

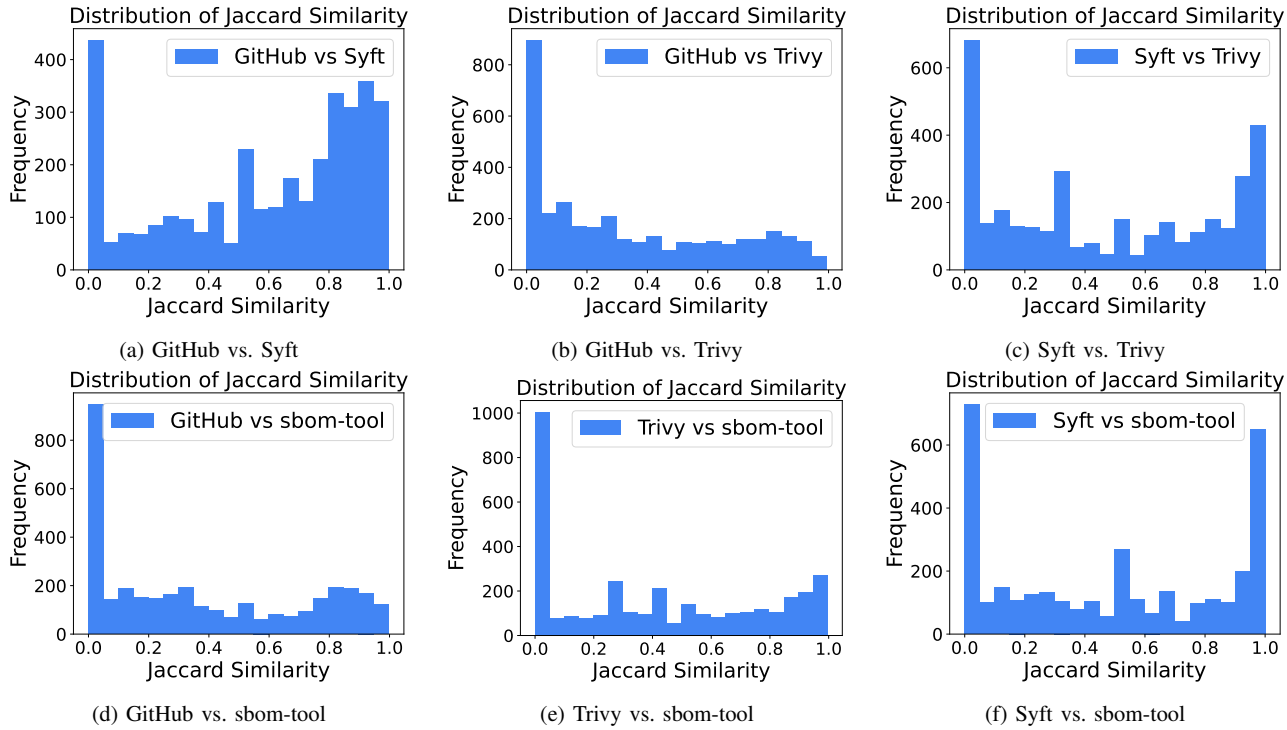
The table illustrates that each tool supports only a subset of commonly used metadata files. Overall, the SBOM tools have good support for lockfiles in which transitive dependencies and pinned versions are available, but they struggle with raw metadata. The GitHub Dependency Graph has the best support for raw metadata such as `Gemfile` and `Cargo.toml`, while other tools show limited or no support for raw metadata. Despite claims by Trivy and Syft to support `package.json`, they do not extract dependencies from the JSON file. In our evaluation, we found that 93% of Python repositories, 47% of JavaScript repositories, and 56% of Rust repositories contain raw metadata only.

B. Incomplete Metadata Parsing

Our evaluation shows that all the evaluated SBOM tools implement custom parsers for metadata. However, certain metadata, like `requirements.txt` defined in PEP 508, poses challenges due to its complex syntax. The self-implemented parsers only support common syntaxes, leading to false negatives. For instance, the lack of support for the backslash "\" as a line continuation in all the SBOM tools causes parsing errors, resulting in incorrect versions or missed dependencies. About 1.8% of Python repositories are affected by this.

C. Transitive Dependency

The offline nature of SBOM tools (except Microsoft SBOM Tool) implies a lack of attempts to resolve transitive dependencies. In the case where lockfiles are not present, the absence of transitive dependencies will adversely affect SBOM applications. Microsoft SBOM Tool attempts to resolve transitive dependencies by querying package managers for each detected dependency, but this functionality is not well-implemented and often fails to retrieve dependency information from package managers. About 74% of Python dependencies are transitive dependencies.



(a) GitHub vs. Syft (b) GitHub vs. Trivy (c) Syft vs. Trivy

(d) GitHub vs. sbom-tool (e) Trivy vs. sbom-tool (f) Syft vs. sbom-tool

Fig. 2: Distribution of Jaccard Similarity among Various Tools

TABLE II: Supported File Types

		Trivy	Syft	sbom-tool	GitHub DG
Go	go.mod	✓	✓	✓	✓
	Go executable	✓	✓	✗	✗
Java	pom.xml	✓	✓	✓	✓
	gradle.lockfile	✓	✓	✓	✓
	MANIFEST.MF	✓	✓	✗	✗
	pom.properties	✓	✓	✗	✗
JS	package.json	✗	✗	✗	✓
	package-lock.json	✓	✓	✓	✓
	yarn.lock	✗	✓	✓	✓
	pnpm-lock.yaml	✗	✓	✓	✗
PHP	composer.json	✗	✗	✗	✓
	composer.lock	✓	✓	✗	✓
Python	requirements.txt	✓	✓	✓	✓
	poetry.lock	✓	✓	✓	✓
	pipfile.lock	✓	✓	✓	✓
	setup.py	✗	✗	✗	✓
Ruby	Gemfile	✗	✗	✗	✓
	Gemfile.lock	✓	✓	✓	✓
	.gemspec	✓	✓	✓	✓
Rust	Cargo.toml	✗	✗	✗	✓
	Cargo.lock	✓	✓	✓	✓
	Rust executable	✓	✓	✗	✗

D. Limited Support for Version Constraints

Raw metadata often contains version ranges or constraints instead of pinned versions; for example, developers use `>=1.2.3 <2.0.0` to get the latest version while ensuring backward compatibility. Trivy and Syft handle version constraints by silently discarding dependencies without pinned versions, resulting in false negatives. The GitHub Dependency Graph reports version ranges as they appear in the metadata, introducing additional parsing challenges for SBOM management. In our evaluation, only 46% of dependencies declared in `requirements.txt` have pinned versions, indicating that Trivy and Syft may miss more than half of the dependencies even when transitive dependencies are not considered. Microsoft SBOM Tool addresses this by pinning a version after querying the corresponding package manager for the latest version within the specified range.

E. Inconsistent Package Naming Convention

When dealing with packages having compound names, SBOM tools name them differently. For Java, a package is located using the group ID and artifact ID. Syft uses the artifact ID as the package name, Microsoft SBOM Tool concatenates the group and artifact ID with a dot "." as the package name, while Trivy and the GitHub Dependency Graph use a colon sign ":" for this purpose. Similarly, Swift package manager CocoaPods supports subpecs when declaring a dependency. Subspecs are a way of chopping up the functionality of a library, allowing people to install a subset of the library. Syft and Trivy report the subspecs, while Microsoft SBOM Tool reports their main dependency names. Additionally, Golang uses a leading letter "v" when specifying versions (e.g., v1.0.0). Syft and Microsoft SBOM Tool adhere to this convention, while Trivy and the GitHub Dependency Graph omit this leading

letter. Such inconsistencies can potentially compromise the accuracy of vulnerability detection.

F. Different Dependency Definition

SBOM tools employ different strategies regarding whether to include development dependencies (e.g., test suites, linters, etc.) in SBOM files. Trivy focuses solely on production dependencies and ignores development dependencies, whereas Syft and GitHub Dependency Graph include both types. Our evaluation reveals that in JavaScript, 76% of dependencies declared in `package.json` are development dependencies. It is crucial to note that there is no definitive answer regarding which approach is better. Including development dependencies in the SBOM report offers several advantages, such as more comprehensive vulnerability assessments and license violation checks, but it may also introduce false alarms as the code of development dependencies rarely goes into the final product. The root problem lies in the absence of an existing field in SBOM formats representing the dependency scope. While most metadata have distinct fields for this purpose, such as the `scope` field in `pom.xml` and the `devDependencies` in `package.json`, the current SBOM formats lack this support and may cause confusion in downstream applications.

G. Multiple Projects and Metadata

Our evaluation indicates that, on average, over 10% of the detected dependencies appear more than once in a repository, causing duplicate entries in SBOM files. This is primarily due to multiple metadata files present in a repository, either because of having multiple subprojects or submodules or having both raw metadata and lockfiles present. The SBOM tools analyze metadata individually without merging dependencies in the same project. Duplicate entries in SBOMs can lead to confusion and potentially inflate the apparent package count. Our evaluation shows that there are 5.7 metadata files in a Python repository and 12.8 metadata files in a JavaScript repository on average.

H. Accuracy on Ground Truth

Our large-scale evaluation employed a differential analysis due to the lack of ground truth. In this section, we quantify the accuracy of each SBOM tool on `requirements.txt` using our manually crafted ground truth. The ground truth is obtained by dry-running `pip install` (Python 3.11, pip 23.1.2), and we consider a correct dependency $(name, version)$ pair as a correct match. Dry run simulates the installation process and the dependencies reported by `pip install` are those that will be installed in our environment. This evaluation aims to highlight the differences between the reported libraries and the ones actually installed.

The evaluation result is presented in Table III. Most SBOM tools fail to detect over 90% of the dependencies in `requirements.txt` due to incomplete syntax support and the lack of transitive dependency resolution. The Microsoft SBOM Tool excels in this test because it attempts to resolve transitive dependencies, but it ignores the `extras` field, and OS and Python requirements. The low recall suggests that relying solely on these SBOM tools in practice may have serious negative impacts on downstream applications, such as vulnerability detection and license violation checks.

TABLE III: SBOM Accuracy on `requirements.txt`

	Trivy	**Syft**	**sbom-tool**	**GitHub DG**
Precision	0.25	0.25	0.74	0.13
Recall	0.10	0.10	0.73	0.08

TABLE IV: requirements.txt Attack Samples

	Trivy	**Syft**	**sbom-tool**	**GitHub DG**
requests [security]>=2.8.1	-	-	-	-
numpy \ ==\ 1.19.2	-	-	numpy 1.25.2	-
-r SOME_REQS.txt	-	-	-	-
./path/to/local_pkg.whl	-	-	-	-
https://remote_pkg.whl	-	-	-	-
urlib3 @ git_link@hash	-	-	-	-

VI. Parser Confusion Attack

Motivated by the findings in Section V-H, we present a parser confusion attack [20] to illustrate how adversaries can obscure malicious dependencies. A parser confusion attack exploits inconsistencies among different parsers processing the same input, enabling malicious actors to craft input that is benign for one parser but harmful for another. Our case study shows that SBOM tools, employing custom metadata parsers, introduce a new attack vector for constructing parser confusion attacks within the SBOM ecosystem. In this study, we use Python's `requirements.txt` as an illustrative example.

Constructing the attack: Given that `requirements.txt` lacks a locking mechanism and exhibits a rich syntax, it becomes a suitable candidate for this type of attack. For instance, none of the SBOM tools support the backslash as a line continuation; Trivy and Syft rely on the double-equal sign to separate package names and versions; installations from wheel packages are not universally supported; and many more. Table IV provides some input patterns that can be used to bypass detections based on our manual analysis and benchmark (discussed in Section VII). It shows how attackers can leverage different syntax elements to either conceal specific dependencies or confuse SBOM tools, leading to inaccurate results. In the table, a dash ("-") signifies that the corresponding SBOM tool cannot detect anything from the given dependency declaration.

Achieving Damage: When the SBOM tools encounter unsupported syntax, the default behavior is to silently ignore the associated dependency. Adversaries can exploit this and inject malicious or vulnerable dependencies in metadata using unsupported syntax, effectively evading the tools' detection entirely. In our dataset, the two most common patterns are installing from other requirement files (`-r`) and installing from version control systems, each appearing in over 50 `requirements.txt` files.

VII. Best Practice and Benchmark

Drawing from our evaluation, we present what we believe are the most optimal solutions to address identified issues and minimize the attack surface. We propose the following best practices for metadata-based approaches:

Package Manager Dry Run for Lockfile Generation: The root cause of the large discrepancies lies in the limitations of self-implemented parsers, particularly in their support for metadata and metadata syntax. Instead of relying on these parsers, we recommend employing a package manager dry run to generate lockfiles. This simulates the dependency installation process, providing both transitive dependencies and accurate version information for each package. Adopting this approach ensures the creation of a precise and reliable SBOM file, thereby enhancing resilience against confusion attacks.

PURL and CPE Support: Each dependency should include a PURL (Package URL) entry and a CPE (Common Product Enumerator) entry for consistent package naming convention, maximum compatibility with vulnerability databases, and facilitate software identification.

Our evaluation benchmark is available on GitHub at https://github.com/DeepBitsTechnology/sbom-benchmark. This benchmark includes manually crafted metadata files and ground truth datasets for common languages. These metadata files try to cover all supported syntaxes for each language, and can be used to evaluate of the SBOM tools' capability to handle corner cases. This initiative aims to guide the development of SBOM tools, emphasizing completeness and accuracy. We are working on adding support for more programming languages.

VIII. Discussion

This study aims to assess the quality of SBOMs produced by widely used SBOM tools. Our analysis exposes deficiencies in the SBOM generation process employed by these tools. Trivy, Syft, and GitHub Dependency Graph do not identify transitive dependencies or determine an appropriate version when no pinned version is provided. In contrast, the Microsoft SBOM Tool reaches out to package managers to validate package names and ascertain a suitable version.

While conducting our evaluation, we encountered a significant challenge stemming from the absence of a well-defined benchmark for accurately assessing the quality of the generated SBOMs. Currently, the industry lacks a standardized dataset and uniform statistical methods for conducting evaluations in this area. In response to this issue, we created our own dataset.

Our experiment focuses on metadata-based Source SBOM generation on file system. It is important to note that certain SBOM tools, such as Trivy, may exhibit different behaviors depending on the specific targets of their scans. For example, scanning metadata files is enabled for both file system and git repository scans, while the activation of wheel packages is restricted to Docker image and Rootfs scans.

It is worth mentioning that our evaluation was specifically limited to a subset of SBOM tools, namely Trivy, Syft, Microsoft SBOM Tool, and GitHub Dependency Graph. Despite our careful selection of these prominent tools, the dynamic and ever-evolving landscape of SBOM generation solutions implies that our findings may not cover the entirety of available options. There is a possibility that subtle variations presented by other tools might have been inadvertently overlooked.

While metadata-based SBOM generation is relatively simple to implement, this approach has inherent limitations. First, declared dependencies may only be partially built into the final product or not be used at all, potentially leading to false alarms. Transitive dependencies are not well-captured, causing false negatives. Moreover, developers might add code directly to the project for experiments or testing, and metadata-based approaches are unable to detect such cases. We recommend implementing def-use analysis to determine whether each library within the project has been used or not. Additionally, code clone detection [26], [27], [33] can identify libraries introduced via copy & paste. Employing these techniques helps eliminate false positives and false negatives, enhancing the overall correctness of the SBOM.

IX. Related Work

Software Supply Chain Attacks Malicious [30] or vulnerable packages [21] have resulted in increasing [28] software supply chain attacks (SolarWind [18], NotPetya [24], etc.). Various approaches have been proposed [31], [32]. SBOM [10] demonstrates its efficiency in managing risks in the software supply chain and has been advocated by both the industry and goverment stakeholders [3], [7].

SBOM & Vulnerability Exploitability eXchange (VEX) VEX, as defined by NTIA, is a "companion artifact" to a SBOM [15], allowing manufacturers to share product vulnerability exploitability in a standardized, automatable format. Ahmed el al. [17] applied SBOM tools to assess how code debloating reduces vulnerabilities in Docker images. Numerous tools (DependencyTrack [2], DeepSCA [11], Nadgowa [29], Girdha [23], etc.) have been developed to support SBOM generation and consumption. In particular, DeepSCA is a complimentary online service that generates different types of SBOMs and conducts risk analysis for most popular languages and platforms with or without the source code.

Software Composition Analysis (SCA) Apart from metadata-based parsing, SCA is also a promising technique for generating SBOMs. When source code is available, SCA solutions such as CENTRIS [33] and Tamer [26] can be combined with program analysis to identify components that are actively invoked in the software, yielding more accurate SBOMs. When the source code is not available, binary-focused SCA tools like BAT [25], OSSPolice [22], B2SFinder [34], and LibScout [19] utilize string literals and other language-specific features to discern components in the examined binaries. Though their accuracy might not be optimal, they still enhance transparency to a certain degree.

X. Conclusion and Future Work

In this paper, we conducted the first large-scale differential analysis to examine the correctness of SBOM generation solutions. We generated SBOMs using four popular SBOM generators for 7,876 open-source projects and systematically studied the correctness of these SBOMs. Our evaluation uncovered significant deficiencies in current SBOM generators. Additionally, we identified the design flaws in each SBOM generator, and devised a parser confusion attack against these generators, introducing a new path for injecting malicious, vulnerable, or illegal packages. Finally, based on our findings, we established best practices for creating SBOM generators and introduced a benchmark to aid their development.

In the future, we plan to extend our benchmark to support languages beyond just Python. Additionally, we aim to establish a ranking system to qualitatively measure the quality of SBOM generators in the market, allowing security professionals to select the most suitable tools and SBOM generator vendors to evaluate and improve their offerings.

References

[1] Annual number of software packages impacted by supply chain cyber attacks worldwide from 2019 to 2023 ytd. https://www.statista.com/statistics/1375128/supply-chain-attacks-software-packages-affected-global/.

[2] Depedency track. https://dependencytrack.org/.

[3] Executive order on improving the nation's cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[4] Github dependency graph. https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph.

[5] International open standard (iso/iec 5962:2021) - software package data exchange (spdx). https://spdx.dev/.

[6] Microsoft sbom tool. https://github.com/microsoft/sbom-tool.

[7] National cybersecurity strategy implementation plan. https://www.whitehouse.gov/wp-content/uploads/2023/07/National-Cybersecurity-Strategy-Implementation-Plan-WH.gov_.pdf.

[8] Owasp cyclonedx software bill of materials (sbom) standard. https://cyclonedx.org/.

[9] Pytorch machine learning framework compromised with malicious dependency. https://thehackernews.com/2023/01/pytorch-machine-learning-framework.html.

[10] Software bill of materials. https://www.ntia.gov/page/software-bill-materials.

[11] Software supply chain arsenal. https://tools.deepbits.com/.

[12] Syft. https://github.com/anchore/syft.

[13] Trivy. https://trivy.dev/.

[14] Types of software bill of materials. https://www.cisa.gov/resources-tools/resources/types-software-bill-materials-sbom.

[15] Vulnerability-exploitability exchange (vex)–an overview. https://www.ntia.gov/files/ntia/publications/vex_one-page_summary.pdf.

[16] Why 2023 is the year for software supply chain attacks. https://hadrian.io/blog/why-2023-is-the-year-for-software-supply-chain-attacks.

[17] F. A. Ahmed and D. Fatih. Security analysis of code bloat in machine learning systems. 2022.

[18] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad. Solar winds hack: In-depth analysis and countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2021.

[19] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 356–367, 2016.

[20] C. Carmony, M. Zhang, X. Hu, A. V. Bhaskar, and H. Yin. Extract me if you can: Abusing PDF parsers in malware detectors. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, Feb. 2016.

[21] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018.

[22] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[23] S. Girdhar. Frankfurt university of applied sciences.

[24] A. Greenberg. The untold story of notpetya, the most devastating cyberattack in history. *Wired, August*, 22, 2018.

[25] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011.

[26] T. Hu, Z. Xu, Y. Fang, Y. Wu, B. Yuan, D. Zou, and H. Jin. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 89–100, 2023.

[27] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.

[28] J. Martínez and J. 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.

[29] S. 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.

[30] M. Ohm, H. Plate, A. Sykosch, and M. 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.

[31] M. Ohm and C. Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–11, 2023.

[32] M. Ohm, A. Sykosch, and M. Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*, pages 1–6, 2020.

[33] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 860–872. IEEE, 2021.

[34] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu, et al. B2sfinder: Detecting open-source software reuse in cots software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1038–1049. IEEE, 2019.