SBOM Generation Tools Under Microscope: A Focus on the npm Ecosystem

Md Fazle Rabbi
Idaho State University
Pocatello, ID, USA
mdfazlerabbi@isu.edu

Arifa I. Champa
Idaho State University
Pocatello, ID, USA
arifaislamchampa@isu.edu

Costain Nachuma
Idaho State University
Pocatello, ID, USA
costainnachuma@isu.edu

Minhaz F. Zibran
Idaho State University
Pocatello, ID, USA
minhazzibran@isu.edu

ABSTRACT

Generating accurate Software Bill of Materials (SBOM) is challenging due to the complex dependencies in the diverse components used in software and also the way software is built into executables. A handful of tools claim the capability of automatic SBOM generation from software distributions while little is known about their applicability, strengths, and limitations. Our study makes quantitative and qualitative comparisons of the four such tools (i.e., ORT, cnn, syft, cdxgen) that claim to be capable of generating SBOM from JavaScript projects. For the comparison, we operate these four tools on 50 open-source JavaScript npm projects. We find significant performance variations when evaluating their ability to extract component details, especially in detecting dependencies. The findings of this study are useful in the design and development of SBOM generator tools, in end-users' selections of such tools, and thus in the overall improvement of the security and transparency in software supply chain.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; • **Security and privacy → Software and application security**; • **Computing methodologies → Empirical studies in software engineering**.

KEYWORDS

Software Supply Chain, Software Bill of Materials, SBOM Generator, JavaScript

ACM Reference Format:
Md Fazle Rabbi, Arifa I. Champa, Costain Nachuma, and Minhaz F. Zibran. 2024. SBOM Generation Tools Under Microscope: A Focus on the npm Ecosystem. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3605098.3635927

1 INTRODUCTION

Software development has witnessed tremendous change as a result of the widespread use of open-source software (OSS) and third-party dependencies [26]. Developers frequently use existing code, libraries, and dependencies to speed up application development. However, this reliance creates significant problems for security, licensing, vulnerabilities, and software supply chain (SSC) management. This is where SBOMs come in, serving as a foundational blueprint. They provide a comprehensive list of all the components, libraries, and dependencies present within a software application [25]. From a security perspective, if a component/ingredient used in a software is found insecure, the entire software can be considered insecure necessitating immediate measures to secure the system.

In the current software environment, cyberattacks frequently target vulnerabilities in both primary software components and third-party libraries and dependencies [27]. For instance, the SolarWinds attack [2] demonstrated the risks posed by hidden vulnerabilities deep in the SSC. This highlights the importance of SBOMs since they provide a clear, organized understanding of the components within a software product [8]. SBOM is becoming an essential tool for navigating these challenges.

Furthermore, regulatory bodies have focused on enhancing software transparency. In May 2021, the US government took a significant step to improve accountability and security of SSC [5]. It was mandated that federal agencies and their contractors supply SBOMs for their software. While this initiative is praiseworthy, it raises questions about its adoption by non-mandated organizations within the United States. Furthermore, there are concerns about its implementation by organizations outside of the US who are not obligated to produce or provide SBOMs.

While maintaining and providing SBOMs for new and ongoing software projects can be relatively easy, doing so for legacy software involves major challenges. One approach to creating SBOMs traditionally involves manual code inspection and/or consulting the original developers, both of which are time-consuming, error-prone, and, in the worst case, impossible. Moreover, these types of software often lack proper documentation, which makes the integration of SBOMs even more challenging. Ideally, this problem can be solved, if we can build tools capable of the automatic generation of SBOM from software executables or distribution packages.

Automatic SBOM generation from a software executable or distribution package is technically a challenging task due to several reasons. Software packages are complex, with extensive dependencies, code obfuscation, and frequent updates. Additionally, the complexity is increased by the lack of a universal SBOM standard, making it necessary for SBOM generators to be adaptable to different standards. Only a handful of tools claim the ability to accurately produce SBOMs from software distribution packages although their capabilities are limited to the programming languages (e.g., Java, JavaScript, Python) primarily used in writing the software. As the area is new and there are not many tools claiming to be automatic SBOM generators, there is lack of studies examining the strengths and weaknesses of such tools.

In this paper, we present the first study to compare the capabilities of tools that claim the ability to generate SBOM from JavaScript (Node Package Manager) npm project distributions. Because such tools are typically language-specific we choose a language, and in this study, we emphasize on the JavaScript ecosystem, as it has been the most used programming language on GitHub since 2014 [1]. We use a robust methodology that includes tool and project selection, SBOM generation, ground truth creation, and comparative metric-driven analysis.

The rest of the paper is organized as follows: In Section 2, we begin by providing background information on SBOM. Moving on to Section 3, we describe the methodology used in our study, including topics such as the selection of tools and projects, SBOM generation, the creation of a ground truth, and the explanation of the metrics used for comparison. Section 4 presents a complete qualitative and quantitative analysis of four SBOM tools. Section 5 outlines the limitations of our work. Finally, in Section 6, we present conclusions and provide directions for future research.

2 BACKGROUND AND RELATED WORK

There have been many studies on software quality assurance involving the investigation of bug patterns [19, 20, 24], security vulnerabilities [11, 21], code smells [16, 30, 31], code quality [3, 16], human aspects [6, 7, 12, 13, 17, 18] of software development and maintenance as well as comparison of methods/tools [14, 15, 22, 23] for measuring such aspects.

The concept of a Bill of Materials (BOM) is not new either and it has its roots in the manufacturing sector. However, the adaptation of BOMs for software, commonly termed as SBOMs [29], has emerged as a relatively recent phenomenon. The increasing complexity of software projects, combined with the growing emphasis on security and compliance, has necessitated an organized approach to understanding and documenting software components. Reflecting this urgency, the Executive Order on Improving the Nation's Cybersecurity [5] has emphasized the significance of SBOMs as a fundamental mechanism to increase transparency and strengthen the software supply chain. By defining minimum elements and fostering a foundation for software transparency, the SBOM initiative seeks to facilitate effective vulnerability management and elevate trust in our digital infrastructure [25].

The constantly evolving software ecosystem is served by a variety of SBOM formats. While notable formats like the Software Package Data Exchange (SPDX) of the Linux Foundation place a strong emphasis on licensing and copyright information, ISO/IEC-standardized Software Identification Tags (SWID) place a strong emphasis on uniquely identifying software components [28].

However, amidst this spectrum of options, CycloneDX has distinguished itself. Released in its latest version, v1.5, in June 2023, it has not only witnessed rapid adoption but is also the preferred choice of numerous organizations [10]. Currently, it's estimated that about 100,000 organizations have integrated CycloneDX into their operational framework [9]. CycloneDX, which was created as a part of the Open Worldwide Application Security Project (OWASP) project, successfully strikes a compromise between thorough documentation and ease of use, making it both machine-readable and human-parsable. Its emphasis on extensibility, which covers everything from hardware and firmware to libraries and proprietary frameworks, underlines both its expanding prominence in the software industry and its importance to our research.

While automatic SBOM generation tools are scarce, studies comparing such tools are even scarcer. Recently, Balliu et al. [4] have explored the effectiveness of SBOM tools in Java Maven projects. Their study inspires us to evaluate SBOM generation tools for JavaScript projects. While they focused on Java projects, we examine the performance of tools claiming to identify SBOMs in JavaScript projects.

3 METHODOLOGY

We test SBOM generators or generation tools against popular JavaScript npm projects to evaluate their effectiveness. Our methodology, as shown in Figure 1, starts with the selection of both the projects and the SBOM tools. We then employ these tools to produce SBOMs and carry out a comparative analysis of the outcomes. The following subsections detail each step of our approach.

3.1 Selection of SBOM Generators

We start by searching on GitHub for projects labeled 'sbom' with at least 100 stars, as this typically indicates active community involvement and support.

Following the initial search, we identify a total of 36 projects that match the criteria. However, upon examining the documentation of each project, we find that many of these projects primarily focus on analyzing existing SBOMs to identify potential vulnerabilities in other projects. This means they take SBOMs as input and output vulnerability of other projects. They do not provide SBOMs themselves.

For the requirements of our study, we filter the tools based on:

(1) Their capability to create SBOMs in the CycloneDX format.
(2) Support for JavaScript npm projects.
(3) Command-line functionality for straightforward scripting and automation.

Through this process of elimination, we are left with four key tools: OSS Review Toolkit (ORT), cyclonedx-node-module (cnn), syft, and CycloneDX Generator (cdxgen). We choose to work with the latest stable versions of these tools as of July 15, 2023. The specific versions of each tool utilized are mentioned in the second row of Table 1.

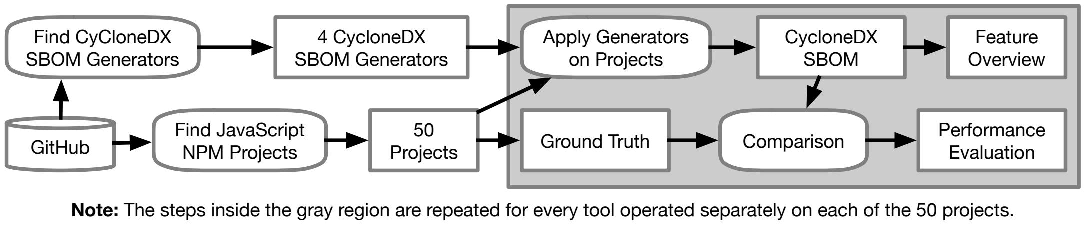

Find CycloneDX SBOM Generators
4 CycloneDX SBOM Generators
Apply Generators on Projects
CycloneDX SBOM
Feature Overview
GitHub
Find JavaScript NPM Projects
50 Projects
Ground Truth
Comparison
Performance Evaluation
Note: The steps inside the gray region are repeated for every tool operated separately on each of the 50 projects.

Figure 1: Procedural steps in the methodology our study

Table 1: Overview of four SBOM generators

	ORT	**cnn**	**syft**	**cdxgen**
Version	841f770bfb	1.13.0	0.85.0	9.2.2
Identifies Components and Versions	Yes	Yes	Yes	Yes
Details Dependencies	No	Yes	No	Yes
CycloneDx Format	1.4	1.4	1.4	1.5
Provides Checksum	Yes	No	No	Yes
Reproducibility	Yes	Yes	Yes	Yes

3.2 Selection of JavaScript Projects

In our study, finding suitable npm projects requires a systematic approach based on specific file identifiers. The presence of a package.json file at the root of a project typically indicates npm usage. However, this doesn't exclude the potential use of other package managers, such as Yarn or pnpm. The distinct package-lock.json file serves as a clear indicator of npm usage. In contrast, the presence of a yarn.lock file signals Yarn as the chosen package manager.

To optimize our search, we utilize the advanced search feature of GitHub. We search for JavaScript repositories that contain a package-lock.json file and exclude those with yarn.lock or pnpm-lock.yaml files. To ensure the relevance and maturity of our chosen repositories, we give preference to those with most star count, using it as a metric for quality and community interest.

After applying our search criteria, we select the first 50 JavaScript npm projects that fit our criteria to serve as the subjects of our study. A comprehensive overview of these projects can be found in Table 2. Each project is delineated by its name, star count (in thousands), size (quantified in thousand lines of code, kLOC), and the number of npm packages it incorporates. For instance, the `javascript-algorithms` project boasts 174 thousand stars and comprises 1783 kLOC, utilizing 11 npm packages.

The chosen projects offer a varied mix in both size and recognition. We have projects with substantial lines of code reflecting their extensive frameworks, while others are more concise, representing specialized tools. Such a diverse sample provides a comprehensive representation, laying the foundation for drawing broader and more generalizable insights from our study.

3.3 Generation of SBOMs and Ground Truths

3.3.1 SBOM Generation Process. For each of the 50 selected npm projects, we apply the four chosen SBOM tools. While the majority of tools produce JSON formatted SBOMs, ORT outputs in XML. To maintain a consistent data format for our subsequent analyses, we employ a Python script to convert the XML output from ORT to JSON.

3.3.2 Ground Truth Establishment. To accurately evaluate the effectiveness of SBOM tools, a ground truth, a benchmark against which tool outputs can be measured, is indispensable. We generate this benchmark using the npm list command, which offers a comprehensive tree of dependencies for each project. Employing the `-depth` parameter with this command enables us to obtain a granular view of dependencies, detailing which components are interdependent. A custom Python script then processes this tree, extracting vital details: component names, their versions, and interdependencies.

3.4 Comparative Analysis of SBOMs

3.4.1 Metrics and Evaluation Criteria. Our evaluation focuses on three primary components: component names, versions, and dependencies. The evaluation revolves around three key metrics:

- **Accuracy**: Represents the fraction of correctly identified components to the total components. Given that there are no true negatives (TN) in this evaluation, since SBOM tools do not output what is absent, the formula reduces to:

$$\text{Accuracy} = \frac{TP}{TP + FP + FN} \tag{1}$$

  Where:
  - TP: True Positives - Correctly identified components.
  - FP: False Positives - Incorrectly identified components.
  - FN: False Negatives - Missed components.
- **Precision**: Assesses the fraction of correctly identified components among all identified components.

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

- **Recall**: Measures the fraction of correctly identified components to all actual components in the ground truth.

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

Together, these metrics furnish a holistic picture of each tool's performance: accuracy gives a general overview, precision addresses the trustworthiness of identified components, and recall gauges the tool's efficacy in pinpointing all pertinent components.

3.4.2 Comparative Methodology. We employ a systematic approach to evaluate the outputs of each SBOM tool against the ground truth. The methodology unfolds as follows:

Table 2: Details of the JavaScript projects

Project Name	# of stars (thousands)	Size (kLOC)	npm packages	Project Name	# of stars (thousands)	Size (kLOC)	npm packages
javascript-algorithms	174	1783	11	ws	20.2	509	9
bootstrap	165	1564	49	joi	20.1	1254	12
javascript	136	948	2	lowdb	19.9	1361	18
three.js	94	2018	20	p5.js	19.8	2084	48
Web-Dev-For-Beginners	75.1	782	1	hyperapp	19	53	2
express	61.7	630	48	vue-admin-template	18.8	2532	33
html5-boilerplate	55.2	1023	17	htmx	18.5	1555	10
Leaflet	38.2	1264	29	react-testing-library	18.1	2785	13
zx	38.2	3399	22	MagicMirror	18	2750	35
hexo	37.3	2096	37	mysql	17.9	249	9
phaser	35.1	2274	22	dotenv	17.6	2298	10
wtfjs	31.6	888	13	You-Dont-Need-Lodash-Underscore	17.3	418	6
tesseract.js	31.3	1300	35	statsd	17.2	979	7
standard	28.4	551	16	material	16.6	2517	67
webtorrent	28.2	2204	76	pouchdb	15.9	1755	69
layui	27.9	365	10	markdown-it	15.9	1144	40
JavaScript	27.5	758	10	jasmine	15.6	871	17
Modernizr	25.5	1432	33	eleventy	14.9	1175	53
eslint	23.3	2492	95	riot	14.8	2171	31
PhotoSwipe	23.1	1772	13	Luckysheet	14.5	2687	41
core-js	22.8	884	45	filepond	14	2136	21
mostly-adequate-guide	22.7	682	3	shelljs	13.9	724	17
highlight.js	21.9	1765	34	mithril.js	13.7	627	22
scrollreveal	21.7	1204	27	mathjs	13.5	2563	65
ava	20.5	3016	57	mui	13.4	1259	23

(1) **Component Verification**: We juxtapose the component names and versions from the SBOM tools' outputs with the established ground truth, calculating accuracy, precision, and recall.
(2) **Dependency Verification**: Our assessment of the tools' proficiency in listing dependencies is twofold:
  (a) **Dependency Mapping**:
    - Using the generated SBOMs, we establish a dictionary associating each component with its subsequent dependencies.
    - In parallel, the command npm list yields a hierarchized dependency structure. We parse this to form a comparable mapping that links each package (with its version) to direct dependencies.
  (b) **Scoring Mechanism**:
    - By contrasting the identified dependencies from the SBOMs with the ground truth, we allocate scores for correct matches.
    - In an ideal match, the score peaks at the total dependency count presented in the SBOM dictionary.
    - The ground truth score originates from the aggregate dependency count tied to each package, derived from the npm list output.
(3) **Results Aggregation**: The average scores for each parameter are calculated after findings from all 50 projects are combined. This offers a comprehensive evaluation of each tool's performance.

Following our research, we discuss the implications of our findings, highlighting the benefits and potential areas for improvement for each SBOM tool related to JavaScript npm projects.

4 RESULTS

4.1 Comparison of Features

The overview of four SBOM generators - ORT, cnn, syft, and cdxgen - are detailed in Table 1. Notably, all four generators exhibit the capability to identify components and their respective versions. The performance of these identifications is evaluated in Section 4.2.1 and Section 4.2.2. A crucial aspect of SBOM generators is their ability to provide a proper dependency list or tree, delineating the interrelationships between components and their versions. Both cnn and cdxgen demonstrate this critical functionality, whereas ORT and syft do not. The effectiveness of cnn and cdxgen in providing these dependencies is assessed in Section 4.2.3.

ORT, cnn, and syft generate SBOMs in CycloneDX 1.4, while cdxgen produces them in CycloneDX 1.5. Additionally, checksums are crucial for verifying component integrity. In this regard, ORT employs the SHA-1 algorithm, and cdxgen utilizes SHA-512, whereas cnn and syft do not offer any checksum provision. To assess the reproducibility of the SBOM generators, we generate SBOMs twice for each of the 50 projects using all four generators, resulting in 400

SBOMs. However, 36 instances result in either empty SBOM files or no SBOMs generated at all. Consequently, we evaluate 364 SBOMs and find that all four generators exhibit the capability to reproduce the SBOM, except the `serialNumber` and `timestamp`, which are expected to vary with each SBOM generated.

4.2 Performance Evaluation

In the evaluation of SBOM generators, three crucial aspects are considered: identification of component names, detection of their respective versions, and detailing of dependencies among them. Each of these aspects are vital and need to be evaluated separately. To facilitate a comprehensive comparison, three key metrics are employed: accuracy, precision, and recall.

4.2.1 Evaluation of Component Name Identification. In this section, the ability of SBOM generators to accurately identify component names is assessed. Table 3 encapsulates the performance of each tool over 50 projects. Table 3 reveals that cnn outperforms the other tools, registering the highest average accuracy (96.29%) and precision (99.97%), followed sequentially by cdxgen and syft. Conversely, ORT lags behind, exhibiting the lowest average accuracy (74.06%), precision (74.75%), and recall (76.29%). While this table provides a holistic overview, a more granular analysis is required to understand the performance of each tool.

Table 3: Performance in identifying component names

	Accuracy (%)	Precision (%)	Recall (%)
ORT	74.06	74.75	76.29
cnn	96.29	99.97	96.32
syft	80.49	80.78	85.68
cdxgen	83.30	83.91	97.49

Figure 2 depicts the distribution of the accuracy of the four SBOM generators in identifying component names over selected 50 projects using box plots. Here, the median value is denoted as 'Med' for convenience. We notice that ORT achieves the highest median accuracy (99.12%), while cdxgen has the lowest median accuracy of 97.31%. The comparison of these medians with the average or mean accuracies from Table 3, indicates the supremacy of medians over means. This suggests left-skewed distributions of accuracy for all four SBOM generators.

We can see from Figure 2, that the ORT tool has no outlier as the range and Interquartile Range (IQR) covers the whole 0 to 100% area. This means that accuracies are spread all over the area, highlighting the inconsistency in its performance. Moreover, ORT gives 0% accuracy for 11 projects where it cannot produce SBOM or gives empty SBOM. This contributes in ORT having the lowest mean accuracy, even though it has the highest median value among all the tools. On the contrary, cnn has seven outliers but the range and IQR of it is much compact compared to other SBOM tools. This very low spread specifies the consistency of cnn in finding component names.

Another observation is that for the `mostly-adequate-guide` project, both cnn and cdxgen score their lowest accuracy of 19.84% and 7.87%, respectively. Whereas ORT and syft, each achieves accuracy greater than 95% for the same project. This highlights a

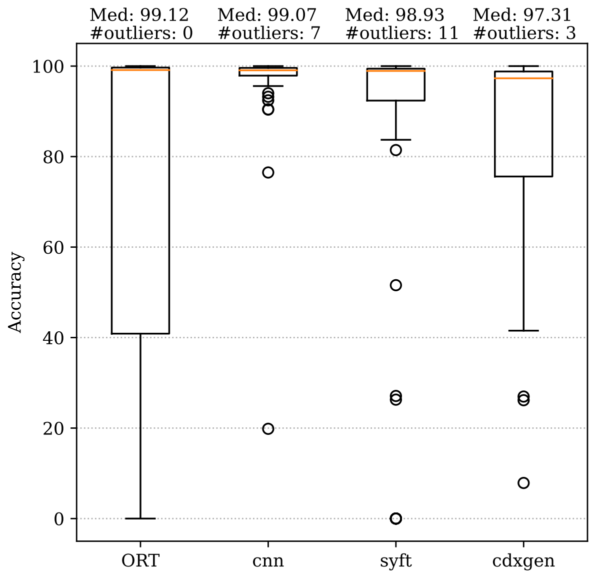


Figure 2: Distribution of accuracy in component detection

crucial point that for every project, at least one or more generators consistently achieved accuracy exceeding 95%.

In terms of overall accuracy, cnn stands out as it has the highest mean accuracy, the second highest median accuracy, and the lowest spread. The performance of cnn in generating component name is consistent, thus making it more reliable among the other generators.

The precision of the four SBOM generators when identifying component names is displayed in Figure 3. Both cnn and ORT display a median precision of 100%, whereas the median precision value of cdxgen is lowest (97.96%). Here the mean precision (Table 3) and median precision of cnn align closely, denoting resemblance to symmetric distribution. But the median is much higher than mean for other tools which indicates left-skewed distribution of precision across the chosen 50 projects.

In case of cnn, the precision spread measured in terms of the IQR and range is lowest among all four tools. This makes cnn the most consistent tool in terms of precision, although it has 6 outliers. For instance, the least precision for cnn is observed in the `joi` project at 99.46%. Conversely, despite having a perfect median precision of 100%, ORT gives varying precision values across different projects. This marks ORT as inconsistent in correctly giving component name. Similarly, syft also exhibits inconsistency with 10 outliers where six outliers have 0% precision.

The boxplots in Figure 4 illustrates the recall distribution of the four SBOM generators. Interestingly, the medians of the four SBOM generators are very close, spanning from 99.12% to 99.60%. The higher medians compared to the mean recall values from Table 3, infer left-skewness in recall measure. Moreover, the low spread of all tools implies consistent recall performances in component name identification. It is noticed that all the tools have outliers, with ORT having the most at 12, and cdxgen with the fewest at four.

In terms of recall, although cdxgen reports the highest mean and second highest median, it tends to identify more component names than other tools. This increases the likelihood of matching given

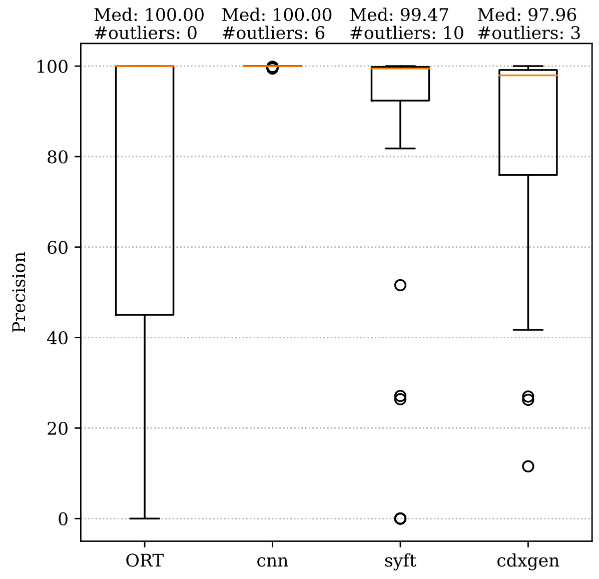


Figure 3: Distribution of precision in component detection

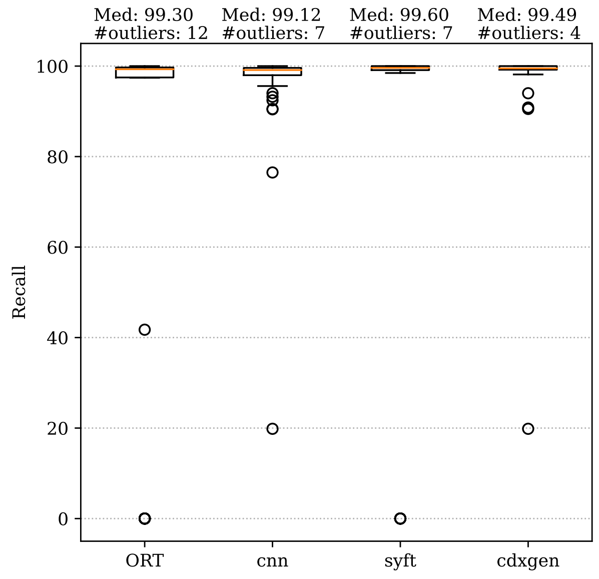


Figure 4: Distribution of recall in component detection

names with actual names. But this also results in a higher number of incorrect identifications, impacting its accuracy, precision as well as recall.

For instance, in the `bootstrap` project, out of the actual 750 component names, cdxgen identifies 758, whiles cnn identifies 745. Of these, cdxgen correctly matches 746 names, one more than that of cnn. Despite the higher recall of cdxgen, cnn demonstrates 100% precision providing a count closer to the actual number of component names. It is worth noting that for the `mostly-adequate-guide` project, even though cnn gives lowest accuracy and recall (both 19.84%), still achieves 100% precision by correctly identifying 74 names out of the 373 actual component names.

4.2.2 Evaluation of Version Identification. Table 4 outlines the performance of four SBOM generators in identifying component versions. It is evident that cnn stands out with an accuracy rate of 84.28% and an impressive precision rate of 99.90%. But its recall rate of 84.36% is slightly lower than cdxgen, which has the highest recall rate at 92.57%. Comparatively, syft exhibits a balanced performance with 80.44% accuracy, 80.48% precision, and 85.94% recall. However, ORT lags behind the other tools in all three metrics. To gain a deeper understanding of the performance of each tool, our subsequent analysis employs box plots. These graphical representations reveal not only the central tendency but also the spread of data, the variability and potential outliers.

Table 4: Performance in identifying component versions

	Accuracy (%)	Precision (%)	Recall (%)
ORT	74.22	74.52	76.44
cnn	84.28	99.90	84.36
syft	80.44	80.48	85.94
cdxgen	80.58	83.68	92.57

Figure 5 provides a visual representation of the distribution of accuracies for the four SBOM generators in identifying component versions. The median accuracy of cnn is just 86.69%, whereas the medians for the other generators are well above 97%. Since the mean and median of cnn are relatively close, this results in a more symmetrical box plot. Conversely, the medians of other tools are significantly higher than the mean values listed in Table 4, leading to left-skewed box plots.

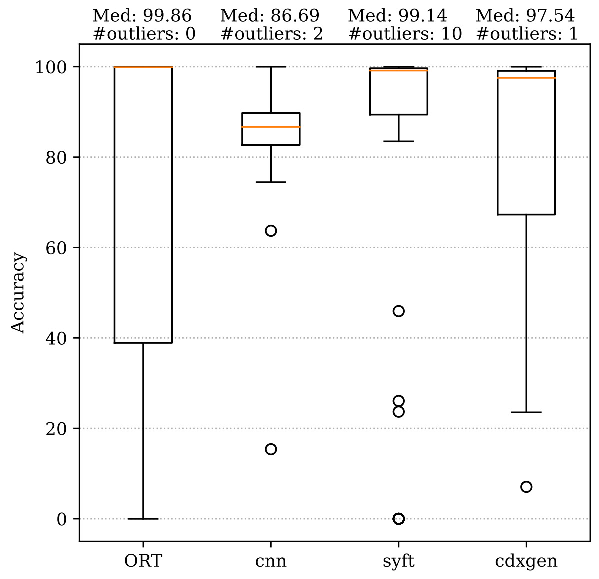


Figure 5: Distribution of accuracy for version identification

Although the IQR and range of syft is lowest among others, it has the highest count of outliers at 10. Six of these outliers have

0% accuracy which makes syft inconsistent. In contrast, the data spread measured of cnn, by the IQR and range, is low. This ensures consistent component version identification, although neither its mean nor median exceeds 90

ORT gives the highest median accuracy of 99.86%, but the spread is very high. Moreover, it manifests 0% accuracy for 11 projects which causes the performance inconsistency. In spite of having a high spread similar to ORT, cdxgen has only one outlier, for the `mostly-adequate-guide` project where it achieves just 7.06%.

The precision of the four SBOM generators in identifying component versions is depicted in Figure 6. Both cnn and ORT achieve the perfect median score, while cdxgen gives the lowest median of 97.83%. Except for cnn, the median of other tools is significantly higher than the mean values recorded in Table 4.

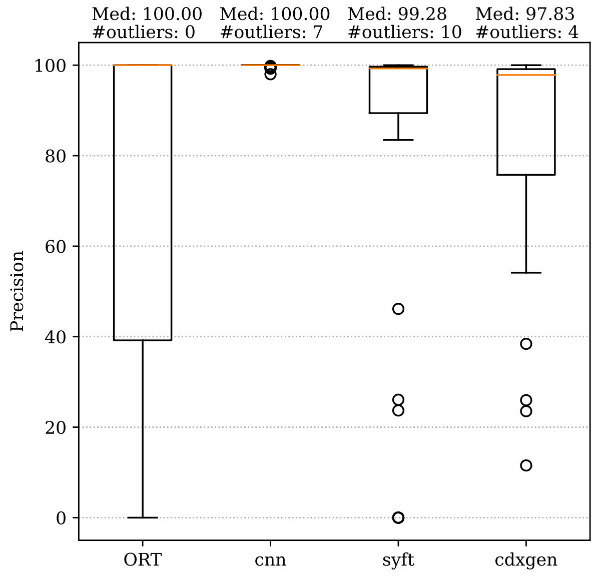


Figure 6: Distribution of precision for version identification

Upon a closer inspection, we find that the IQR and range of cnn have a tiny variation and has 7 outliers. However, the lowest precision (98%) recorded by cnn is for the `javascript` project which is a outlier. Furthermore, cnn achieves 100% precision in 43 out of the 50 projects assessed.

ORT displays a vast spread in precision, ranging from 0% to 100%. This is why ORT shows no outliers despite having multiple instances 0% precision. However, this suggests potential inconsistencies in its performance. Similarly, the reliability of syft is also compromised by the presence of 0% precision values. While cdxgen has no 0% precision, the spread of precisions is high and it has four outliers of lower precisions. Additionally, cdxgen has the tendency of providing extra component names alongside their versions.

Figure 7 portrays the recall rates of the four SBOM generators. The median recall of both ORT and syft achieve the highest rate of 100%, while cnn has the lowest recall of 86.69%. As noticed in case of precision in identifying component versions, similarly the median recall of other tools is significantly higher than mean recall, except for cnn.

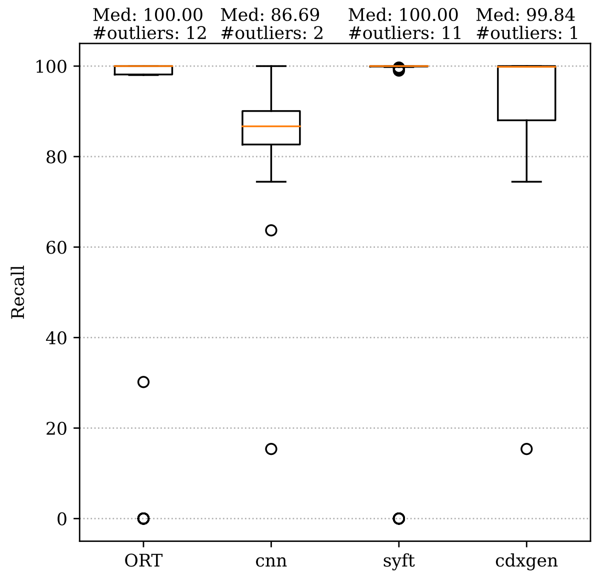


Figure 7: Distribution of recall for version identification

Syft, while having a narrow spread i.e. low IQR and range, it has 11 outliers. Among these 11 outliers, seven presents a recall value of 0%. The second placeholder ORT has the highest number of outliers and 0% (11 times) recall. In contrast, cnn offers more consistent performance, despite having lower mean and median recall with just two outliers. However, cdxgen demostrates relatively stable recall and has just one outlier. Both cdxgen and cnn identify the `mostly-adequate-guide` project with the lowest recall of 15.38%.

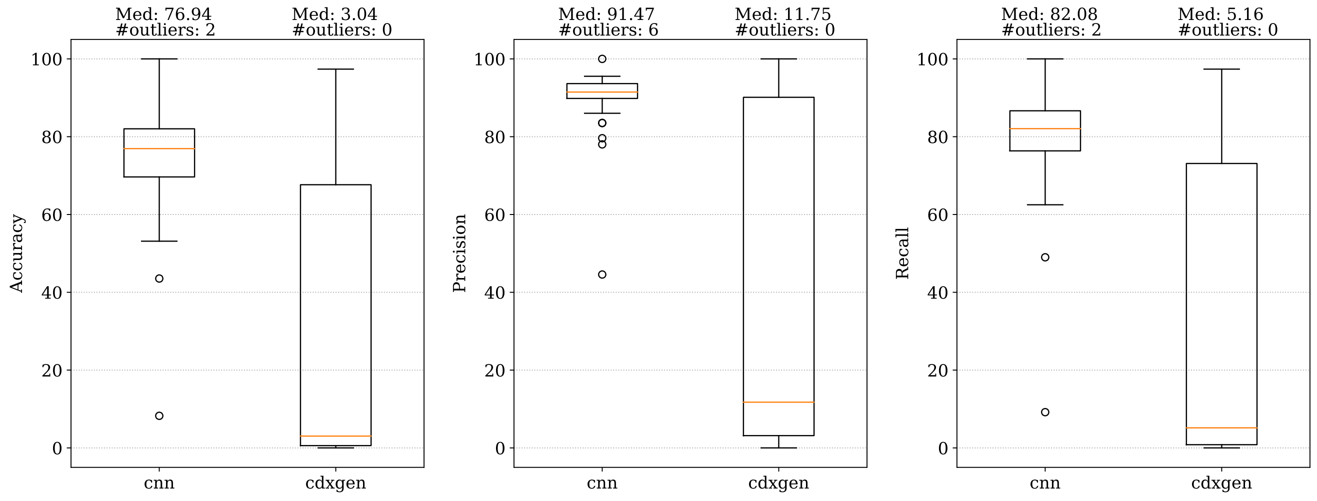
4.2.3 Evaluation of Dependency Detection. The performance of cnn and cdxgen in identifying component dependencies is summarized in Table 5. ORT and syft do not offer dependency identification features, hence they are excluded from this particular analysis. cnn stands out with an average accuracy of 74.06%, a precision rate of 90.22%, and a recall value of 79.46%. In contrast, cdxgen lags behind considerably, with recorded average accuracy, precision, and recall scores of just 29.62%, 40.94%, and 32.09%, respectively. These results suggest that cnn outperforms cdxgen in the identification of component dependencies.

Table 5: Performance in capturing component dependencies

	Accuracy (%)	Precision (%)	Recall (%)
cnn	74.06	90.22	79.46
cdxgen	29.62	40.94	32.09

To gain a more comprehensive understanding, Figure 8 illustrates the performance distribution using boxplots, representing accuracy, precision, and recall metrics for cnn and cdxgen.

From Figure 8(a) and (c), it is observed that accuracy and recall appear to demonstrate similar patterns. cnn exhibits a median accuracy of 76.94% and a recall rate of 82.08%, both of which is significantly higher than cdxgen. The symmetry observed in the box plots of cnn is due to the median accuracy and recall of it being



(a) Accuracy of dependency detection (b) Precision of dependency detection (c) Recall rate for dependencies

Figure 8: Performance of SBOM generators in component dependency identification

quite similar to its mean (average) values listed in Table 5. On the contrary, the accuracy and recall distribution of cdxgen is right skewed, as the median is closer to the bottom of the box and the whisker is shorter on the lower end of the box.

The spread, i.e., the IQR and range of cnn, is narrower than that of cdxgen, reflecting its consistency in identifying component dependencies. Notably, cnn displays only two outliers, identified in the projects `wtfjs` and `mostly-adequate-guide`. Conversely, cdxgen does not have any outliers since its IQR and range extend from 0% to over 90%. However, cdxgen fails to report any dependencies in seven instances, resulting in a 0% score.

In Figure 8(b), cnn demonstrates an impressive median precision of 91.47%, while cdxgen exhibits a very low precision of 11.75%. This suggests that the reported dependencies of cnn, though occasionally fewer than the actual count, are represented with higher accuracy. The spread of cnn is also very small, indicating consistent precision across projects. cnn has six outliers, including the highest precision of 100% in `hyperapp` project and the lowest precision of 44.57% in the `mostly-adequate-guide` project. In contrast, cdxgen has instances with 0% precision, implying instances where it fails to generate any dependencies, a shortcoming not observed in cnn.

In summary, cnn consistently outperforms cdxgen in terms of accuracy, precision, and recall for detecting component dependencies. However, cnn has instances where it presents below-average performance. This implies that while cnn is typically a more dependable option for dependency detection, there might be particular scenarios or types of dependencies where it faces challenges.

5 THREATS TO VALIDITY

Our study, while providing comprehensive insights into SBOM generation for JavaScript npm projects, has its intrinsic limitations tied to the methodology. In choosing SBOM tools, we prioritize those generating CycloneDX formatted outputs, potentially excluding other notable formats like SPDX. Additionally, we give preference to tools that offer command-line functionality, which might leave out proficient tools with different interfaces.

In terms of project selection, our approach prioritizes JavaScript npm projects. While this offers a focused perspective, it might not capture the essence of projects employing a mix of package managers beyond npm. However, it is crucial to understand that not all SBOM tools equip themselves to effectively capture every nuance of each JavaScript package manager.

Despite these limitations, our research provides a comprehensive picture of the effectiveness of SBOM tools within the context of JavaScript npm projects, establishing itself as a valuable resource for both academia and industry practitioners.

6 CONCLUSION

In this paper, we present the first known study comparing four SBOM generation tools which are ORT, cnn, syft, and cdxgen. We attempt to understand the unique features and overall abilities of these tools. Our methodology systematically evaluates these four SBOM generation tools by applying them to 50 popular open-source JavaScript npm projects. We establish ground truths and conduct comparative analyses using accuracy, precision, and recall metrics in terms of component name, version, and dependency. Moreover, the use of box plots enables granular analysis of the distribution, spread, and potential outliers in the performance of these metrics.

Our study highlights the superior performance of cnn in effectively identifying component names and versions. In contrast, even while ORT and syft show their distinctive advantages, their omission of dependencies makes them less suitable for use in real-world applications. This absence is especially crucial in the context of SSC, where an understanding of the complex web of dependencies is not only desirable but also essential, particularly when it comes to identifying and addressing vulnerabilities.

Although cnn and cdxgen can both provide dependency information, cdxgen's unreliable performance, including fully failing to supply details at times, makes it questionable for critical applications. On this front, cnn appears as the only tool that can be relied upon. Its consistent provision of component names, versions, and dependencies underscores its potential as an invaluable tool in SBOM generation, especially when precision is vital.

This pioneering research highlights the capabilities and limitations of emerging SBOM generation tools for JavaScript ecosystems. Our quantitative results and comparative methodology will help developers in selecting and integrating SBOMs into projects. Our findings also furnish valuable feedback for tool creators to enhance automated SBOM generation from software packages.

In our future work, we intend to focus on the effectiveness of SBOM generation tools in complex hybrid environments utilizing many package managers. We can expand the assessment to additional programming languages, open-source projects, and SBOM generation tools. Examining the intricate network of dependencies can provide insights that lead to creating even more advanced SBOM generating tools. Furthermore, integrating SBOM generation into the fabric of continuous integration and deployment (CI/CD) systems would enable dynamic SBOM analyses and adaptations.

ACKNOWLEDGEMENT

This work is supported in part by the ISU-CAES Seed Grant at the Idaho State University, USA.

REFERENCES

[1] 2023. *The top programming languages*. Retrieved Sep 5, 2023 from https://octoverse.github.com/2022/top-programming-languages

[2] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad. 2021. Solar Winds Hack: In-Depth Analysis and Countermeasures. In *12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, 1–7.

[3] D. Alwad, M. Panta, and M. Zibran. 2018. An Empirical Study of the Relationships between Code Readability and Software Complexity. In *27th International Conference on Software Engineering and Data Engineering*. 122–127.

[4] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger. 2023. Challenges of Producing Software Bill Of Materials for Java. *arXiv preprint arXiv:2303.11102* (2023).

[5] Solution Brief. 2021. Executive order on improving the nation's cybersecurity. (2021).

[6] A. Champa, M. Rabbi, F. Eishita, and M. Zibran. 2023. Are We Aware? An Empirical Study on the Privacy and Security Awareness of Smartphone Sensors. In *21st IEEE International Conference on Software Engineering, Management and Applications (SERA)*. 1–8 (to appear).

[7] A. Champa, M. Rabbi, M. Zibran, and M. Islam. 2023. Insights into Female Contributions in Open-Source Projects. In *20th IEEE International Conference on Mining Software Repositories*. 357–361.

[8] Cybersecurity and Infrastructure Security Agency (CISA). 2023. *Software bill of materials (SBOM)*. Retrieved Sep 5, 2023 from https://www.cisa.gov/sbom

[9] FOSSA. 2023. *A practical guide to cyclonedx*. Retrieved Sep 5, 2023 from https://fossa.com/learn/cyclonedx#cyclonedx-vs-spdx

[10] M. Fulton. 2021. *Sonatype Embraces CycloneDX Standard for Integrating Software Bills of Materials (SBOMs)*. Retrieved Sep 5, 2023 from https://www.sonatype.com/press-releases/sonatype-embraces-cyclonedx-standard-for-integrating-software-bills-of-materials-sboms

[11] M. Islam and M. Zibran. 2016. A Comparative Study on Vulnerabilities in Categories of Clones and Non-Cloned Code. In *10th IEEE Intl. Workshop on Software Clones*. 8–14.

[12] M. Islam and M. Zibran. 2016. Exploration and Exploitation of Developers' Sentimental Variations in Software Engineering. *Internation Journal of Software Innovation* 4, 4 (2016), 35–55.

[13] M. Islam and M. Zibran. 2016. Towards Understanding and Exploiting Developers' Emotional Variations in Software Engineering. In *SERA*. 185–192.

[14] M. Islam and M. Zibran. 2017. A Comparison of Dictionary Building Methods for Sentiment Analysis in Software Engineering Text. In *ESEM*. 478–479.

[15] M. Islam and M. Zibran. 2018. A Comparison of Software Engineering Domain Specific Sentiment Analysis Tools. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. 487–491.

[16] M. Islam and M. Zibran. 2018. On the Characteristics of Buggy Code Clones: A Code Quality Perspective. In *12th IEEE Intl. Workshop on Software Clones*. 23 – 29.

[17] M. Islam and M. Zibran. 2018. Sentiment Analysis of Software Bug Related Commit Messages. In *SEDE*.

[18] M. Islam and M. Zibran. 2018. Sentiment Analysis of Software Bug Related Commit Messages. In *27th Intl. Conference on Software Engineering and Data Engineering*. 3–8.

[19] M. Islam and M. Zibran. 2020. How Bugs Are Fixed: Exposing Bug-fix Patterns with Edits and Nesting Levels. In *35th ACM/SIGAPP Symposium on Applied Computing*. 1523–1531.

[20] M. Islam and M. Zibran. 2021. What Changes in Where? An Empirical Study of Bug-Fixing Change Patterns. *ACM Applied Computing Review* 20, 4 (2021), 18–34.

[21] M. Islam, M. Zibran, and A. Nagpal. 2017. Security Vulnerabilities in Categories of Clones and Non-Cloned Code: An Empirical Study. In *11th ACM/IEEE Intl. Symposium on Empirical Software Engineering and Measurement*. 20–29.

[22] R. Joseph, M. Zibran, and F. Eishita. 2021. Choosing the Weapon: A Comparative Study of Security Analyzers for Android Applications. In *Intl. Conference on Software Engineering, Management and Applications*. 51–57.

[23] D. Murphy, M. Zibran, and F. Eishita. 2021. Plugins to Detect Vulnerable Plugins: An Empirical Assessment of the Security Scanner Plugins for WordPress. In *Intl. Conference on Software Engineering, Management and Applications*. 39–44.

[24] A. Rajbhandari, M. Zibran, and F. Eishita. 2022. Security Versus Performance Bugs: How Bugs are Handled in the Chromium Project. In *Intl. Conference on Software Engineering, Management and Applications*. 70–76.

[25] National Telecommunications and Information Administration (NTIA). 2023. *Minimum Elements for a Software Bill of Materials (SBOM)*. Retrieved Sep 5, 2023 from https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf

[26] Y. Wang. 2023. The Power of Openness-How Open Source Software is Reshaping Software Engineering and Industrial Adoption. (2023). https://doi.org/10.36227/techrxiv.23896002.v1

[27] A. Wirth. 2022. Log Jam: Lesson Learned from the Log4Shell Vulnerability. *Biomedical Instrumentation & Technology* 56 (2022). https://doi.org/10.2345/0899-8205-56.3.72

[28] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu. 2023. An empirical study on software bill of materials: Where we stand and the road ahead. *arXiv preprint arXiv:2301.05362* (2023).

[29] N. Zahan, E. Lin, M. Tamanna, W. Enck, and L. Williams. 2023. Software Bills of Materials Are Required. Are We There Yet? *IEEE Security & Privacy* 21, 2 (2023), 82–88.

[30] M. Zibran and C. Roy. 2013. Conflict-aware Optimal Scheduling of Code Clone Refactoring. *IET Software* 7, 3 (2013), 167–186.

[31] M. Zibran, R. Saha, C. Roy, and K. Schneider. 2013. Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study. In *28th ACM/SIGAPP Symposium on Applied Computing*. 1123–1130.