











Challenges of Producing Software Bill of Materials for Java

Musard Balliu , Benoit Baudry , Sofia Bobadilla , Mathias Ekstedt , Martin Monperrus ,
Javier Ron , Aman Sharma , Gabriel Skoglund , César Soto-Valero , and
Martin Wittlinger  | KTH Royal Institute of Technology

Software bills of materials (SBOM) promise to become the backbone of software supply chain hardening. We deep-dive into six tools and the SBOMs they produce for complex open source Java projects, revealing challenges regarding the accurate production and usage of SBOMS.

Modern software applications are virtually never built entirely in-house. As a matter of fact, they reuse many third-party dependencies, which form the core of their software supply chain.¹ The large number of dependencies in an application has turned into a major challenge for both security and reliability.² For example, to compromise a high-value application, malicious actors can choose to attack a less well-guarded dependency of the project.³ Even when there is no malicious intent, bugs can propagate through the software supply chain and cause breakages in applications.⁴ Gathering accurate, up-to-date information about all dependencies included in an application is, therefore, of vital importance.

Introduction

The software bill of materials (SBOM) has recently emerged as a key concept to enable principled engineering of software supply chains. This takes the well-known concept of “bill of materials” for manufacturing physical goods into the world of software development. The purpose of an SBOM is to capture relevant information

about the internals of a software artifact. First and foremost, an SBOM is expected to include a complete inventory of all of the third-party dependencies of the artifact.

Accurate SBOMs are essential for software supply chain management,⁵ vulnerability tracking, build tampering detection,⁶ and high software integrity. For example, software developers leverage SBOMs to identify vulnerable software components in a timely manner. This is usually done by matching software component versions against vulnerability databases and reporting a warning whenever a vulnerable component is part of an application. For example, in 2021, a serious vulnerability present in the popular Java logging component Log4J was discovered. This component was extensively used by a large number of open source and proprietary projects, and consequently, it was a tedious and costly endeavor to identify all impacted projects.⁷ Had all of these Java projects published an SBOM, it would have facilitated the precise identification and remediation of vulnerable applications.

The software supply chain of modern applications includes hundreds of components, and to have humans producing SBOMs by hand is an unreasonable, time-consuming, and error-prone task. Yet, the

Digital Object Identifier 10.1109/MSEC.2023.3302956
Date of current version: 29 August 2023

full automation of SBOM production is a process that poses several challenges.⁸ First, the SBOM must elicit all direct dependencies, which are explicitly declared by the application's developers in a build configuration file, as well as the indirect dependencies that come from the transitive closure of dependencies. Tracking down every single dependency that is being used is hard when software architectures are formed by deeply nested components, some of which are potentially resolved at runtime. Identifying the exact version of a binary dependency in an SBOM is even harder as this requires tracing the binary components back to source code repositories. Second, while some package managers are able to list the dependencies, SBOMs are meant to include extra information about the software supply chain, such as checksums for all dependencies and data about third-party tools used in the build. Finally, the SBOM aims at being both human-readable for auditing and legal cases, as well as machine-readable for automatic verification. These challenges open an exciting area for research and innovation, as witnessed by the recent emergence of many SBOM tools supported by diverse open source communities, startups, and big tech companies alike. From a research perspective, there is a crucial need for laying down systematic foundations of what SBOMs are, and the challenges related to their engineering.

This article presents an in-depth study of SBOM producers in the Java ecosystem. Our focus on Java is motivated as follows. First, it is one of the top-three languages in the world by most notable metrics. Second, its mature ecosystem of third-party dependencies, mainly through Maven, is critical in government services, financial services,⁹ medical infrastructure, and enterprise software systems.¹⁰ Third, SBOM production is intrinsically related to programming language specifics, as it must capture each and every aspect of dependency resolution, compilation, linking, and packaging, all being unique for a given software stack.

For our study, we created a curated selection of six mature and actively maintained SBOM producers. We executed each producer on a set of 26 active open source Java projects. We observed significant variations in the quality of SBOMs generated by these SBOM producers. In particular, they captured a different set of dependencies for the same project. Based on further manual analysis, we highlight urgent challenges and opportunities to consolidate the state-of-the-art of SBOM production, in order to support thorough security and reliability analyses for software supply chains.

SBOM

In 2021, the United States National Telecommunications and Information Administration (NTIA) set out

to identify a minimal set of requirements for SBOMs.¹¹ These requirements outline which data fields should be present, how SBOMs should support automation, and which practices and processes should be employed when creating, distributing, and using SBOMs. The NTIA concluded that three existing formats meet the requirements: CycloneDX, Software Package Data Exchange (SPDX), and Software Identification.

CycloneDX aims to be a standard for bills of materials for software, hardware, software as a service, and operations. It has a strong security focus, originating

Listing 1. Excerpt of a CycloneDX SBOM for the Java project `async-http-client`

```
{ "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "metadata": {
    "timestamp": "2023-02-20T16:14:42Z",
    "tools": [
      { "name": "CycloneDX Maven plugin",
        "version": "2.7.5"
      }
    ],
    "component": {
      "group": "org.asynchttpclient",
      "name": "async-http-client-project",
      "version": "2.12.3",
      "hashes": [ { "alg": "SHA-512",
        "content": "e5435852...7b3e6173" },
        ...
      ],
      "licenses": [...],
      "externalReferences": [ {
        "url": "http://github.com/
        AsyncHttpClient/async-http-client"
      } ],
      "bom-ref": "pkg:maven/org.
      asynchttpclient/async-http-client
      -project@2.12.3?type=pom"
    }
  },
  "components": [
    { "group": "com.sun.activation",
      "name": "jakarta.activation",
      "version": "1.2.2",
      "bom-ref": "pkg:maven/com.
      sun.activation/jakarta.
      activation@1.2.2?type=jar"
    } . . .
  ],
  "dependencies": [ {
    "ref": "pkg:maven/org.asynchttpclient/
    async-http-client-project@2.12.3?type=
    pom",
    "dependsOn": [
      "pkg:maven/com.sun.activation/
      jakarta.activation@1.2.2?type=jar"
    ]
  } . . . ] }
```

from the Open Worldwide Application Security Project. In this article, we focus on the CycloneDX standard. This choice is motivated by the rapid development of the standard, as witnessed by the release of many tools for producing CycloneDX SBOMs.

“Listing 1” shows an excerpt of a CycloneDX SBOM for the Java component `async-http-client`. This particular example contains three root elements—`metadata`, `components`, and `dependencies`—following the CycloneDX standard. The `metadata` element records information about the tool which produced the SBOM and the project on which the producer was executed. The `components` element is a list that includes information about each dependency found in the project. Each component’s item may also contain hashes to help identify its exact version, which can be used to ensure build integrity. The `dependencies` element is a list that records the relationship among all of the previously-listed dependencies. In the example, `jakarta.activation` is a direct dependency of the analyzed project.

We remark that “Listing 1” is a simplified SBOM for the sake of clarity. In practice, an SBOM will contain much more data. The full CycloneDX SBOM of `async-http-client` describes 109 dependencies and provides eight hashes generated through different algorithms for each component (<https://github.com/chains-project/SBOM-2023/blob/main/results-march-2023/async-http-client/cdxgen/bom.cdxgen.json>). Furthermore, the SBOM standard allows recording additional elements, such as references to external resources (e.g., the issue tracker), vulnerabilities, and code signatures.

Given the importance of Java in enterprise and government IT, the production of Java SBOMs is an active area. The critical necessity for grounded and correct Java SBOMs is at the core of this article’s significance. In

what follows, we purposely produce SBOMs for complex multimodule Java applications, which are archetypal of enterprise Java software systems.

Methodology to Study SBOM Producers

The core of our study consists in curating and executing state-of-the-art SBOM production tools on a set of mature Java projects. Then, we perform a comparative analysis of the SBOMs following the methodology illustrated in Figure 1.

SBOM Producers

To curate the list of SBOM producers, we started by identifying producers targeting CycloneDX SBOMs for Java projects. We scanned through all of the candidates from the official CycloneDX tool center and queried GitHub with the keyword “SBOM” for projects with at least 100 stars. This process yielded 24 producers.

We further selected the producers that meet the following criteria. Each selected producer should: 1. produce an SBOM containing the dependencies of the project; 2. be able to analyze Java projects that build with Maven; 3. be open source; and 4. be run as a command-line tool and not only as an online tool. The last two criteria are essential for automating our experiments and for reproducible science.

Ultimately, this process resulted in a curated set of six SBOM producers: `Build-Info-Go`, `CycloneDX-Generator`, `CycloneDX-Maven-Plugin`, `DepsScan`, `jbom`, and `OpenRewrite`, as shown in Table 1. We used all of these producers’ most recent stable releases as of 5 May 2023.

SBOM Conceptual Framework

After a deep analysis of the considered SBOM producers, we postulated the following framework of SBOMs that we will apply to our experimental results.

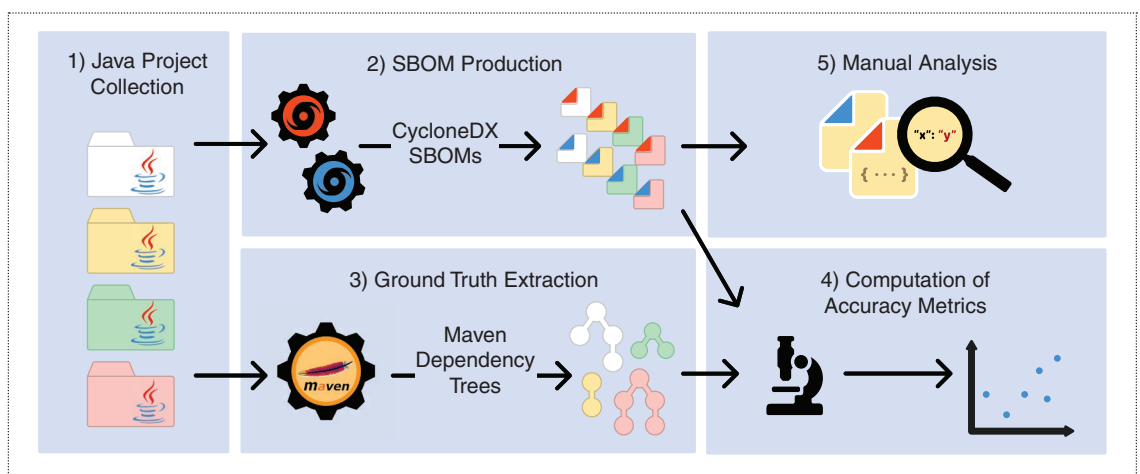


Figure 1. Overview of the methodology to study CycloneDX SBOM production for Java.

- *Build integrity*: SBOMs can contain checksums of software components for verifying build integrity, but the format of checksums is open.
- *Dependency hierarchy*: SBOMs can contain either a flat list of dependencies or structured trees of dependencies, which impacts subsequent consumption.
- *Production step*: SBOMs can be computed at different stages of the build and deploy lifecycle, and this can change the resulting SBOMs significantly.
- *Dependency resolution*: SBOMs must faithfully capture the dependency resolution as it happens in build tools, which is often not documented.

Projects Under Study

To compare the quality of SBOMs generated by different producers, we ran them on a dataset of Java projects. This dataset is meant to include mature, active Java projects that rely on a significant number of dependencies. A recent work on dependency management in Java has curated a list of projects that meet these criteria.¹² Since our work also involves dependency analysis, we decided to reuse their dataset of projects. The dataset includes 31 Maven projects with stable releases and frequent activity, indicating the project's maturity. We excluded *teavam* and *moshi* as these projects have migrated from Maven to using Gradle as the build system, as well as *auto* and *subzero* since they are not valid Maven projects due to the lack of a *pom.xml* file in their root directory. We merged *jenkins-core* and *jenkins-cli* as a single project *jenkins* as we executed SBOM producers at the root directory of the project rather than submodules to avoid dependency resolution errors. This process gave us a set of 26 popular, actively maintained open source Maven projects for our analysis.

Table 2 details the set of analyzed Java projects. Each project is identified by the name and commit at

which we analyzed the project. The projects include between 733 and 1.5 million lines of application code and are composed of up to 211 Maven modules. They have between two and 191 direct dependencies, and between one and 582 indirect dependencies.

Protocol to Compare SBOM Producers

Figure 1 illustrates the five main steps of the protocol for our experiment. Step 2 in Figure 1 is "SBOM Production," where we run each SBOM producer on each project. To support the reproducibility of our experiment, we saved the specific git hash of each project and ran the SBOM producers in a docker container. This SBOM generation procedure is fully automated, and it ensures that there are no interactions among the producers as the SBOM production for each project is isolated and starts in the same state. The repository with our study subjects and the experimental pipeline is publicly available (<https://github.com/chains-project/SBOM-2023>).

An SBOM captures a rich set of information about the software supply chain, including the network of direct and indirect dependencies. As part of our study, we assessed the accuracy of the dependencies in the SBOM with respect to a ground truth. Step 3 in Figure 1 represents the process of extracting the ground truth. We used the complete list of dependencies returned by the command `tree` of the `maven-dependency-plugin@3.4.0`. This plugin is an integral part of the Maven build system, and it is the most common plugin used to perform this single task in the supply chain: `resolve dependencies`. It provides a deterministic dependency tree for a specific version of a Maven project. Moreover, it has been in production since 2007 and is being continuously maintained, with the latest release as recent as 2023. It is very mature and stable, and consequently is the best ground truth for our study.

Table 1. Curated set of SBOM producers subject to our study, supporting Java and the CycloneDX standard.

SBOM Producer	Version	Checksums	Hierarchy	Reproducibility	Production Step	Scope
Build-Info-Go	1.9.3	✓ (3)	✓	✗	Build (Maven compile phase)	✗ (0)
CycloneDX-Generator	8.4.3	✓ (8)	✓	✓	Build (Maven package phase)	✓ (1)
CycloneDX-Maven-Plugin	2.7.8	✓ (8)	✓	✓	Build (Maven package phase)	✓ (1)
Depscan	4.1.2	✓ (8)	✓	✓	Source (static source code)	✓ (1)
jbom	1.2.1	✓ (2)	✗	✗	Analyzed (post maven package phase)	✓ (1)
OpenRewrite	4.45.0	✗ (0)	✓	✓	Build (Maven package phase)	✓ (2)

Table 2. Descriptive statistics of the analyzed Java projects.

Project Name	kLOC	Maven Modules	DD	ID	Total
tika	163	108	186	563	749
alluxio	295	66	143	582	725
jooby	65	54	129	368	497
neo4j	686	124	191	273	464
flink	1,528	211	121	270	391
steady	99	20	78	267	345
para	29	6	82	224	306
jenkins	181	10	99	200	299
accumulo	399	18	121	158	279
selenese-runner-java	21	1	22	114	136
undertow	150	10	28	107	135
handlebars.java	22	11	36	84	120
error-prone	225	10	61	53	114
async-http-client	29	14	40	69	109
couchdb-lucene	3.9	1	25	51	76
mybatis-3	62	1	27	37	64
launch4j-maven-plugin	1.5	1	12	50	62
checkstyle	304	1	22	35	57
orika	43	5	25	30	55
commons-configuration	51	1	33	21	54
spoon	155	1	22	32	54
webcam-capture	19	2	16	35	51
javaparser	181	11	18	33	51
CoreNLP	615	3	23	18	41
jacop	89	1	6	5	11
jHiccup	0.7	1	2	1	3

DD: number of unique direct dependencies; ID: number of unique indirect dependencies; kLOC: number of thousands of lines of application code; Maven modules: number of Maven modules; Total: total number of unique dependencies. Rows are ordered with respect to the total number of dependencies in the projects.

In Maven, a dependency is identified by a name and a version number. The name is a combination of its `groupId` and `artifactId`, separated by a colon, for example, `com.google.guava:guava`. We considered two dependencies identical if their name and version match precisely. As shown in step 4 in Figure 1, we compared the accuracy of SBOMs by computing the precision and recall of dependency lists computed by each producer. The *precision* is the share of dependencies in the SBOM that are correct with respect to the ground truth. The *recall* is the share of correct dependencies that are in the SBOM.

Note that the ground truth considers all dependencies required for producing a software artifact,

including test dependencies. While these test dependencies are not included in the deployed software, they are relevant in the context of supply chain attacks. A malicious test dependency has the potential to interact with the build system and introduce malicious code at build time.³ To trace vulnerable or malicious test dependencies, it is important that these are included in the SBOM.

The last step of our methodology, step 5 in Figure 1, consists of manually analyzing a sample of SBOMs to get a concrete grasp of the content of the SBOMs produced. This provides us with detailed insights about the challenges that SBOM producers face to correctly

retrieve all of the dependencies in an application's software supply chain.

Experimental Results

We followed our protocol and ran six SBOM production tools on 26 Java projects. The results provide key insights about the tools' behavior as well as the quality of the produced SBOMs.

Producer Insights

Table 1 summarizes the essential features of SBOM producers that we have identified, and to what extent these features are present in the tools.

Checksum diversity. Table 1 summarizes the number of different checksum algorithms that each SBOM producer uses. The production of different checksums is useful because it maximizes the likelihood of integrating the SBOMs with third-party tools that expect a specific checksum. Three producers compute eight types of checksums for each dependency jar: CycloneDX-Generator, CycloneDX-Maven-Plugin, and Depscan provide md5, sha1, sha256, sha512, sha384, sha3-384, sha3-256, and sha3-512 for each dependency in the SBOM. One producer, OpenRewrite, does not provide any checksums, which is considered a serious limitation. Our observations help practitioners to select SBOM producers accordingly.

Dependency hierarchy. An essential feature of SBOM producers is eliciting all of the dependencies in the software supply chain of an application. Beyond a flat list, some analyses—such as vulnerability analysis, debloating, and installation via package managers—require the complete tree of relationships among the different components in the chain. The CycloneDX specification provides the attribute `dependencies` to serve this purpose. We note that five of six producers report the hierarchy among dependencies. However, `jbom` cannot link the dependencies together since it acts after the build step where some dependencies cannot be resolved. For example, for `mybatis-3`, `com.fasterxml.jackson.core:jackson-core` version 2.13.2 is an indirect dependency at the fourth level. The producers `Build-Info-Go`, `CycloneDX-Generator`, `CycloneDX-Maven-Plugin`, and `Depscan` report this information correctly.

Reproducibility. SBOMs are meant to be reference documents, and potentially may become legally binding. To that extent, one must produce them reliably. In that respect, we claim that SBOM production should be reproducible. We say an SBOM producer is

reproducible if it generates strictly identical files contentwise over multiple runs. We excluded metadata, such as timestamp. We generated SBOMs twice for each producer and found that `Build-Info-Go` and `jbom` are not reproducible: they do not preserve the order of SBOM elements. Moreover, `jbom` also produces different hashes of the components. While this is a fixable engineering issue, it highlights the necessity to consolidate the maturity of SBOM tooling before it can be relied upon in court.

Production step. There are six steps at which an SBOM could be produced: design, source, build, analyzed, deployed, and runtime (<https://www.cisa.gov/sites/default/files/2023-04/sbom-types-document-508c.pdf>). The considered SBOM producers do not produce SBOM at the same step. We report the step at which SBOM is produced per the documentation provided by the developers. `Build-Info-Go`, `CycloneDX-Generator`, `CycloneDX-Maven-Plugin`, and `OpenRewrite` produce an SBOM at the build step. The build step can further be broken down into more steps, as Maven splits a build into multiple phases (<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>). `Build-Info-Go` produces an SBOM when the Maven build system is compiling. Meanwhile, the other three producers perform SBOM production when the artifact, JAR, for example, is being generated. This phase is also called package in Maven. `Depscan` produces an SBOM from the source files. Finally, `jbom` produces an SBOM by analyzing the final jar file, corresponding to the Cybersecurity and Infrastructure Security Agency (CISA) "Analyzed" step.

These different steps are significant regarding the production of SBOMs, since the information available about the software supply chain varies at these different stages. Indeed, software projects go through a build/continuous integration/continuous deployment life cycle and, at every point, the information available is different.¹³ For example, before the build phase, an SBOM producer cannot know what will be finally included in the binary. Similarly, after the build, information about some dependencies may be lost because the build system has removed redundant or unnecessary dependencies.

The CycloneDX standard does not address this aspect, and the producers do not clearly document or motivate the phase they consider. SBOM producers should state the production step at which they collect information about the software supply chain to help SBOM consumers decide which SBOM is most appropriate for their needs.

Scopes. The CycloneDX JSON specification supports an optional scope attribute for each component. This attribute can take the values *required*, *optional*, or *excluded*, based on the dependency's behavior at runtime. According to the specification (<https://github.com/CycloneDX/specification/blob/1.4/schema/bom-1.4.xsd#L514>), the *required* scope denotes that the component is required at runtime; the *optional* scope denotes components that “[...] are not capable of being called due to them not being installed or otherwise accessible by any means.” Finally, *excluded* components “[...] provide the ability to document component usage for test and other nonruntime purposes.”

We observed significant differences among SBOM producers regarding the identification of scopes. As an example, `org.slf4j:slf4j-api@2.0.1` is a dependency of `mybatis-3`. CycloneDX-Generator, CycloneDX-Maven-Plugin, and Depscan report its scope as *optional*, OpenRewrite reports the scope as *required*, while the other producers report no scope at all.

We note that each SBOM producer only uses a subset of the allowed scope values. CycloneDX-Generator, CycloneDX-Maven-Plugin, and Depscan either label components as *optional* or provide no scope value. `jbom` labels all components as *required*. OpenRewrite marks components as either *optional* or *required*, and Build-Info-Go does not report scope for any component. It is not clear from the documentation of the producers how these values are computed. Due to the lack of clarity in the standard and the absence of ground truth, it is impossible to determine which one is correct.

Providing clear information as to how and when in the software lifecycle a component is used—the scope as we understand the standard—is an important feature of an SBOM. However, our results show that no SBOM consumer can rely on the scope values produced by current SBOM producers.

Dependency Identification Accuracy

Figure 2 shows the accuracy of the considered SBOM producers per our ground truth. The x axis is the precision and the y axis is the recall for each producer. A point \bullet represents the accuracy of dependencies captured in the SBOM and the isolines represent the standard F1-score combining precision and recall. We report the average precision and recall of an SBOM producer, over all projects. For our experiment, we performed 156 executions of SBOM producers, which produced 119 SBOMs and 37 failures. For the latter, SBOMs were either empty or contained no dependency because the build failed or the producer failed. We excluded these data points from our study.

At the bottom left of Figure 2, `jbom` has the lowest precision and recall. Next, OpenRewrite has the highest precision of 96%, but with a low recall. Higher up in the figure, we find CycloneDX-Maven-Plugin with 92% precision and 66% recall; CycloneDX-Generator and Depscan perform very similarly. Finally, Build-Info-Go is at the top right corner with the best score of detected dependencies according to the dataset and ground truth.

We highlight five main reasons why producers fall short on creating a fully-accurate SBOM with respect to the ground truth: exclusion of test dependencies from the SBOM; failure to resolve maven properties (<https://maven.apache.org/pom.html#Properties>); failure to correctly resolve the version of a dependency; advanced dependency resolution techniques; and the project itself is counted as a dependency. We elaborate on each of these points below.

SBOM producers like OpenRewrite and CycloneDX-Maven-Plugin do not include test dependencies by default in the SBOM they produce. This explains the low recall of 39% and 66%, respectively. Although Build-Info-Go has the highest F1-score, we observe that it misses test dependencies for some projects, while achieving 100% recall on some other projects, for example `jenkins`, which clearly contains test dependencies.

When a producer does not correctly resolve Maven properties, the SBOM cannot be compared to the ground truth. For example, `jbom` reports version `{guava.version}` for `com.google.guava:guava`, instead of `31.0.1-jre`, for `alluxio`. This eventually yields a list of dependencies that are not comparable with the list of dependencies in the ground truth.

To verify that a dependency is correctly reported, the `groupId`, `artifactId`, and `version` must match. However, `jbom` incorrectly retrieves the version for some dependencies. For example, it reports version `0.4` of `com.pholser:junit-quickcheck-core` for `CoreNLP`, which does not exist in the ground truth. OpenRewrite faces similar challenges, reporting version `4.1.78.Final` for `io.netty:netty-handler` in `selenese-runner-java`, while the correct version is `4.1.79.Final`. A major difficulty for retrieving the version number occurs when different versions of the same library appear as indirect dependencies at different locations in the dependency tree. The correct version identification must faithfully capture the actual resolution embedded in the build system.

Moreover, the resolution of dependencies is affected by the different ways SBOM producers use to retrieve dependencies. Depscan and CycloneDX-Generator perform equally on most projects. For example, they both have the same results on `selenese-runner-java`. However, Depscan correctly reports a dependency `ch.uzh.ifi.seal:changedistiller` version

0.0.4 for steady while CycloneDX-Generator misses it. In this case, Depscan reports a dependency that is stored as local jar in the project. This illustrates that they both have different methods for resolving dependencies. On the other hand, a closer look at the architecture of Build-Info-Go shows that it relies on the Maven application programming interfaces to invoke the build and retrieve deeper information, thus producing results that are closer to the ground truth. This suggests that SBOM producers benefit from being tailored to a language and a build system to plug deeply into the build process to obtain correct information.

Finally, we have observed that some SBOMs include the source project in the dependency list. This is yet another reason why Build-Info-Go falls short of perfect alignment with the ground truth. It reports all dependencies for `selenese-runner-java`, `accumulo`, `jenkins`, `checkstyle`, `error-prone`, `jooby`, `launch4j-maven-plugin`, `orika`, and `mybatis-3`, but in each of these cases, it incorrectly considers the root module as a dependency.

Overall, Figure 2 shows significant differences among the accuracy of the SBOMs produced by six state-of-the-art producers. These results reveal discrepancies in the list of dependencies in the SBOMs, with different dependency versions and missed dependencies. To better illustrate the different accuracy levels, we manually analyzed a sample of the SBOMs. To sample the files, we used the following criteria: We selected SBOMs produced by Build-Info-Go and `jbom` as these producers are at both ends of the accuracy range; we analyzed SBOMs for project `spoon`, as three of the authors are maintainers and hence have a deep understanding of this project. After applying the previous filters, we sampled four SBOM files: two SBOMs with the highest and lowest precision on dependencies, produced by Build-Info-Go and `jbom`, and two SBOMs with the highest and lowest precision on direct dependencies produced. This analysis was conducted by two of the authors, both experts in Java programming. In the case of discrepancies, they met and discussed to resolve them and reach a conclusion.

The ground truth indicates that the single module of `spoon` has 22 direct dependencies and 32 indirect ones (see Table 2). The SBOM produced by Build-Info-Go correctly contains 23 dependencies, and the only incorrect one is the `fr.inria.gforge.spoon:spoon-core` itself. The precision is consequently high, but some dependencies are clearly missing. Build-Info-Go excludes test dependencies for `spoon`. On the other hand, the SBOM produced by `jbom` reports 125 dependencies, but only 29 of them are correct. The other 96 dependencies are the result of failure of `jbom` to resolve Maven properties, versions, or metadata `groupId`.

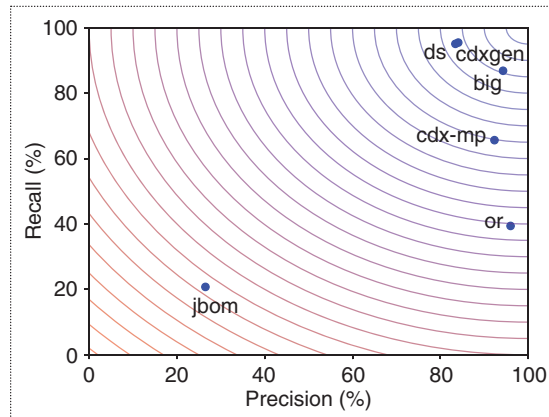


Figure 2. Mean precision and mean recall of each SBOM producer, excluding producer failures. big: Build-Info-Go; cdxgen: CycloneDX-Generator; cdx-mp: CycloneDX-Maven-Plugin; ds: Depscan; jbom: jbom; or: OpenRewrite

The next two case studies come from Build-Info-Go. First, we inspected the SBOM produced for `selenese-runner-java`, and we found that Build-Info-Go fetches all 136 dependencies. It also includes the complete dependency tree hierarchy information. Such precise information is important and makes the SBOM consumable. However, we noticed that even a solid producer such as Build-Info-Go does not always achieve high precision. For example, the SBOM of `javaparser` includes 14 correct dependencies of 51. The majority of the dependencies the producer misses are test-scoped. We observed an inconsistent behavior in Build-Info-Go as it sporadically includes test dependencies.

The SBOM produced by `jbom` for `async-http-client` contains only two correct dependencies of 109. On a deeper inspection, we observed that most dependencies in the SBOM are identified with wrong versions, resulting in poor precision. We analyzed the SBOM of `mybatis-3` produced by `jbom`. This SBOM includes all the direct dependencies, precisely with correct version numbers as they were specified. However, all indirect dependencies are missed.

Overlap analysis. Figure 3 is a Venn diagram that captures the overlap between the SBOMs of CoreNLP generated by the six SBOM producers. For each SBOM producer, we used the set of true positives dependencies. Intersection areas mean producers have the same correct dependency in their SBOM. Every SBOM producer has a different color for their outline. For example, we use yellow for CycloneDX-Generator. The labels indicate the number of dependencies in the intersection area

and areas without a label are empty (meaning no dependency in common). We have six different intersection areas. The largest one is in the middle and shows that 20 dependencies are correctly identified by every producer. The second-largest area indicates that 12 dependencies are correctly captured by every producer except `jbom`. For example, `jbom` misses `javax.xml.bind:jaxb-api` because it either resolves an incorrect version, or it resolves some dependencies as `null`. Two areas have only one dependency in the intersection. One intersection area contains the producers `CycloneDX-Generator`, `CycloneDX-Maven-Plugin`, `jbom`, and `Depscan`, which correctly capture `junit:junit:4.13.1`, while `OpenRewrite` and `Build-Info-Go` miss it. `OpenRewrite` entirely skips test dependencies by design, and `Build-Info-Go` misses it. The other area is the intersection of `CycloneDX-Maven-Plugin`, `CycloneDX-Generator`, and `Depscan` that correctly detect `org.hamcrest:hamcrest-core:1.3` in the SBOM. `jbom` misses this dependency because it is included as a jar with a relative path in the repository. It only identifies correctly the `groupId` and `artifactId`, while the version is set to null.

Experimental Limitation

It may be argued that SBOM producers should simply reuse the ground truth we consider as the basis for SBOM production, that is the code of Maven in our experiment. However, SBOMs can be extracted at

multiple steps, per our discussion on production steps above. All of these extraction steps are valid and potentially useful depending on the goal and the SBOM consumption. Our ground truth only captures one single production step. To that extent, some inaccuracies we have reported may be due to the mismatch between the ground truth and the targeted production steps of some SBOM producers.

Take-Aways

In theory, extracting SBOMs is easy. Our results show that in practice, this is not the case. In this section, we discuss the benefits of our work for two target audiences, Java developers and standardization committees, and reason about the difficulty of confronting theory and practice.

Java developers. Our in-depth study shows that `Build-Info-Go` is the best SBOM producer for Java developers. The reasons are that: 1. it produces different checksums; 2. it supports dependency hierarchies; and 3. it achieves the highest precision and recall thanks to a tailored integration in Maven. Yet, `Build-Info-Go` has room for improvement. First, the precision and recall of 94% and 87%, respectively, can be increased, with several important fixes. Second, assuming that the standard clarifies the matter, it could also provide the scope of the dependencies.

Standardization committees. Our study identifies two shortcomings in the CycloneDX standard. The specification needs to require producers to specify the exact step at which the SBOM is produced, and it must precisely define the notion of “scope,” which would help both SBOM producers and consumers. We believe that the latter is more important as the current state is ambiguous for developers, and ambiguity upstream typically means incorrectness downstream.

Difficulty. Our study reveals difficulties of different nature in producing complete and useful SBOMs. The challenges of checksums, tree hierarchy, and determinism can all be fixed with additional engineering effort. However, clarifying the meaning of production steps and scopes is fundamentally hard because it requires the appropriate abstraction over multiple build pipelines in different software stacks, and this abstraction would require consensus in the SBOM community.

Open Challenges

Our experiments revealed a number of challenges for the accurate production and the effective consumption of SBOMs.

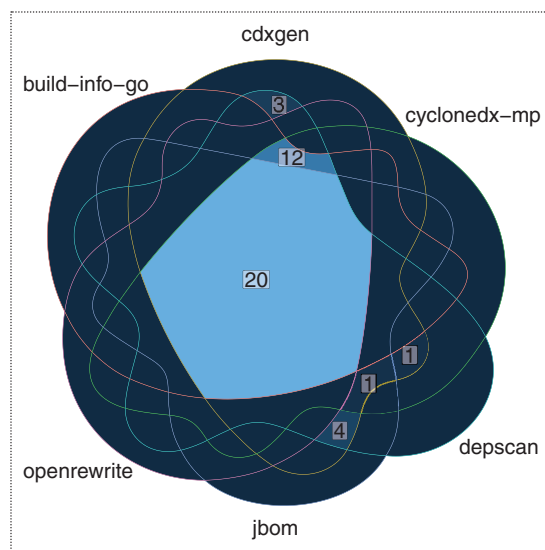


Figure 3. Venn diagram of different SBOM producer results. Only the true positives (correctly identified dependencies) are compared. Intersection areas mean that multiple SBOM producers have overlapping correct dependencies. In this project, all producers correctly identify a majority of 20 dependencies.

SBOM and Tooling Dependencies

In our analysis, we observe that the bulk of SBOMs consists of collecting accurate dependency trees for an application project. Yet, the software supply chain of an application is made of many more components. For example, the version control system, the testing and build tools, and the infrastructure to deploy or distribute the application are key components of the supply chain. In recent years we have witnessed attacks, such as the Solarwinds incident, which successfully compromised a system through these components.³ The CycloneDX standard attempts to document such information by providing the attribute `externalReferences`. However, there is currently scarce support to generate these attributes and our study shows that the SBOM producers implement this partially and with inconsistencies. The comprehensive collection and documentation of all tools involved in the supply chain is a pressing challenge to produce SBOMs that are amenable to thorough hardening procedures.

SBOMs for Threat Analysis

In the longer term, the value of SBOMs will increase with enabling automatic security analyses. For example, one key challenge is to let SBOM producers qualify the trust that one can have in the dependencies. This type of assessment of the supply chain relates to threat modeling and analysis, which is already considered good practice for DevOps organizations.¹⁴ To guide which properties an SBOM should include to support reasoning about trust and threats, the attack taxonomy of Ladisa et al.³ constitutes an excellent starting point. Furthermore, the work of Zahan et al.¹⁵ proposes concrete metrics as warning signs of supply chain vulnerabilities that could be mapped to the taxonomy, such as “too many maintainers,” which can match the “take over legitimate account” as well as the use of installation scripts, which relates to the “running a malicious build job” technique.

SBOMs at Runtime

The next challenge will be to bring SBOMs online, as a foundation to enforce security requirements at runtime. For a given SBOM pertaining to a software application, one can develop lightweight dynamic analysis to enforce mandatory access control policies. This can be achieved by monitoring the usage of dependencies at runtime and ensuring that only the dependencies within the SBOM are used by the application, thus preventing the entire class of vulnerabilities that rely on the dynamic inclusion of malicious code and packages. A major challenge for such an approach is that it would require accurate static information about dependencies, which is a challenging endeavor, as we have shown in this article.

SBOMs in Other Software Stacks

The production of SBOMs for other software stacks is likely to face similar challenges as those seen for Java. We note that some programming ecosystems already partially address certain challenges. For instance, ecosystems, such as npm, Go, and Rust record checksums for all publicly available dependencies in autogenerated lock files. In theory, the data provided by these instruments can already be aggregated and used to produce meaningful SBOMs. In the specific case of Go, the lock file information can be validated against an immutable, verifiable database, providing integrity guarantees that can be leveraged in SBOMs. Nonetheless, a definitive solution is yet to be established and widely used in either of these software stacks.

We performed a deep dive into the meaning of SBOMs and its realization in the Java ecosystem, one of the most commonly used enterprise programming languages. Our research findings indicate strong interest and vibrant activity in this essential area for software supply chain security and reliability. Yet, we also revealed that SBOMs today rely on a technical foundation that is unstable. Our empirical insights shed light on important weaknesses that require attention, starting with incorrect or incomplete dependency lists recovered in SBOMs. These findings call for further work in clarifying the SBOM standards, as well for more work on improving the quality of SBOM producers' output. Studying SBOM quality for other languages (e.g., Rust) and for other SBOM formats (e.g., SPDX) would be very valuable for the community. Both academia and industry agree that SBOMs promise great benefits; now the time is ripe to all work together to unleash their full potential. ■

Acknowledgment

This work was supported by the Consistent Hardening and Analysis of Software Supply Chains project funded by the Swedish Foundation for Strategic Research, the WebInspector project funded by the Swedish Research Council, as well as by the Wallenberg Autonomous Systems and Software Program funded by the Knut and Alice Wallenberg Foundation.

References

1. R. Cox, “Surviving software dependencies,” *Commun. ACM*, vol. 62, no. 9, pp. 36–43, 2019, doi: 10.1145/3347446.
2. A. Gkortzis, D. Feitosa, and D. Spinellis, “Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities,” *J. Syst. Softw.*, vol. 172, Feb. 2021, Art. no. 110653, doi: 10.1016/j.jss.2020.110653.

3. P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of attacks on open-source software supply chains," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 1509–1526, doi: 10.1109/SP46215.2023.10179304.
 4. C. Rezk, Y. Kamei, and S. Mcintosh, "The ghost commit problem when identifying fix-inducing changes: An empirical study of apache projects," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3297–3309, Sep. 2022, doi: 10.1109/TSE.2021.3087419.
 5. N. Harutyunyan, "Managing your open source supply chain-why and how?" *Computer*, vol. 53, no. 6, pp. 77–81, Jun. 2020, doi: 10.1109/MC.2020.2983530.
 6. K. Nikitin et al., "CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1271–1287.
 7. L. Tal. "The Log4j vulnerability and its impact on software supply chain security." Snyk. Accessed: Mar. 17, 2023. [Online]. Available: <https://snyk.io/blog/log4j-vulnerability-software-supply-chain-security-log4shell/>
 8. "Survey of existing SBOM formats and standards," United States Department of Commerce–National Telecommunications and Information Administration, Washington, DC, USA, 2021. Accessed: Mar. 17, 2023. [Online]. Available: https://www.ntia.gov/files/ntia/publications/sbom_formats_survey-version-2021.pdf
 9. C. Soto-Valero, M. Monperrus, and B. Baudry, "The multibillion dollar software supply chain of Ethereum," *Computer*, vol. 55, no. 10, pp. 26–34, Oct. 2022, doi: 10.1109/MC.2022.3175542.
 10. F. Massacci and I. Pashchenko, "Technical leverage: Dependencies are a mixed blessing," *IEEE Security Privacy*, vol. 19, no. 3, pp. 58–62, May/June. 2021, doi: 10.1109/MSEC.2021.3065627.
 11. "The minimum elements for a software bill of materials," United States Department of Commerce – National Telecommunications and Information Administration, Washington, DC, USA, 2021. Accessed: Mar. 17, 2023. [Online]. Available: <https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom>
 12. C. Soto-Valero, N. Harrant, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Softw. Eng.*, vol. 26, no. 3, pp. 1–44, May 2021, doi: 10.1007/s10664-020-09914-8.
 13. B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2630–2642, doi: 10.1109/ICSE48619.2023.00219.
 14. S. Rafi, W. Yu, M. A. Akbar, S. Mahmood, A. Alsanad, and A. Gumaei, "Readiness model for devops implementation in software organizations," *J. Softw., Evol. Process*, vol. 33, no. 4, Apr. 2021, Art. no. e2323, doi: 10.1002/smr.2323.
 15. N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proc. 44th Int. Conf. Softw. Eng., Softw. Eng. Pract. (ICSE-SEIP)*, 2022, pp. 331–340, doi: 10.1145/3510457.3513044.
-
- Musard Balliu** is an associate professor at the School of Electrical Engineering and Computer Science at KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include computer security, programming languages, formal methods, and software engineering. Balliu received a Ph.D. in computer science from KTH Royal Institute of Technology, Sweden. Contact him at musard@kth.se.
-
- Benoit Baudry** is a professor of software technology at KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include software engineering, focusing on software testing and automatic diversification. Baudry received a Ph.D. from the University of Rennes. Contact him at baudry@kth.se.
-
- Sofia Bobadilla** is a research engineer at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. Her research interests include bots in software development, software reliability, data science, and multimedia information retrieval. Bobadilla received a B.A. in computer science from the University of Chile, Santiago, Chile. Contact her at sofbob@kth.se.
-
- Mathias Ekstedt** is a professor of industrial information and control systems at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include the intersection of cybersecurity and software systems architecture modeling and analysis. Ekstedt received a Ph.D. from the KTH Royal Institute of Technology. He is the cofounder and director of KTH's Master program in cybersecurity. Contact him at mekstedt@kth.se.
-
- Martin Monperrus** is a professor of software technology at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include software engineering, with a focus on software reliability. Monperrus received a Ph.D. from the University of Rennes, France. He is a Member of IEEE. Contact him at monperrus@kth.se.
-
- Javier Ron** is a Ph.D. student at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include software dependability and distributed systems. Ron received an M.Sc. from the KTH Royal Institute of Technology, Sweden. Contact him at javierro@kth.se.

Aman Sharma is a Ph.D. student at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include exploring techniques to secure software supply chains. Sharma received a Bachelor of Technology from the Indian Institute of Technology, Roorkee, India. Contact him at himatamansha@kth.se.

Gabriel Skoglund is an M.A. student at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include computer security and cryptography. Skoglund received a B.Sc. in computer science from the KTH Royal Institute of Technology, Sweden. Contact him at gabsko@kth.se.

César Soto-Valero is a software engineer working in the financial sector in Stockholm, Sweden. His research interests include code quality, code evolution, and blockchain technologies. Soto-Valero received a Ph.D. from the KTH Royal Institute of Technology. Contact him at cesarsotovalero@gmail.com.

Martin Wittlinger is a research engineer at the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden. His research interests include Java code analysis, transformation, and build tools. Wittlinger received an M.Sc. in computer science from Karlsruhe Institute of Technology, Karlsruhe, Germany. Contact him at marwit@kth.se.