University of Applied Sciences Munich
Faculty of Computer Science and Mathematics

Masterthesis
for Obtaining the Academic Degree of
Master of Science
in Course of Studies IT Security

Quality Assessment of SBOM Generation Tools and Standards on Open Source Projects

---

Author:	Marius Biebel
Matriculation number:	
Submission Date:	26. April 2024
Supervisor:	Prof. Dr. Peter Trapp
Advisor:	Prof. Dr. Erik Krempel

1 Acknowledgements

Many thanks go to my professors, Peter Trapp and Erik Krempel, for their support and guidance. They gave me the freedom to explore this topic and provided me with valuable feedback. Especially from a scientific perspective. And also with the advice to not get lost in the details.

Also, I would like to thank the CycloneDX and Software Package Data Exchange (SPDX) community for their work and for letting me join their working groups and sharing insights into the background of their work. I especially want to thank the OpenSSF Software Bill of Materials (SBOM) Everywhere working group for their work and the discussions we had. It was a treat to work with you, and I hope we can continue our work in the future.

I was also backed by my employer, the German Patent and Trademark Office, which allowed me to work part-time and study IT security.

Last but not least, I want to thank my parents and friends for their support. Without them, I would not have been able to finish this thesis.

2 Abstract

With the increasing complexity of modern software composition and an ever-growing software supply chain, where numerous resources are sourced from open-source projects, the need to keep track of these resources has arisen. Following the Log4j incident [29], the American Government passed Executive Order (EO) 14028, mandating a SBOM for all software products sold to federal government agencies. Similarly, the European Union passed the Cyber Resiliance Act (CRA), which requires an SBOM for all digital products in the European market. For these reasons, machine-readable SBOM formats have emerged in recent years, implemented by a wide variety of projects that produce and consume such SBOMs.

This thesis investigates a collection of projects that generate SBOMs at various stages of the software development lifecycle. Each generator is applied to open-source projects to produce SBOMs. This study examines and compares the features provided by these SBOMs. Additionally, it assesses the completeness of the enumerated packages/components by analyzing the overlap among the SBOMs generated for each project.

The thesis highlights the distinctions between various tools and phases, highlighting potential bugs in the implementation of the tools investigated. It elucidates the variances in the SBOMs generated at different phases of the software development lifecycle. An analysis was conducted to identify which components of the CycloneDX and SPDX schemas were enriched with data during the SBOMs generation process. Furthermore, the research reveals that the tooling examined produces results of varying quality and depth. A metric was introduced to quantify the overlap among the diverse SBOMs, yielding mixed results.

It is concluded that the quality and applicability of a produced SBOM can vary drastically depending on the use case. This variation is partly attributable to the different methodologies implemented by the investigated tools but also partly based on divergent results in the quality or depth of the generated SBOMs, where identifiers are produced in different ways or values are not sufficiently enriched.

The thesis aims to propose initial methods for validating the enrichment of a SBOM and assessing its completeness. This was done by testing the implemented generators on real-world projects.

Contents

3 Introduction

The availability of freely accessible open source software has become one of the fundamental driving forces behind innovation in software development. Nowadays, a vast majority of systems are built on some form of open source software. This may include entire application frameworks, such as the Java Jakarta framework or the Spring framework. It also applies to application servers like Nginx or Apache servers that offer to host a self-developed software. And this fact also reaches down to the basic OS layer. Prominently, the Linux Kernel and lots of operating systems based on the Linux Kernel are open source and widely used. It can be said that open source is an inevitable building block for the majority of services used in our modern day lifes.

While open source software offers numerous advantages, it also presents a potential attack surface for adversaries. The supply chain for a product based on open-source software becomes increasingly complex, particularly when the system is based on a mixed ecosystem where several different programming languages and platforms work together to compose a project. Nowadays, it is common for larger projects to be composed of several technologies, such as C or C++ for performance or driver implementations, Java or Python for actual system logic, and a frontend system like a PHP application server or a JavaScript framework like React, Vue.js, or Angular. Additionally, bigger projects are often composed for a wide variety of platforms, such as Windows and Linux, or different architectures like 32 or 64-bit systems, or x86 or ARM processor architecture. Moreover, there are different virtualization systems like Docker, Kubernetes, or Clouds that get served with their customized builds. All of this complexity makes it challenging to keep track of versions and flavors of the affected products.

Given this complexity, there is a clear need to keep track of the increasingly complex software composition, especially as more and more vulnerabilities are found in open source projects. The Log4j vulnerability is a case in point. This vulnerability was based on a simple expression language tag, which would be interpreted by the logging library if such a statement was logged. An attacker could use this to load code from a remote source via an JNDI injection and execute arbiter code from a remote source via URL. Log4j is a widely used library in the Java ecosystem. Finding and replacing this vulnerable library in all affected products was a prime example of the need to track the used products and their dependencies in an infrastructure. [29]

The American government's EO 14028 has further catalyzed this debate [30]. In the EO, the White House addressed the need for enhancing software supply chain security. To achieve this, software suppliers for American federal government agencies will be required to provide a SBOM for their products. This will enable software users to keep track of all the components contained in a product and ensure that they are up-to-date or if there are recently discovered vulnerabilities in a software product. In addition, the National Institute of Standards and Technology (NIST), National Telecommunications and Information Administration (NTIA), and Cybersecurity and Infrastructure Security

Agency (CISA) have published requirements and guidance for the generation and use of an SBOMs [36, 43, 44].

Additionally, the European Union introduced the need for SBOMs as part of the CRA. The CRA focuses on manufacturers and retailers of products that incorporate digital elements, such as hardware, software, and IoT devices. Under the CRA, manufacturers are required to identify and document vulnerabilities and components contained in their products in a machine-readable format. The European Union (EU), with the CRA, implements strategic legislation aimed at enhancing transparency and accountability in the manufacturing and distribution of digital products. This effort is to protect consumers and companies in the European market. [11, 12]

While software vendors are motivated to comply with government regulations to secure their government contracts, open-source projects have no incentive to comply with such compliance regulations. As a result, the vast majority of open-source projects do not supply their builds with an SBOM. To address this issue, the OpenSSF initiated the SBOM-Everywhere project as part of its Open Source Software Security Mobilization Plan. The goal of this working group is to focus on tools and advocacy efforts to remove barriers for the generation and consumption of SBOMs in open-source projects. [13, 40]

Although there are already a lot of tools and standards available for producing SBOMs, three different standards are emerging for representing SBOMs in a machine-readable format.

The first widely known standard is the SPDX, which is standardized in ISO/IEC 5962:2021 [2] and maintained by the SPDX working group under the umbrella of the Linux Foundation.

The second widely used standard is the CycloneDX format, which is maintained by the Open Worldwide Application Security Project (OWASP) foundation [15].

The third related standard is the Software Identifier (SWID), which is standardized in ISO 19770-2:2015 [1] and focuses on the lifecycle management of software to enable effective asset management in large infrastructures. Although SPDX and CycloneDX were designed with the generation of SBOMs in mind, the SWID standard could achieve the requirements of an SBOM, but it was not designed with the requirements of an SBOM in mind. At the time of writing, there is a lack of proper tooling to generate and consume SWID as an SBOM efficiently.

3.1 Motivation

Although the importance of SBOMs for asset management regarding security and licenses as a risk management measure is clear, the technology is still relatively new. Thus, the quality of produced SBOMs varies drastically. While there are tools such as the National Telecommunications and Information Administration (NTIA) Conformance Checker [35], the SBOM Scorecard Project [31] and the SBOMQS Quality metrics [32], these tools only check an SBOM for formal correctness, validating if specific fields are initialized. This is done with the background of the minimum elements for an SBOM, which were set by the NTIA on behalf of the Whitehouse EO [36]. However, they cannot check the SBOM on a

content level. Therefore, an automatically generated SBOM for a complex project structure might not detect dependencies or miss out on aspects of the project's overall complexity. Additionally, while some of the required fields of such quality metrics are meaningful for SBOMs related to commercially supplied and supported software by software vendors, applying such fields to open-source projects might be challenging.

3.2 Related Work

The necessity of SBOMs for tracking the usage of open-source projects is highlighted by Hatta in his paper [27]. He elucidates that numerous vital and foundational open-source projects are often developed and maintained by hobbyists, implying that professional-level development practices cannot always be expected. Hatta states that while Linus's Law predicts that bugs in code will vanish if just enough eyeballs look at it, the amount of eyeballs needs to be secured to ensure this advantage of open source.

However, vulnerabilities in open-source supply chains are not limited to mere bugs with security ramifications. There also exists the threat of attackers attempting to compromise a project to inject malware into downstream applications. Ohm et al. [37] demonstrate 174 instances of attacks on open-source software during the period from 2015 to 2019. Their findings indicate that attackers exploit package repositories as an effective and scalable platform for malware distribution.

While the idea of increasing SBOM generation in open-source projects is commendable, its implementation poses significant challenges. In 2020, the Linux Foundation conducted a contributors survey, which did not include SBOMs specifically. Nonetheless, the survey's overall statistics revealed that the adoption of security policies and the overall security posture is often not prioritized in an open-source project. [39]

In January 2022, the Linux Foundation conducted a survey to assess the adoption and organizational readiness for SBOMs. This survey included 412 organizations worldwide. It highlighted SBOMs, globally unique software identifiers, and component verification via reproducible builds as critical elements in securing the software supply chain. The survey revealed that over 90% of participants had initiated work on SBOMs. While a high commitment to producing SBOMs was observed, numerous questions remain about the value SBOMs offer to consumers, their production and consumption methods, and the applicable standards [28].

Xia et al. conducted a survey in February 2023 on the present adoption of SBOMs and the challenges that lie ahead. Their findings indicated that the quality of SBOMs remains a concern, along with the lack of proper tooling to consume and share SBOMs.[49]

A blog article from March 2023 by the OpenSSF provides guidelines for crafting high-quality SBOMs [33]. It discusses methods for assessing SBOMs quality and emphasizes that most SBOMs fail to meet the minimum standards set by the NTIA. Their analysis of 3000 SBOMs found that less than 1% complied with these standards. The article also highlights the limited availability of SBOMs for research, citing the SBOM-shelter project on Github as a notable source. Managed by ChainGuard, this project features 52 real-world SBOMs and over 3000 laboratory-generated SBOMs from container images [7].

Critics argue that simply increasing the quantity of SBOMs is insufficient and does not necessarily improve the overall quality of SBOMs. John Meyers of ChainGuard published two articles [3, 4] on this issue, describing it as a chicken-and-egg problem in relation to the generation and consumption of SBOMs. While the initial problem of generating the first SBOMs in open-source projects seems to have been addressed, a new issue has arisen. Meyers refers to it as "SBOM chickens laying bad SBOM eggs" [4].

Shortly before submitting this thesis, Zhao et al. published a paper evaluating Software Composition Analysis (SCA) tools in Java. It introduces an Evaluation Model for assessing SCA tools, considering their dependency-resolving capabilities and effectiveness. The study involves qualitative and quantitative evaluations, examining six SCA tools across different aspects like tool capabilities, dependency detection accuracy, and vulnerability accuracy. Key findings include the varying support levels for dependency management features and the need for improvements in SCA tools. [50]

3.3 Research Goals

The conducted research in this field leaves a gap in terms of evaluating the quality of the generated SBOMs regarding the completeness of SBOMs. Therefore, this paper introduces primary research on a comparison of the results of different tools for the generation of SBOMs in different stages of the software lifecycle regarding the completeness of the detected dependencies.

While there is some research on the quality of SBOMs, it mostly focuses on compliance with the schematics of the defined data structure or the minimum elements defined by regulatory authorities. However, at the time of writing, there is no research that examines the quality of generated SBOMs measured at the real complexity of included dependencies.

4 SBOM Standards

The necessity for a SBOM standard originates from the need to exchange information generated by SCA. The increasing integration of open-source software within the software development lifecycle has highlighted the importance of tracking implemented software components. This need has become more critical with the widespread use of package managers such as Java's Maven, JavaScript's NPM, and Python's Pip. However, mere scanning was insufficient. The outcomes of SCA required documentation in a standardized format, ensuring ease of exchange and suitability for automated processing. Consequently, several standards were proposed, with the following three gaining recognition by the American government due to EO 14028 [30], by the NTIA [36].

This chapter provides an exploration of SBOM standards. The different sections in of the SPDX and CycloneDX standard will be introduced and there use cases are presented. Also a brief introduction to the SWID standard is provided.

4.1 Software Package Data Exchange

SPDX is a standard initially specified in February 2010 by the FOSSBazaar working group under the Linux Foundation. It has since evolved into an independent project, yet remains under the umbrella of the Linux Foundation. In 2021, SPDX was formalized as an ISO standard, designated ISO/IEC 5962:2021 [2, 46].

SPDX supports a diverse array of data formats, including JSON, XML, YAML, tag:value, and RDF. These file formats are interoperable, allowing for seamless conversion between them. Additionally, SPDX provides tools for the validation and conversion of different file formats [45].

The current iteration of the SPDX standard, version 2.3, comprises several components. The information provided herein pertains to the publicly available SPDX specification, version 2.3 [46].

- SPDX Document Creation Information
- Package Information
- File Information
- Snipped Information
- Other Licensing Information Detected
- Relationship between SPDX Elements
- Annotation Information
- Review Information

SPDX Document Creation Information

The Document Creation Information is the only mandatory section in the SPDX specification. It encompasses details about the document, such as the SPDX version number, the creator, the creation date, external and internal document identifiers, document name,

and namespace. Additionally, it includes fields for creator and document comments, as well as the data license field. This field specifies the license of the SBOM document itself. By design, it is set to the Creative Commons CC0 1.0 Universal license to ensure that the document can be freely reused.

Package Information

Essential elements in the SPDX package information include the package name, version, identifier, file name, supplier, originator, download location, and analysis status of package files. Additionally, the document should include the package verification code, checksums, homepage, source information, concluded license, and all license information from the package files. The declared license field, comments on the license, copyright text, summary description, detailed description, package comment, external reference field, and external reference comment should also be included. Furthermore, the package attribution text field, primary package purpose field, release date, build date, and a valid until date are part to the package information in SPDX.

File Information

Additionally, the SPDX specification can include a list of files associated with an application. This list comprises filenames, unique identifiers, file types, checksums, concluded license fields, comments, copyright texts, contributors, attributions, and file dependencies.

Snippet Information

Another use case in SPDX involves snippets. These are employed to incorporate additional licensing information that may arise from external code or intellectual property utilized in the software. They can also be used to encapsulate security-related information, such as CPE data. Snippets feature identifier fields, byte and line range fields, and include details on licenses and copyrights. Additionally, they provide space for comments, attributions, and other relevant information.

Other Licensing Information Detected

The SPDX SBOM may also include additional information about licensing situations. This facilitates the introduction of detailed information related to licensing. It can encompass additional license texts and license identifiers. Furthermore, these elements can be cross-referenced with other components or include supplementary comments.

Relationship between SPDX Elements

SPDX also allows for explicitly mapping relationships between elements within the SPDX format. These relationships can be annotated with comments to clarify their relevance and purpose for potential consumers of the SPDX SBOM. So, the SPDX document can represent a dependency structure and the relationship between the packages in a graph-like structure. The structure can also reference other documented assets like files, snippets,

or other SPDX documents. The relationship graph adds extra complexity because the type of relationship defines the direction in which the entry is pointed in the relationship graph. So, a relationship graph can be made up of relationships pointing in different directions.

Annotation Information

It is possible to annotate the document with additional information in cases where modifications were made or supplementary details were included. This annotation includes the name of the Annotator, the date of annotation, the type of annotation (e.g., review or other), an annotation identifier, and a field for additional comments.

Review Information

Initially, this section was intended to contain details regarding individuals or organizations that reviewed or approved the SPDX document. However, this section was deprecated in Version 2.0. It previously included information about the reviewer, the review date, and a review comment. Such information is now incorporated into the annotation information.

4.2 CycloneDX

CycloneDX is a standard developed and maintained by the OWASP Foundation. Initially designed and prototyped in 2017 as part of the OWASP Dependency-Track project, it has been continuously updated since then. The most recent version at the time of writing is the CycloneDX 1.4 standard, released in June 2023. Its primary intended use cases include vulnerability detection, license compliance, and analysis of outdated components. [15] All subsequent information pertains to the CycloneDX v1.5 specification reference. [14]

CycloneDX supports XML, JSON, and Protocol Buffers as data formats for CycloneDX. [15]

CycloneDX requires the inclusion of the bomFormat, the specification version, and the SBOM-version of the current project. Additionally, it is recommended to assign a serial number in the form of a UUID to each CycloneDX SBOM.

Metadata

The metadata section may include information about the SBOM, such as the time of creation, tools used to generate the SBOM, authors, and the components the SBOM describes. It may also detail the institution responsible for manufacturing the described software and license information associated with the SBOM. Additionally, further properties can be added using a key-value pattern.

Components

The Components section aims to enumerate all related software and hardware elements. For adding a component, it's mandatory to define its type. These types include application,

framework, library, container, operating system, device, firmware, or file. Each component must have a basic name, represented as a string. Optionally, in CycloneDX, a component may include additional details such as mime-type, supplier, author, publisher, product group name, version, an extended description, and references to other SBOM components. Additional attributes can encompass hashes, licenses, copyright information, Common Platform Enumeration (CPE), Package Uniform Resource Locator (PURL), and an SWID. External references, evidence from extraction or analysis, release notes, and properties added via a key-value pattern are also permissible. Moreover, Components facilitate documenting supply chain information through pedigree objects.

Services

The Services section is designed to document details about microservices at the network layer, as well as services in intra-process communication. Each service requires a name, specified as a string. Additional information can include the provider's details, group name, version, and a detailed description of the service. Furthermore, endpoints for service accessibility, authentication requirements, and cross-trust zone or boundary crossing details are essential. Data classification information should cover the flow of data, specifying whether it is inbound, outbound, bi-directional, or of unknown direction, along with a string-based classification of data sensitivity. Other vital details include license information, external references, related or included services, release notes, and properties presented as key-value pairs.

External References

CycloneDX enables the creator of a SBOM to add external references to the document for information that may be relevant but is not included in the SBOM. An external reference requires a URL and the type of reference, which is part of an enumeration. Additionally, it can include comments and hashes of the external reference, if applicable.

Dependencies

Dependencies in CycloneDX consist of a mandatory reference to a component that describes the dependency. They also include a list containing identifiers for components upon which the described component depends.

Compositions

A composition element is used to describe how components are assembled or composed. This element requires information about the type of aggregation regarding the completeness of the described relationships, such as complete, incomplete, incomplete_first_party_only, incomplete_third_party_only, unknown, or not_specified. Additionally, the composition can include a list of references to identify further resources associated with the assembly.

Vulnerabilities

If known at the time of generation, or if the SBOM is enriched at a later stage of its lifecycle, known vulnerabilities can be integrated into the SBOM. A vulnerability entry may include references to other elements within the SBOM, an identifier such as a Common Vulnerabilities and Exposures (CVE) number, and the source that published the vulnerability, including its URL and name. It can also encompass links to related vulnerabilities, rating details like severity, attack vector, or scores, associated Common Weakness Enumeration (CWE) identifiers, and a comprehensive description. Additional information may cover detailed insights into the vulnerability, recommendations for remediation or mitigation, related advisories published online, the date of the vulnerability's publication, updates to the information, acknowledgments to organizations or individuals such as researchers who discovered the vulnerability, and tools used for identification, confirmation, or scoring. Further, it may include an analysis or impact assessment of exploitability, the components or services impacted by the vulnerability, and additional properties presented in a key-value pair format.

Signatures

CycloneDX allows users to sign either the entire document or specific sections of it using the JSON Signature Format (JSF).

4.3 SWID

The SWID tag is defined in the ISO/IEC 19770-2:2015, which is part of the ISO/IEC 19770 family of standards. These standards provide a comprehensive framework for Information Technology Asset Management (ITAM). They are designed to assist organizations in effectively managing their software and hardware assets. The ISO 19770 family addresses various aspects of software management, including software identification, entitlement, and inventory, in an automated fashion. [1]

Therefore, ISO/IEC 19770-2:2015 is not solely focused on SBOMs but also introduces standardization across the entire lifecycle of a software product. SWIDs are intended to be used by the developers who create them and should then be passed along to licensors, packagers, distributors, and consumers. There, they should be utilized for purposes such as licensing, security, dependency tracking, and asset management. The standard delineates how SWIDs should be managed in all these phases, including how they should be added, updated, or removed from the devices on which the software is installed. [1]

The SWID itself is an XML file, defined by a XSD schema outlined in the standard. Although the standard is recognized by the NTIA for use as an SBOM, it was not originally designed for this specific use case. For instance, the minimum data requirements set by the SWID standard does not mandate the inclusion of information pertaining to the technical composition of utilized packages or third-party licenses. It only necessitates the provision of the software's Name and TagID, along with the Role, RegID, and Name of the tag creator. However, the SWID specification acknowledges the broad variety of

potential applications for SWIDs. Consequently, the schema offers numerous additional fields for optional use. SWID tags can facilitate the listing of packages and dependencies of a software product through the incorporation of <*Payload*> and <*Evidence*> elements in their primary tags. [1, 48]

However, CycloneDX and SPDX are specifically tailored for use as an SBOM and are backed by OWASP and the Linux Foundation, respectively. Therefore, no known tooling currently implements SWID as a format to generate an SBOM.

5 Methodology

The following section describes the scope of the thesis. It describes the selection of tools and the assessment process of the SBOM generation, conversion, and analysis. It also clarifies the limitations of this thesis and outlines guardrails for its scope.

5.1 Selection of Open Source Projects

An analysis was conducted on the most commonly used Docker Containers to select subject projects. For this, containers tagged on DockerHub as *Docker official image* are considered. Also, some of the *Docker Sponsored Open Source Software* images were considered with a focus on images provided by the software developers. Not all *Docker Sponsored Open Source Software* images were considered. The focus lies on images provided by the software developers, not images provided by third-party providers. Based on this, reference repositories on GitHub related solely to these Docker containers were selected. However, this is only applicable if a single repository is related to the Docker container and is hosted on GitHub. From the chosen GitHub projects, those that also manage their releases on GitHub were added to the release category. This limitation was implemented to facilitate automated data analysis based on unified APIs that can be implemented for automation.

5.2 Selection of SBOM Generators

A generator was considered eligible for this study if it complied with the following criteria to define a scope for this thesis:

- It must output results in one of the specified standards: SPDX or CycloneDX.
- It should be publicly available under a license that permits free usage in open-source projects.
- It must support a range of common programming languages, and not be specialized in only one language.
- It must be under active development or maintenance. Projects that were archived, declared deprecated, or saw no active development in the past two years were not considered eligible for this study.

To identify eligible SBOM generators, common search engines were utilized. Additionally, the publications of the OpenSSF Tooling working group were consulted, where a collaborative effort to survey current tools is undertaken by experts and peers in this field [41].

All identified tools matching the criteria were considered; however, those based on a tool already investigated in this analysis are not investigated in this thesis. Assuming that the results would be similar to the integrated tool to generate the SBOM.

The following tools were identified and deemed eligible:

- Syft [18]
- Trivy [17]
- CdxGen [19]
- Microsoft SBOM Tool (MST) [21]
- ScanCode Toolkit [23]
- Tern [24]
- Github Dependency Graph (GDG) [20]

While the Snyk CLI offers a wide variety of free services, including the generation of SBOMs, it requires a user account and user registration. The terms of service for Snyk specifically prohibit the use of their services for any form of competitive benchmarking [42]. For this reason, Snyk was not included in this study.

While SPDX and CycloneDX enjoy support of a wide variety of different tools, SWID is not supported by any of the selected generators. Therefore, SWID was excluded from further investigation.

Derived from the selected tooling, the most commonly supported phases in the software development lifecycle were identified. On one hand, most tools support the analysis of a given directory, which can contain source code or build outputs such as release files. Another common feature is the ability to scan a container image to generate a SBOM. Based on these common features, three phases were identified.

The *sources phase* refers to a SBOM based on the source code of a sample project provided by GitHub. The *release phase* refers to a SBOM based on the release files of a sample project provided by projects that manage their releases with the corresponding GitHub feature. The *container phase* refers to a SBOM based on the container image of a sample project provided by DockerHub.

5.3 SBOM Quality Assessment

For the purposes of this study, a system was developed to automate the generation, storage, and analysis of SBOM data. The system incorporates various components, each of which is described in the subsequent sections.

5.3.1 Generation

The SBOMs were generated within a custom Docker container, which was based on Arch Linux. This container was configured to install all necessary tools for SBOMs creation. To minimize storage and bandwidth requirements - and to avoid exceeding rate limits of API endpoints - all tools were consolidated into a single container. Resources were fetched from the API and subsequently, SBOMs were generated sequentially for each project. Although SBOMs generation for individual projects was not parallelized, the processing of different projects was highly parallelized to accommodate the large sample size within a reasonable time frame.

The generation tooling was encapsulated in the Docker container alongside a SpringBoot Java application, which orchestrated the generation process and managed data storage.

Each generation operation was configured with a maximum timeout of one hour. Generation logs were preserved for subsequent analysis and were recorded using Asciinema, a tool designed to capture and replay terminal sessions. The SBOMs for SPDX and CycloneDX were produced in JSON format and stored in a PostgreSQL database as JSON blobs. If a generator only supports one of the two investigated standards, the SBOMs gets converted accordingly. If a generator was capable of producing SBOMs in both formats, generation was performed in each format accordingly.

5.3.2 Conversion of SBOMs

Not all tools evaluated in this study are capable of generating SBOMs in both the SPDX and CycloneDX file formats. To facilitate a comparative analysis of the two formats, it was necessary to convert the SBOMs into the format that was not originally produced by the tool. For instance, CdxGen and ScanCode produce SBOMs exclusively in the CycloneDX format, thus requiring conversion to SPDX. Similarly, MST and GDG produce SBOMs solely in the SPDX format, necessitating conversion to CycloneDX.

Several tools were considered for the conversion of SBOM between CycloneDX and SPDX. The SPDX project provides a Java library with command-line utility to convert CycloneDX SBOMs to SPDX, but it is not able to convert them vice versa [9]. CycloneDX offers the CycloneDx-CLI, which is capable of converting SPDX in both directions [10]. Syft also offers a tool within their CLI for bidirectional SBOM conversion [18]. Additionally, a project from bom-squad facilitates conversion between SPDX and CycloneDX in both directions [8].

After a series of tests, the Syft SBOM conversion tool was selected for its accurate value mapping and stability in processing data.

Nonetheless, it should be noted that the conversion process may result in less accurate data, or in some instances, the inclusion of empty values by the conversion tool. Consequently, all metrics derived from converted data will be marked with an asterisk '*' henceforth in this paper, indicating the potential bias in the sample data du to conversion of the SBOMs with third party tooling.

A comprehensive examination of the various converters exceeded the scope of this study. Therefore, it has been published as an blog post to allow for a full understanding of the tool selection process. [5]

5.3.3 Interacting with SBOM Data

To facilitate the analysis and interaction with the SBOMs, a web-based tool was developed for comparing and engaging with the SBOMs data. This tool was constructed using Vue.js 3 for the front end and a SpringBoot Java framework for the back end, providing a REST API. It was designed to offer straightforward access to the metadata of the sampled generations and the SBOMs themselves.

The primary aim was to grant insights into the SBOMs generated, particularly concerning the additional data collected. A project overview feature was introduced to track

which generators succeeded in producing SBOMs at various stages, and to provide a preliminary assessment of the data quality.

Additionally, the tool incorporated a project-specific view for comparing the SBOMs created for a single project. Three distinct viewing modes were established:

First, a *Basic Insights* view was devised to facilitate an initial examination of the data and to verify compliance with the NTIA minimum elements.

Second, a *Dependencies* view was constructed to align and display the declared dependencies from the SBOMs in a tabular format, enabling an evaluation of intersections in detected dependencies across different SBOMs. This view allows for the utilization of various identifiers as references. In SPDX, the External Reference Locator and the SPDX name was used for mapping intersecting dependencies. In CycloneDx, the PURL or the dependency name was used. All identifiers could be used with or without versioning of the dependencies. Furthermore, a physical simulation (force plot) was integrated to visually represent the dependencies identified by different SBOMs and how intersections form between them.

Lastly, a *License Information* view was introduced to provide insights on the extent of license data coverage within an SBOM, highlighting the different license-related fields.

listings amssymb

6 SBOM Generators

The various SBOM generators implemented exhibit distinctive features for diverse applications. Some are designed for exclusive phases of the software development lifecycle, while others incorporate specialized features that serve as unique selling propositions. These differences present challenges when attempting to conduct a general comparison among them. The generators distinguish themselves based on the programming languages they support, compatibility with certain build tools or package managers, supported platforms, or their capabilities in scanning specific assets such as virtual machine images or AWS accounts.

6.1 Generator Specifications

Table 6.1 outlines some general specifications for the generators under consideration. Notably, the GDG licensing and programming language information are not available. While the service is free to the Github community, Github does not share the source code publicly. Furthermore, the table reveals that all generators except CdxGen can produce SPDX-formatted SBOMs, and five out of the seven generators are capable of producing CycloneDX SBOMs.

Generator Spesifications					
Generator	License	Language	Backed by	SPDX	CDX
CdxGen	Apache-2.0	JavaScript	OWASP	×	✓
Syft	Apache-2.0	Go	Anchore	✓	✓
MST	MIT	C#	Microsoft	✓	×
Trivy	Apache-2.0	Go	Aquasecurity	✓	✓
Tern	BSD-2-Clause	Python	Community	✓	✓
ScanCode	Apache-2.0	Python	nexB	✓	✓
GDG	-/–	-/–	Github	✓	×

Table 6.1: *Generator Spesifications*

In Table 6.2 a list of supported programming languages for each SBOM generator is conducted, derived from the corresponding documentation of each generator. It is important to note that this list does not fully represent the build tool compatibility within those languages. For instance, the Gitlab Dependency Graph supports Java when utilizing Maven by reading the project's pom.xml. However, it does not extend support to Gradle, which introduces additional complexity with Groovy or Kotlin scripts [20]. Furthermore, the generator Tern is excluded from this list as it does not analyze programming languages per se. Tern is designed to scan container images for installed packages. While Tern's functionality can be enhanced with the ScanCode-toolkit to perform package scanning, this integration is not included in the study since the ScanCode-toolkit is independently evaluated. While ScanCode documents the capability to generate SPDX documents, it is

limited to producing SPDX output in RDF or Tag-Value format [22]. This limitation posed a challenge for this study, as converting the RDF or Tag-Value output to a JSON SPDX file for further analysis was difficult. Therefore, ScanCode was employed solely to generate a CycloneDX file, which was subsequently converted to SPDX using a third-party tool.

Supported Programming Languages[1]						
Language	CdxGen	Syft	MST	Trivy	ScanCode	GDG
C	✓	✓	×	✓	✓	×
C++	✓	✓	×	✓	✓	✓
Clojure	✓	×	×	×	×	×
Dart	✓	✓	×	✓	✓	✓
Elixir	✓	✓	×	✓	×	×
Erlang	×	✓	×	×	×	×
Go	✓	✓	✓	✓	✓	×
Haskell	✓	✓	×	×	×	×
Haxe	×	×	×	×	✓	×
Java	✓	✓	✓	✓	✓	✓
JavaScript	✓	✓	✓	✓	✓	✓
.Net	✓	✓	✓	✓	✓	✓
Nix	×	✓	×	×	×	×
Ocaml	×	×	×	×	✓	×
ObjectiveC	×	✓	✓	×	✓	×
Perl	×	×	×	×	✓	×
PHP	✓	✓	×	✓	✓	✓
Python	✓	✓	✓	✓	✓	✓
R	×	×	×	×	✓	×
Ruby	✓	✓	✓	✓	✓	✓
Rust	✓	✓	✓	✓	✓	✓
Swift	✓	✓	✓	✓	✓	✓
[1]list is an approximation based on the provided documentation						

Table 6.2: *Supported languages by Generator*

6.2 Generation Summary

Table 6.3 provides a comprehensive account of all SBOMs generated using various generators across different phases. The SBOMs were produced based on a sample of 205 subject projects for the container phase, 174 projects for the source phase, and 117 projects for the release phase. Several tools encountered crashes during the attempt to generate an SBOM or hit a one-hour timeout that halted the generation process. Notably, Tern experienced over 28 crashes.

Upon examining the generated SBOMs, the quantity of enumerated dependencies is categorized into three groups: SBOMs containing at least one dependency, those containing at least 10 dependencies, and those with at least 100 dependencies. While the container

Generation Summery							
	Generator	All samples	Successfull	Failed	0< Dependencies	10< Dependencies	100< Dependencies
Container	CdxGen	205	202	3	186	182	158
	MST	205	205	0	205	159	122
	Syft	205	204	1	196	193	170
	Tern	205	177	28	177	164	145
	Trivy	205	204	1	204	194	167
Release	CdxGen	117	117	0	5	2	0
	MST	117	117	0	117	0	0
	Syft	117	117	0	19	10	3
	Trivy	117	116	1	116	0	0
Source	CdxGen	174	163	11	146	113	73
	GDG	174	171	3	171	126	75
	MST	174	174	0	174	99	64
	ScanCode	174	162	12	126	33	10
	Syft	174	174	0	118	103	74
	Trivy	174	168	6	168	89	58

Table 6.3: *Generation Summary*

and source phases both have SBOMs in all of these categories, the release phase SBOMs are notable for predominantly containing only one or no enumerated dependencies.

A detailed description of the generation process of each generator and phase is provided in the Appendix.

The execution time for the generating of all SBOMs was documented in table 6.4. The generation process was aborted after 60 minutes if not completed. This metric is not applicable to the GDG SBOMs, as they were obtained from the GitHub API rather than being generated during the study.

In instances of timeouts, Tern experienced this issue in two separate cases. During the sources phase, timeouts impacted CdxGen on two occasions. ScanCode encountered timeouts in 12 instances. Trivy faced a timeout once in the release phase and twice in the sources phase.

Generator Spesifications						
Mode	Syft	Trivy	MST	CdxGen	ScanCode	Tern
Container	50,0s	102,2s	87,1s	223,4s	-/–	333,7s
Release	12,0s	148,6s	13,9s	8,9s	-/–	-/–
Source	24,9s	97,0s	30,6s	235,3s	577,5s	-/–

Table 6.4: *Generator Average Execution Time*

It is important to note that CdxGen, GDG, and MST generators support output in only one of the investigated SBOMs standards. Consequently, the SBOMs were converted

to the missing format where necessary. Due to issues transforming the RDF-SPDX file generated by ScanCode to a JSON-SPDX file the CycloneDX was thus converted to SPDX as well. Trivy and Tern support output in both SBOMs formats, but only sequentially; hence, generation was performed twice, a factor included in the average generation time documented in Table 6.4. In contrast, Syft is capable of generating outputs in multiple formats simultaneously. Therefore, the generation time listed in Table 6.4 corresponds to the total time taken to produce an SBOMs in both the SPDX and CycloneDX formats.

7 SBOM Data Assessment

After generating the samples of SBOMs, the subsequent step involves assessing their data quality. This assessment can be conducted across various dimensions and is divided into three parts. The first part presents the overall data enrichment of the generated SBOMs, referring to the question: how much information has been added by the generators to facilitate the different features of the SPDX and CycloneDX specifications. The second part applies the minimum elements metrics to these SBOMs. The third part will discuss the results from tools introduced to evaluate the quality of the generated SBOMs.

7.1 SBOM Overall Enrichment

While the different SBOM formats offer a rich set of use cases, they are able to reflect with their schema structure, not all of them are enriched with information by the SBOM generators. The Tables 7.1 and 7.2 show a count of different features of the different SBOM types and which generators made use of them in which phase. Table 6.3 can be used as a reference to the results.

7.1.1 CycloneDX Overall Enrichment

Every CycloneDX SBOM sample includes fundamental data such as *bom-format*, *spec-version*, *serial number*, *version*, and *metadata*. Support for component listing is uniformly provided by all generators examined. Nevertheless, most generators face challenges when analyzing the release phase. Only a select few, specifically CdxGen and Trivy, offer insights into the dependency structure and how they are interconnected. A lot of use cases, such as *signature addition*, *service listing*, *external references*, *compositions*, *vulnerability documentation*, *annotations*, *formulations*, or *property inclusions* are not utilized by any of the generators under investigation.

7.1.2 SPDX Overall Enrichment

All generated SPDX samples contain essential elements such as the *SPDX Identifier (SPDXID)*, *creation information*, *Data License*, *Name*, and the *SPDX specification version*. MST, GDG, and Tern are unique in including *DocumentDescribes* elements. Although all generators produce samples with packaging information, the quality and detail of the package listings vary. Only CdxGen, Syft, and Tern augment these with external licensing data. Syft and Trivy are distinguished during the container phase by their inclusion of data on additional files in the SBOM. MST displays a singular instance of additional file information, but it is not commonly employed. With the exception of GDG, all generators facilitate the depiction of relationships among the resources detailed in the SBOM. Tern is the sole generator that appends comments to the SBOMs it produces. However, segments such as *Annotations*, *External Document References*, *Reviews*, or *Snippets* have not been utilized by any of the generators.

CycloneDX overall enrichment								
	Generator	bom Format	Spec Version	Serial-number	Version	Meta-data	Compo-nents	Depen-dencies
Container	CdxGen	202	202	202	202	202	186	1
	MST*	205	205	205	205	205	205	0
	Syft	204	204	204	204	204	198	0
	Tern	178	178	178	178	178	162	0
	Trivy	202	202	202	202	202	193	202
Release	CdxGen	117	117	117	117	117	5	0
	MST*	117	117	117	117	117	117	0
	Syft	117	117	117	117	117	19	0
	Trivy	117	117	117	117	117	0	117
Source	CdxGen	163	163	163	163	163	146	88
	GDG*	171	171	171	171	171	171	0
	MST*	174	174	174	174	174	174	0
	ScanCode	162	162	162	162	162	126	0
	Syft	174	174	174	174	174	118	0
	Trivy	170	170	170	170	170	101	170
Not all categories listed								

Table 7.1: *CyclonoeDx overall enrichment*

SPDX overall enrichment													
	Generator	SPDXID	Comment	Creation-info	Data-license	Name	Spdx-Version	Document-namespace	External-Licensing	Document-Describes	Packages	Files	Relation-ships
Container	CdxGen*	202	0	202	202	202	202	202	25	0	186	0	202
	MST	205	0	205	205	205	205	205	0	205	205	0	205
	Syft	204	0	204	204	204	204	204	183	0	196	196	204
	Tern	177	177	177	177	177	177	177	158	177	177	0	177
	Trivy	202	0	202	202	202	202	202	0	0	202	81	202
Release	CdxGen*	117	0	117	117	117	117	117	0	0	5	0	117
	MST	117	0	117	117	117	117	117	0	117	117	0	117
	Syft	117	0	117	117	117	117	117	12	0	19	19	117
	Trivy	116	0	116	116	116	116	116	0	0	116	0	116
Source	CdxGen*	163	0	163	163	163	163	163	43	0	146	0	163
	GDG	171	0	171	171	171	171	171	0	171	171	0	0
	MST	174	0	174	174	174	174	174	0	174	174	1	174
	ScanCode*	162	0	162	162	162	162	162	0	0	126	0	162
	Syft	174	0	174	174	174	174	174	20	0	118	118	174
	Trivy	168	0	168	168	168	168	168	0	0	168	0	168
Not all categories listed													

Table 7.2: *SPDX overall enrichment*

7.2 SBOM Schema Compliance

As indicated in Table 9.4, all generated SBOMs include a version number; however, there is variation in the specific SBOM version implemented by different tools. At the time of writing, version 2.3 represents the latest release of the SPDX spesification, and version 1.4 is the most recent CycloneDX specification version. Notably, CycloneDX version 1.5 was released very recently at the time of writing. Consequently, its adoption is still on the move.

Only ScanCode and Tern continue to utilize older versions of their respective specifications. In contrast, all other evaluated tools support recent versions of either the SPDX or CycloneDX specifications. Notably, CdxGen and Trivy have already adopted the new CycloneDX 1.5 specification. However, for the purpose of converting CdxGen samples to the SPDX format, it was necessary to pin the specification version to 1.4 for the CdxGen SBOM generator, so the samples could be converted to SPDX.

	SPDX		CycloneDX	
Generator	Version	Invalid	Version	Invalid
CdxGen	2.3*	0	1.4	3
GDG	2.3	0	1.4*	0
MST	2.2	0	1.4*	0
ScanCode	2.3*	0	1.3	126
Syft	2.3	0	1.4	0
Tern	2.2	0	1.3	29
Trivy	2.3	0	1.5	209

Table 7.3: *Version coverage*

To gain an initial understanding of the validity of the generated data, it was validated against the schema files provided by SPDX and CycloneDX for the respective versions. This process offers insights into whether the standards are implemented correctly, thereby allowing the expectation that other tools could successfully consume the SBOMs.

Upon validating all SPDX samples against the respective versions of the SPDX schemas for versions 2.2 and 2.3, no errors were found in any of the SBOMs.

However, the results were more varied when examining the CycloneDX samples. Only Syft managed to generate all CycloneDX samples without producing any invalid files.

A total of 209 CycloneDX samples generated by Trivy were found invalid when validated against the CycloneDX Schema version 1.5. All the problematic SBOMs generated by Trivy failed during an error matching the exact instance, describing the license schema.

CdxGen produced 3 invalid SBOMs. Two of these were due to instances of properties not allowed in the file. The third was related to duplicate elements in the dependency relationship description.

When validating the CycloneDX samples generated by Tern against the CycloneDX schema version 1.3, 29 SBOMs were identified as invalid. All were disputed due to duplicate elements in the component section of the SBOMs.

ScanCode produced 126 SBOMs that were found invalid validating against the CycloneDX schema version 1.3. All of the invalid SBOMs were related to uninitialized elements that should be represented by a object or a primitive type, but are null in the SBOM.

While all the identified discrepancies are errors in terms of conformity with the respective schema, none rendered the SBOMs unusable for further processing in this study. However, it is noteworthy that ScanCode also produced an invalid metadata properties object, rendering it incompatible for use with other tools, for instance, in converting the SBOMs from CycloneDX to SPDX. This metadata object was fixed before conversion from SPDX to CycloneDx.

7.3 Current Quality Metricses

To assess the quality of a SBOM, it is necessary to define a metrics that can measure the quality of the SBOM. For this reason, several of the currently known metrics are introduced to assess an SBOM. The NTIA has published a set of minimum elements for a SBOM [36]. Also, tools like the SBOM Scorecard and the SBOMQS project are introduced.

7.3.1 NTIA minimum elements

The NTIA has published a list of minimum elements for SBOMs that are independent of the SBOM format. These requirements encompass seven distinct data fields. [36]

- **Supplier Name** refers to the name of the entity, such as a company, project, or developer, that supplies a component and thereby identifies the supplier.
- **Component Name** refers to the name given to a component by the supplier.
- **Version of the Component** refers to an identifier used by the supplier to differentiate between various versions of a software component.
- **Other Unique Identifiers** refers to additional identifiers that are employed to recognize the component in build systems, package repositories, or other relevant databases, such as those pertaining to vulnerabilities or exploits.
- **Dependency Relationship** delineates the connections with other related components that incorporate another component.
- **Author of the SBOM Data** refers to the name of the individual or organization that compiled the SBOM.
- **Timestamp** refers to the date and time when the SBOM was created.

Although the NTIA has defined these requirements without adhering to a technical specification of any standard, both SPDX and CycloneDX have published mappings that align the NTIA's minimum elements with the SPDX [34] and CycloneDx[35] specifications, respectively (see Table 7.4).

The NTIA minimum elements can be categorized into two different types: fields that refer to a single entry within the SBOM, such as the author or timestamp, and fields that relate to components, which may occur multiple times within an SBOM. The latter

NTIA	SPDX	CycloneDX
Supplier Name	bom.packages[].supplier	bom.metadata.supplier bom.components[].supplier
Component Name	bom.packages[].name	bom.components[].name
Version of Component	bom.packages[].versionInfo	bom.components[].version
Other Unique Identifiers	bom.packages[].SPDXID bom.documentNamespace	bom.components[].cpe bom.components[].purl bom.components[].swid
Dependency relationship	bom.relationships[]	bom.dependencies[]
Author of the SBOM data	bom.creationInfo.creators	bom.metadata.authors
Timestamp	bom.creationInfo.created	bom.metadata.timestamp

Table 7.4: *NTIA minimum elements mapping*

category includes the supplier name, component name, version of the component, other unique identifiers, and dependency relationships.

Comparing the alignment of the two standards with the NTIA minimum elements, it can be argued that CycloneDX sets a higher benchmark than SPDX. Unlike SPDX, which focuses solely on the package suppliers for each enumerated package in an SBOM, CycloneDX also considers the supplier of the SBOM itself, but this could also be seen as a duplicate reagarding the author of the SBOM in the NTIA minimum elements. Moreover, while SPDX utilizes SPDXID and the documentNamespace of the SBOM, CycloneDX mandates external identifiers such as PURL, CPE, or SWID to fulfill the requirement of unique identifiers for each package. Notably, SPDXIDs are internal identifiers that can be generated randomly, rendering them less useful for external processing outside the SBOM's internal scope.

Compliance of SPDX SBOM with NTIA Guidelines

In Table 15.1, the comprehensive coverage of the NTIA minimum elements within the generated SPDX samples for this project is presented. These requirements are quantified as the percentage of samples that comply with each specified requirement in the respective phase and generator.

The inclusion of *supplier names* within dependencies yields variable results. It is noteworthy that although Syft does not populate the supplier field in either phase of the SBOM generation process as per the mapping to the NTIA minimum elements, there is partial coverage in the *originator* field which is related to the supplier. According to the SPDX specification, the *package supplier* could be a distribution platform like SourceForge, which may not have a direct relation to the package's origin. To bridge this discrepancy, the SPDX standard introduces a *package originator* field for identifying the individual or organization from which the package originates. The NTIA defines the Supplier Name as: "The name of an entity that creates, defines, and identifies components" [36]. Considering this, one could argue that the *originator* field aligns more closely with the NTIA criterion than the *supplier* field. If we attribute the *originator* field in Syft's analysis, it would achieve

a coverage of 64% for container images, 0.5% for release files, and a scant 0.014% for source code scans, indicating a level of awareness for detecting a supplier consistent with standards.

All SPDX SBOMs meet the NTIA's requirements for the *SBOM Author*. However, it is essential to recognize that the NTIA's mapping of SPDX requirements does not refer to a simple string, but to a complex data structure that can encapsulate various details. The SPDX specification anticipates a key-value pairing for information pertaining to a Person, Organization, or Tool. If an SBOM contains any of these details, it satisfies the NTIA requirements. While all generated SBOMs include the *Tools* parameter, none provide the *Person* parameter. The *Organization* parameter is absent in the SBOMs from GDG and Tern. Conversely, Syft and Trivy list their respective organizations, Anchore and AquaSecurity. MST alone mandates users to specify an organization to produce a SBOM. CdxGen and ScanCode, which were converted using Syft to the SPDX format, list Syft and Anchore as the *creator* due to the conversion. The SPDX specification states: "If the SPDX document was created on behalf of a company or organization, indicate the entity name. If the SPDX document was created using a software tool, indicate the name and version of that tool" [45]. This suggests that the *Organization* field is intended for the entity that generated the SBOM, rather than the supplier of the tooling, which should be documented in a separate field.

In analyzing the coverage of version information, it is evident that some tools struggle to provide version information for their enumerated dependencies. In the release phase, Trivy does not supply any version information with their package information. Particularly in the Sources phase, Syft, Trivy, and ScanCode struggle to retrieve license information. These issues are based on the different approaches to retrieving information from the sources. Some build systems, like Java's Gradle, introduce complex mechanisms to maintain unified versions in complex projects. This complexity makes it difficult to retrieve version information from a project. Tools like CdxGen benefit from communicating with the build system API, which can retrieve such information reliably. However, this comes at the cost of depending on the build tool's stability; if the build tool crashes, the information retrieval process also fails. Also the build tool needs to be availible at the time of SBOM generation. On the other hand, tools like Syft and Trivy implement their own parsers. They don't need to rely on the build tools to retrieve this information and therefore run more stably, but they can't reflect the complete complexity of the build system.

The timestamps in all SPDX SBOMs accurately reflect the time of generation or conversion. Also all SPDX SBOMs enumerate the Component name and a unique identifier for the components. GDG is the only SBOM generator not providing any dependency relationship information. This will be further investigated in Chapter 8.

Compliance of CycloneDX SBOMs with NTIA Guidelines

In Table 15.2, the comprehensive coverage of the NTIA minimum elements within the generated CycloneDX samples for this project is presented. These requirements are

quantified as the percentage of samples that comply with each specified requirement in the respective phase and generator.

Among the SBOM generators examined, all implemented PURLs as a unique identifier for their components; however, only Syft incorporated the use of CPE alongside PURL for component identification. Furthermore, Syft is distinguished by its utilization of SWIDs to describe the base image of a container, successfully assigning SWIDs to 193 out of 204 scanned images. That the results for the CycloneDX unique identifiers are not achiving a 100% coverage like the SPDX results do might be attributed to the higher bar set by the mapping of the NTIA requirements to the CycloneDX schema. Providing valid external reference locators is more difficult that generating a internal identifier. CycloneDX also does that for all dependencies with the *bom-ref* identifier.

CdxGen stands out as the sole generator that provides an *author* field within the SBOM. This field represents a complex data structure capable of encompassing *bom-ref*, *name*, *email*, and *phone*. The specification of CycloneDX states: "The person(s) who created the BOM. Authors are common in BOMs created through manual processes. BOMs created through automated means may not have authors." [14]. CdxGen generates all SBOMs with *Prabhu Subramanian* as author of the SBOM. He is the developer behind CdxGen. According to this CycloneDX specification, it is correct to omit the author for the SBOM if its generated automatically. The CycloneDX specification for authors is therefore not alligned with the spirit of the NITA minimum elements that states "The name of the entity that creates the SBOM data for this component."[36]

Moreover, only Trivy provided the Supplier name for some of the container SBOMs generated. Also only a view tools provide dependency relationship information in their CycloneDX SBOMs. While ScanCode and Trivy initialize the dependency relationship section, only CdxGen and Trivy provide meaningful descriptions of the relationships between components.

Looking at the version information provided, the CycloneDX SBOMs suffer from the same effects as the SPDX SBOMs do when different generators struggle to retrieve the license information.

7.3.2 eBay SBOM Scorecard

In November 2022, eBay introduced a concept for measuring the quality of SBOMs by applying their self designed metrics to a SBOM. eBay evaluates five distinct aspects of the SBOMs, whether in SPDX or CycloneDX format, and aggregates these into a cumulative score. The aspects assessed are *Specification Compliance*, *Package ID*, *Package Versions*, *Package Licenses*, and *Creation Information*. Each aspect is assigned a different weight, which influences the overall SBOM score. [31]

The SPDX SBOM Scorecard Table 15.3 presents an aggregation of scores for the evaluated SBOMs. It reveals that Syft, during the container phase, achieves an exceptionally high score of 92%. In the release phase, MST attains the best score with 60%, and in the sources phase, Syft leads once again with 66%. MST and Tern are the only tools to receive a 100% for their package versioning coverage. Syft is also the sole generator recognized for providing at least some form of package identifiers. Although all generators are credited

with providing complete creation information, only GDG scores 80%; this is attributable to GDG not offering a tool that generated the SBOM, which is technically accurate given that GitHub provides an API to retrieve SBOMs.

In the CycloneDX SBOM Scorecard Table 15.4, the scores for the CycloneDX samples indicate that Syft once again achieves the highest overall score in the container phase with 87%. ScanCode delivers the best results for the source phase with 75%, and MST leads in the release phase with a score o 77%. A comparison of the SPDX and CycloneDX results shows that the proportional outcomes are somewhat correlated. The notable exception is the performance of Trivy across all phases in the CycloneDX samples, where it scores poorly. However, this may not reflect the actual quality of the samples but rather the fact that Trivy is the only generator supporting CycloneDX version 1.5 as an SBOM format, which may not be compatible with the Scorecard tool.

7.3.3 SBOMQS Quality Metrics for SBOMs

SBOMQS is a project by Interlynk-io that provides quality metrics to assess SBOMs in both SPDX and CycloneDX formats. SBOMQS introduces a range of features for evaluating an SBOM. Each feature is scored on a scale from 0 to 10. These features are then categorized into groups such as *NTIA-minimum-elements*, *Structural*, *Semantics*, *Quality*, and *Sharing*.

Comparing the results between the SPDX results in Table 15.5 and the CycloneDX results in Table 15.6 the biggest difference can be seen in the sharing category. While all SPDX SBOMs provide sufficient licensing for sharing, no CycloneDX SBOM provides any licensing information. This is related to the fact, that SPDX specified their standard with a default license in place while CycloneDX does not specify a default license. This can make it much easier to exchange and process a SBOM with SPDX.

While Structurally all SBOMs in SPDX are fine, the Trivy SBOMs were the only ones that received a lower score. Looking at the scores for the NITA minimum requirement generated by SBOMQS the results follow the same trent as this, generated in this study. In the overall average of all tools the SPDX SBOMs achieve a score of 8.14 while the CycloneDX SBOMs only reach a score of 6.77. This might be also related to the issue that CycloneDX maps the requirements to more demanding benchmarks then SPDX does. Nevertheless, the SBOMQS result for the NTIA minimum elements deviate from the results we generated, which can relate to a different implementation, the usage of other mappings for the requirements or the evaluation of additional requirements related to the original NTIA minimum elements.

On the quality and Semantic parts CycloneDX archives a slightly higher score with 4.90 for the quality and 4.19 for the overall semantic in all tools, while SPDX reaches a score of 3.85 for the quality and 3.18 for the semantic of all samples.

8 Dependency Insights

The dependencies enumerated in an SBOM arguably contain the most important information. They encompass various identifiers, such as CPEs and PURLs, and include license information along with extensive data about the dependencies related to a software product. Thus, evaluating the effectiveness of an SBOM in terms of its dependencies involves two aspects: the quantity and the quality of the dependencies.

Discussing the quantity of an SBOM is challenging due to the absence of a single source of truth that can provide a universally accepted benchmark for verifying the completeness of a dependency list. The completeness of the dependencies identified in an SBOM depends on the methodologies of the tools and processes used for its generation. From a process standpoint, SBOMs can be generated at different phases of the software development lifecycle. Most SBOM generation tools are designed to operate at the source code level, during build time, or post-build for analyzing release files or container images. These tools vary in their support for different programming languages and build tools. Additionally, other factors, such as the depth of the scan during SBOM generation, need consideration. Some SBOMs list only first-layer dependencies directly referenced by the build tools, while others include transitive dependencies that arise due to other dependencies built upon them.

Regarding the quality of dependencies in a SBOM, the question arises as to how comprehensively a dependency is enriched. Both SPDX and CycloneDX offer numerous optional fields that can enhance a dependency's informational value. The extent to which SBOM generators enrich this data is crucial. Do they only add available information obtained from library findings or build tool scans, or do they incorporate additional information from package repositories or third-party sources?

8.1 Dependency Intersections between SBOMs by Examples

While there is no single source of truth to check for the completeness of all dependencies in an SBOM, it is insightful to evaluate how the results from different SBOMs intersect across various phases of the software development lifecycle produced by different generators. Ideally, all dependencies listed in each scan result should completely intersect, regardless of the SBOM generator or the phase in which the SBOM is generated. However, in practice, the intersections between different SBOM generators and the various phases in the software development lifecycle are often scattered. On the one hand, this can be by design. A SBOM generated at the sources phase can list test dependencies or build tools that are not available to a container SBOM, while the container SBOM can enumerate packages installed on the base image of the container that was not available to the sourcecode SBOM. On the other side, this can be due to the limitations of the SBOM generator. For instance, a SBOM generator might not support a specific programming language, build tool, or enrich the data for a dependency differently, so it can not be mapped to other results.

To accurately calculate data overlap, several key issues must be addressed. Although an SBOM lists dependencies, there are various identifiers within a dependency that can be utilized for comparison. In SPDX format, each dependency is characterized by a name, a version, and external reference locators. While the semantics of the name and version are not precisely defined, the external reference locators use standardized identifiers such as CPE or PURL intended to identify a dependency uniquely.

While CPE and PURL were integrated to be used as identifiers, they only partially fulfill this role. There is an ongoing discussion regarding the use of PURL as a package identifier. PURL aims to "standardize existing approaches to reliably identify and locate software packages" [38]. Although the PURL specification presents a robust attempt to achieve this, it does not offer a uniform method to compose a PURL for a single package. Different notations can refer to the same package with a PURL, as discussed in the following issue related to the PURL specification [16]. While all SBOMs included a PURL for each dependency, only Syft additionally assigned at least one CPE to each dependency. Therefore, mapping based on CPE was not feasible.

Therefore, while PURL does not provide an ideal solution for identifying and mapping packages across different SBOMs, it might be the best option available at the time of writing. Notably, there are challenges in using PURL as an external reference locator to identify packages in an SBOM. However, for the purpose of this investigation, only a portion of the PURL is required to fulfill this use case. Specifically, this chapter uses only the scheme, type, namespace, and name to map dependencies. The version, qualifiers, and subpath components of a PURL are not considered in this context to investigate intersections between the sample SBOMs.

An SPDX dependency may include multiple external reference locators. However, these values can be redundant within the same SBOM. For instance, in the SPDX SBOM generated by Syft for the Keycloak project, the JUnit dependency is listed 66 times identically (same PURL provided), differing only in the randomly generated SPDXID and the source info, which indicates the location of the dependency discovery. This repetition occurs because Syft added the dependency separately for each of the 66 different locations within the Keycloak repository where the dependency was added. A similar situation is observed in the CycloneDX SBOM generated by Syft for Keycloak. Consequently, for the purposes of this evaluation, a dependency is considered identified if it appears at least once in an SBOM. If a dependency is listed multiple times, it will be consolidated into a single entry to facilitate mapping with other SBOMs.

To illustrate the concept of investigating the intersections between the different SBOMs generated by different tools and in different phases, first, a real-world example is investigated. Therefore, the Keycloak project was first picked for being one of the biggest sample projects in this investigation.

Examining the results of the Keycloak scan, eight SBOMs were generated with meaningful results. Syft was the sole generator to produce an SBOM based on release files that list related dependencies. All other generators failed to identify sufficient dependencies from the release phase. MST also encountered issues during the container and source scans. In its sample, Tern could not provide PURLs with the detected dependencies. Additionally,

it included dependencies in the SPDX format that were not present in the CycloneDX output. Unfortunately, Trivy experienced a crash while scanning the Keycloak sources.

The Table 8.1 illustrates how different dependency entries can be consolidated by name or PURL in both SPDX and CycloneDX formats. The table also highlights the difference in data reduction when disregarding the version and further qualifiers of a dependency. While varying versions of the same dependencies pose less of an issue for release and container-based SBOMs, it significantly affects source-based SBOMs. This approach can condense the data to 4083 distinct dependencies, using PURLs as identifiers to map the dependencies across different SBOMs.

Keycloak distinct Dependencies over all SBOMs by different identifiers											
		SPDX					CycloneDX				
		No version		with version			no version		with version		
	Generator	PURL	Name	PURL	Name	All	PURL	Name	PURL	Name	All
Container	CdxGen	476	476	481	481	481	476	476	482	481	482
	MST	1	44	1	44	44	2	44	2	44	44
	Syft	850	849	851	851	873	850	850	851	852	874
	Tern	0	5	0	5	5	0	0	0	0	0
	Trivy	552	554	553	555	578	551	551	552	553	553
Release	CdxGen	0	0	0	0	0	0	0	0	0	0
	MST	1	1	1	1	1	1	1	1	1	1
	Syft	845	840	846	842	985	845	840	846	842	985
	Trivy	0	1	0	1	1	0	0	0	0	0
Source	CdxGen	2848	2779	3710	3710	3714	2848	2779	3728	3710	3728
	GDG	1656	1824	2025	2504	2504	1657	1824	2026	2504	2504
	MST	1	1	1	1	1	1	1	1	1	1
	ScanCode	231	231	231	231	231	231	231	231	231	231
	Syft	1555	1553	1908	1908	3014	1555	1553	1908	1908	3014
distinct total		4083	6032	5858	8349	12432	4083	5517	6397	7844	12417

Table 8.1: *Keycloak distinct Dependencies over all SBOMs by different identifiers*

The dependency plot for the Keycloak sample project, as shown in Figure 8.1, illustrates the intersections among all generated SPDX SBOMs. Each gray dot represents a project dependency linked to one or more green, orange, or blue dots. Green dots represent SBOM generators based on source code analysis, blue dots denote those analyzing container images, and orange dots correspond to the analysis of release files. The relationships are established based on the PURLs added to the SBOMs' dependencies by the generators in the external reference locators. The plot reveals numerous groups where different SBOMs intersect in their enumerated dependencies. Generally, source-based SBOMs identified the most dependencies, which is expected given their access to Maven and NPM build tools listing these dependencies. Conversely, while linked to source-based results, container and release file analyses only discovered a few dependencies identified by source-based SBOMs. However, they detected unique dependencies not connected to source analyses, possibly due to access to information unavailable in the sources. E.g. in the container

phase, the installed packages of the base-images are often enumerated, which are not available in the sources phase.

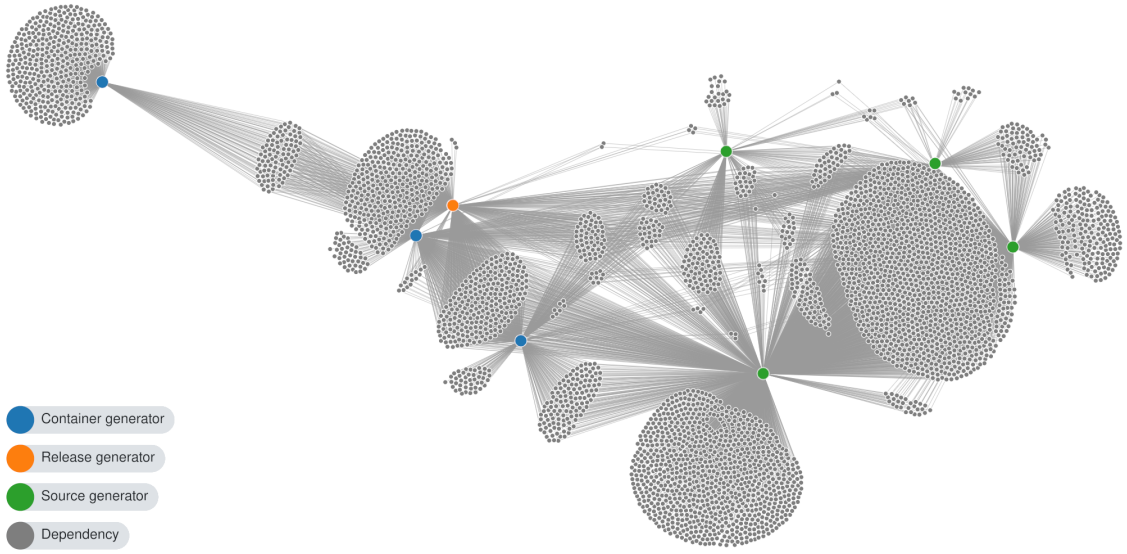

Figure 8.1: *Keycloak Dependency Plot SPDX by Ref*

While there is no package in this example where one dependency is listed in each SBOM, 31 dependencies intersect between 7 SBOMs (All but the CdxGen-Container). 26 dependencies intersect with 6 SBOMs (not the CdxGen-Container and ScanCode-Source). 88 dependencies intersect with 5 SBOMs in three different constellations. 281 dependencies intersect with 4 diffrent SBOMs in 4 diffrent constalations. 1347 dependencies intersect with 3 SBOMs in 10 different constellations. 728 dependencies intersect with 2 SBOMs in 10 different constellations. And 1582 dependencies are appearing in only one SBOM; this is happening in all SBOMs. In total, Keycloak with PURL has 4083 different dependencies listed over all SBOMs. This is true for the SPDX and CylconeDx respective results. The plot shows that relying solely on the quantity of dependencies is not a reliable metric for evaluating the coverage performance of a SBOM generator.

While eight SBOMs are participating in this analysis, no single dependency directly intersects with all of these SBOMs. Upon closer examination, there are dependencies identified by all eight SBOMs, but the PURL is represented differently. For instance, while most tools denote the PURL for the package *jackson-annotations* as *pkg:maven/com.fasterxml.jackson.core/jackson-annotations*, CdxGen, during the container scan, formats it uniquely as *pkg:maven/com.fasterxml.jackson.core/com.fasterxml.jackson.core/ jackson-annotations*. So it's fair to say that CycloneDX also identifies these dependencies. Nevertheless, such variations complicate data comparison and processing of the data.

Additionally, Syft detected the package twice in both their container and release SBOMs, once with the standard PURL and once with an altered format incorporating the Java package namespace. Processing data with such discrepancies poses challenges for later consumption of the SBOM. On the one hand, it could be argued that duplicate entries should be avoided as they artificially inflate the size of the SBOM and make it harder to

provide clear statements on the composition of a software product. On the other hand, the argument is that all available information should be included, even if it leads to duplicates, as this can enhance the SBOMs. The objective is achieved if a vulnerability is identified during a later phase of SBOM consumption due to one of the entries providing a sufficient identifier. Furthermore, if a consumer can link a vulnerability to both entries, it is still considered a single vulnerability. Hence, while these duplicates present challenges for this thesis, they are not necessarily problematic in practical application. However, this also indicates that assessing the quality of an SBOM based solely on the number of dependencies listed overlooks the quality aspect, focusing merely on the quantity of the listed dependencies. Looking at the overall intersections and comparing how much an SBOM participates in a general consensus between different tools can provide a better understanding of the quality of an SBOM.

8.2 Quantifying the Intersecting Consensus

While Figure 8.1 provides a comprehensive view of the current intersections in SBOMs generated for the same project by different tools, its ability to quantify these results for comparative analysis is limited. To address this, a metric is introduced to quantify the intersections in the data, enabling a measurable comparison and assessment of consensus among the generated SBOMs.

This metric operates on the premise that a dependency identified by a larger number of generators is more significant than one identified by a single generator. This assumption suggests that such a widely recognized dependency contributes more substantially to the overall consensus in the intersecting SBOMs, as visualized in Figure 8.1. One might argue that the value should be inversely proportional, with rare dependencies (identified by only one generator) being more critical due to their scarcity. However, this approach is not suitable for the dataset at hand, as these rare dependencies do not reflect a consensus across the SBOMs and might lead to the inclusion of false positives and maleformelly enumerated dependencies.

To assess the consensus on discovered dependencies, each dependency is assigned a weight that increases with each generator that identifies it. Therefore, dependencies recognized by multiple generators are assigned more significant values than those identified by only one.

The consensus is quantified using the following algorithm: for each SBOM, the algorithm assigns a score to each listed dependency. The score is calculated by dividing the total number of applicable generators by the number of generators identifying the dependency. This score is then aggregated across all dependencies in the SBOM, providing a measurable consensus metric.

$$consensus = \sum \frac{generator\_identified}{total\_applicable\_generators}$$

The *generators identified* represent the total number of generators/phases that enumerate the investigated dependency.

The *total applicable generators* refer to the total number of generators/phases that enumerate dependencies of the same dependency type (e.g., maven, deb, apk).

Ideally, the *generators identified* and the *total applicable generators* are identical, yielding a score of 1, which indicates a perfect consensus between all SBOMs. If not all generators/phases enumerate the dependency, the result will range between zero and one, signifying partial dependency coverage in the collective assessment of the investigated generators/phases.

An example illustrating the nature of this metric: if one out of ten generators fails to enumerate a dependency, it lowers the overall results, resulting in a maximum coverage of 90% for this package. In this scenario, the generators listing the dependency receive a score of 0.9 for listing the package, which is aggregated with the scores of all other dependencies. Conversely, the generator failing to list the dependency receives a score of 0 for this package.

To provide better context, this metrics is presented with the count of all dependencies. This count approximates the highest possible score a generator/phase could achieve if every dependency scored 100% coverage. The discrepancy is expressed as a percentage, emphasizing the conversion rate from all identified dependencies to the aggregated score representing the overall consensus.

$$conversion\_rate = \frac{consensus}{package\_count}$$

However, it is essential to mention that this metric is specifically designed to compare the performance of different tools against each other. Ecosystem types detected by only one tool in a single phase cannot be assessed using this metric, as they would consistently score as 1, representing perfect 100% coverage. On the one hand, these can be considered a wildcard, representing data points that boost the performance of a given tool without any competitors. On the other hand, this negatively impacts tools that have implemented a variety of ecosystem types not supported by others. Nevertheless, for a competitive analysis using this metric, the inclusion of other competitors is necessary. For this reason, ecosystem types detected by only one tool are excluded from the analysis conducted with this metric.

8.3 Interpreting the Results

Before examining the results in Table 8.2, it is essential to explain the overall functionality of these results. The Table lists the results, broken down by package type (identified by the PURL) and the generator/phase that enumerated them. Both dimensions are aggregated. Results for package types are listed on the left side of the table, while those aggregated by phase/generator are at the bottom. The overall aggregated result is located at the bottom left of the table. While individual results list only the metrics, the aggregated results include the total package count, the metrics itself, the conversion rate, and the sample size of applicable projects, providing further context for understanding the results.

Metrics detailed view																				
	Aggregation				Container						Release		Source							
Type	Consensus	Conversion	Count	Samples	CdxGen	MST	Syft	Tern	Trivy	CdxGen	Syft	CdxGen	GDG	MST	ScanCode	Syft	Trivy			
alpm	114	100%	114	1			114	114												
apk	901	72%	1247	46			828	498	830											
cargo	1191	36%	3303	7								773	852	238	121	916	585			
clojars	9	50%	18	2	1							8								
composer	1052	40%	2567	24	652		637		501			807	792		28	631	653			
deb	21752	73%	29391	141		18819	21588	20293	21534						9					
gem	286	29%	961	11	53		149	79	130			146.6	141	147	23	146	146			
generic	86	50%	173	128			86									0.5				
GDG	437	50%	859	110	15							437								
golang	4188	30%	13620	48	2360		2812		2389		339	2602	2995	3826	17	3003	3114			
hackage	311	100%	311	1								311				311				
hex	6	85%	7	1								6				6	4			
maven	5887	16%	35239	58	1387		2733		2653	24	589	2372	1955	1734	956	1877	1126			
npm	22730	39%	58138	89	4200		5686	578	5502			19528	21188	14853	22	21161	12392			
nuget	103	30%	343	10			81		81			39	37	6	1		4			
pypi	701	15%	4429	86	196		260	172	236	1		350	209	206	5	220	128			
rpm	6742	16%	41465	172	5514	662	1137	907	1033		2					1				
Consensus	66498	34%	192185	197	14380	19500	36153	22683	35026	24	930	27385	28172	21013	1185	28274	18156			
total					59907	25585	62622	31196	57882	112	2292	61056	67390	51333	3617	63707	34697			
count																				
Convers.					24%	76%	58%	73%	61%	22%	41%	45%	42%	41%	33%	44%	52%			
Samples					186	118	194	163	194	5	19	146	115	117	116	117	99			

Table 8.2: *Metrics detailed view*

Generally speaking, the higher the number, the better the results. For example, achieving a high conversion rate is easier with fewer competitors or when the sample size of packages or overall projects is small. Conversely, a moderate conversion rate in an enormous sample size might be considered more remarkable.

Upon examining the calculated results in Table 8.2, different ecosystems were detected across various categories. Notably, while Syft achieved the highest total package count and score, it did not have the best conversion rate in the container category. Similarly, GDG, despite having the highest total package count in the sources phase (though not the highest metric score), did not achieve the best conversion rate in this phase.

While the conversion rate offers valuable insight into performance, its accuracy is compromised by the phenomenon where tools negatively impact each other's results. For instance, a tool that identifies only half of the packages might overlap with 90% of their enumerated data. Conversely, a tool that detects all packages may show less than 50% overlap. This situation favors tools with lower coverage by artificially inflating their conversion rates. This effect distorts the true meaning of this metric and adversely affects the evaluation of tools with higher coverage.

Nevertheless, while the conversion rate is distorted, it still indicates meaningful trends. The overall conversion rate for all samples is 34%. This rate is only surpassed by the results of CdxGen for the container and release phase and the ScanCode results in the sources phase.

As previously mentioned, scanning in the release phase is challenging. While CdxGen is one of the few tools capable of reporting results in this phase, it is not surprising that these results are of lower quality.

Regarding the results of CdxGen in the container phase, the low conversion rate might be attributed to variations in the composition of the PURL identifier, leading to the inability to map the enumerated packages with other results, as already discussed.

The results of ScanCode in the sources phase can be attributed to a diverging methodology of the tooling. Unlike other dependency-centric tools, which focus on investigating a project's software supply chain, ScanCode concentrates on the project itself. For example, while investigating the Keycloak or Jenkins projects, ScanCode listed the package modules of the project instead of the imported dependencies. Consequently, ScanCode's results are often found outside the overall consensus.

It is also important to note that comparing the results of aggregated data introduces a bias due to varying feature samples. For instance, while one generator supports three different package types, another may support eight different types. Therefore, comparisons should be approached with caution.

Referring to the left side of Table 8.2, similar effects are observed in the aggregation of package types as previously identified in the aggregation of different generators and phases. A low sample size and total package count often indicates a high conversion rate, while a high package count and sample size from a variety of generators and phases typically result in a lower conversion rate.

An interesting effect can also be observed in the package types that attained a conversion rate of 50%. This is primarily due to the fact that two generators identify packages of the same type, yet there is no overlap in the enumerated packages within their respective

SBOMs. Consequently, this divergence in their findings reduces the overall average score by 50%, by the nature of the metrics. Package types such as Clojars, Generic, and GDG fall into this category.

At the lower end of the ranking are Maven and RPM, both with an average score of 16%, and PyPI with an average score of 15%. Despite their low average scores, these package managers have a significant sample size and a large number of detected dependencies. The low score for these package types is attributed to their support by most generators across all phases. However, the detected dependencies vary widely. For instance, the lowest score for Maven packages is recorded by CdxGen in the release phase with 24 dependencies, while Syft in the same phase achieves a score of 589. The scores exhibit linear growth across different phases and tools, culminating in the Syft container result with a score of 2733. The overall score is 5887, representing a conversion rate of 16%.

This evenly distributed spread not only affects the performance of individual generators but also impacts the performance of all others. This effect is most pronounced with package managers supported by SBOM generators in different phases. *Deb*, *generic*, *golang*, *maven*, *npm*, *nuget*, *pypi*, and *rpm* all have one or more generators that achieve a score of less than 1% relative to the best-scoring generator and phase for a given package type.

To partly resolve these distortions, Table 8.3 introduces an aggregation by package type grouped by phase. This separated diverging performance levels between phases.

While reducing the complexity arising from the varying phases in which the SBOMs were generated, it is evident that the overall results have improved significantly. However, RPM's conversion rate falls below the total average, as calculated in Table 8.2. This discrepancy may be attributed to the wrongful attribution of *deb* packages by CdxGen, which adversely affects the overall score.

Summarizing the findings of the investigation into the listed dependencies, it is evident that while SBOMs based on sources or containers add value to the intersecting dependency list, the release phase encounters difficulties in contributing relevant information. This discrepancy may arise from the relative ease with which an SBOM generator can identify a code repository or unpack the layers of a Docker container. In contrast, unpacking a release file without prior knowledge of the technologies employed in its creation can pose significant challenges. To address this issue, the analysis of release files could benefit from improved unpacking procedures to ensure accurate extraction of information.

While the results of the calculated metrics showed interesting insights, comparing the generated SBOM dependency results also shows that the results of the introduced metrics can only provide a distorted picture of the overall situation in this multidimensional problem. Additionally, the metrics only focuses on the intersection of dependencies, not taking into consideration all other aspects of an SBOM. Therefore, it is not recommended to use this results or approach to evaluate SBOMs outside of an academic context.

However, the metric has shown a first approach to address this multidimensional problem and relate to issues in the overall data by investigating the subsets in the data, that performed worse than the overall dataset. Thus, the impact of diverging PURL identifiers, wrongfully attributed package types, or differences in the overall methodology could be identified.

Metric phase view												
	Container				Release				Source			
Type	SUM	AVG	Count	Smpl.	SUM	AVG	Count	Smpl.	SUM	AVG	Count	Smpl.
alpm	114	100%	114	1								
apk	1247	100%	1247	46								
cargo									3303	100%	3303	7
clojars	1	50%	1	1					9	50%	17	1
composer	1366	75%	1801	18					1313	77%	1696	20
deb	14671	50%	29342	139					25	50%	49	8
gem	377	56%	666	7					236	61%	383	9
generic	86	50%	172	128					1	50%	1	1
GDG	15	100%	15	3					437	50%	859	110
golang	3014	63%	4780	43	392	98%	399	3	5666	44%	12720	33
hackage									311	100%	311	1
hex									7	100%	7	1
maven	9864	41%	23730	48	1546	77%	1988	8	6870	44%	15504	45
npm	8640	89%	9704	54					31792	56%	56009	78
nuget	143	58%	243	1					92	64%	142	10
pypi	1100	38%	2836	64	2	100%	2	2	840	40%	2055	49
rpm	13819	33%	41456	169	4	40%	10	10		33%	1	1
total	54455	47%	116107	195	1944	81%	2399	21	50900	55%	93057	162

Table 8.3: *Metric phase view*

9 SBOM License Insights

While SPDX and CycloneDX both manage their licensing information alongside their package/component lists, the approach to document the data differs.

9.1 SPDX License Features

SPDX introduces five additional values for their packages that can contain licensing information. The most frequently used one is the *Declared License* field, intended to house licensing information for a package as declared by the authors of the package. The *Concluded License* field is designed to represent the license information concluded by the SPDX document creator from the package or from alternative sources of information if the license cannot be determined. Following this is the *All Licenses Information from Files* field, intended to encompass all license information derived from files within the packages themselves. The *Comments on License* field is intended to contain comments on the licensing information. It should be used when the *Concluded License* field or the *License Information from File* field has been used to describe a license, enabling consumers of the SBOM to understand the source of the information. Lastly, there is the *Copyright Text* field, intended to hold the complete copyright text that may be contained within the package. This is particularly important if a proprietary license was used that cannot be referenced to using the SPDX license list. [45]

If a license is added as *Concluded*, *Declared*, or *from File*, the licensing information is referred to by the SPDX License List. The license list standardizes the description of known licenses, allowing them to be referenced in an SPDX SBOM. Therefore not the whole license text of each license must be included in a SBOM. Annex D of the SPDX specification describes how licenses should be referenced to and introduces the SPDX License Expression, a logic for describing the relationships between multiple licenses discovered within a package. These relationships are expressed using logical operators like *AND*, *OR*, or *WITH* to represent the license relationships. [45]

9.2 CycloneDX License Features

CycloneDX introduces a separate licensing subsection within the component specification to provide detailed licensing information. This section allows for two different formats to describe the licensing information. Either CycloneDX lists licenses in a format described by CycloneDX itself or the SPDX License Expression can be used to describe license information. In both options, CycloneDX leverages the SPDX license list to refer to well-known licenses.

Although CycloneDX does not explicitly document the source from which the license information is derived, it augments the licensing information with additional features. Firstly, there is the *bom-ref* field, which serves as a reference within the SBOM itself. Next, the *ID* field describes the license using a valid SPDX license ID from the SPDX

license list. If the license is not recognized in the SPDX license list, it can be added to the *name* field, and the full license text can be included in the *text* field. Additionally, an external reference, such as a download location, can be provided in the *url* field. While the download location primarily refers to the package itself, it may also prove useful for later license deduction for a given package.

The licensing object can encompass further information to describe license properties, such as alternate *IDs* used for identification, *licensor*, *licensee*, *purchaser*, *purchase order*, *license types* (e.g., academic, appliance, device, evaluation, perpetual), *last renewal date*, and *expiration date*. Any additional information can be included in the *properties* section of the license as a key-value mapping.

9.3 Comparison of License Features

The CycloneDX and SPDX license coverage Table 9.1 illustrates the number of dependencies identified and the proportion containing license information. While in the Dependency section, it was necessary to remove duplicate entries for mapping purposes, duplicates were not removed for investigating the licensing aspect of SBOMs. During the container phase, CdxGen and ScanCode experience an issue where converting the SBOM from CycloneDX to SPDX results in the loss of most, if not all, license information. Furthermore, it is worth noting that Tern includes significantly more licensing information in the SPDX format compared to the CycloneDX output. This is because the package information is much more enriched in Terns SPDX SBOM than in CycloneDX. In all other cases, the differences in the quantity between CycloneDX and SPDX are neglectable.

CycloneDX and SPDX License Coverage							
		SPDX			CycloneDX		
	Generator	All	Licensed	Ratio	All	Licensed	Ratio
Container	CdxGen	63466*	2365*	3%*	65035	35621	54%
	MST	28560	0	0%	28560*	0*	0%*
	Syft	72391	52496	72%	72584	52274	72%
	Tern	50149	46380	92%	32808	4783	14%
	Trivy	66025	42519	64%	60931	40271	66%
Rel.	CdxGen	112*	0*	0%*	112	0	0%
	Syft	5965	740	12%	5965	740	12%
Source	CdxGen	72968*	9560*	13%*	73249	9643	13%
	GDG	92008	64841	70%	92008*	62630*	68%*
	MST	73295	294	0%	73295*	0*	0%*
	ScanCode	14767*	0*	0%*	14901	12928	86%
	Syft	98891	2859	2%	98891	2844	2%
	Trivy	42595	3426	8%	44137	3218	7%

Table 9.1: *CycloneDX and SPDX License Coverage*

The CycloneDX license coverage table by language (Table 9.2) shows the coverage of licensing information in all packages in the CycloneDX SBOMs. The coverage is calculated

for each package type based on the PURL by dividing the total count by the count of packages that provide licensing information. The table only shows the quantity of packages with license information, but not the quality of the information that is produced by the generators. It can be seen, that CdxGen, GDG, ScanCode, Syft, Tern, and Trivy produced mixed results regarding the license coverage in all dependencies. MST produced no license information whatsoever for the listed packages in their SBOMs. This is also true for the SPDX license information that MST generated originally. The information was not lost during conversion to CycloneDx.

Moreover, the CycloneDX license coverage table by language (Table 9.2) indicates that the quality of license information is predominantly influenced by the generator used. In the context of container images, it is observed that package types associated with an OS package repository (e.g., apk or rpm) achieve high coverage. Furthermore, container and release scans yield good results for programming languages that are not compiled but build for an interpreter, such as JavaScript-npm or Python. However, the licensing data for such languages could be improved if the SBOMs were generated from sources. The most significant challenges in obtaining licensing information in containers and releases are encountered with compiled programming languages, where only binaries are distributed. For instance, while Golang supports dependency detection in binaries, no licensing information is incorporated at the container or release stage. Conversely, substantial licensing data can be added when generating SBOMs from source code.

Upon examining Table 9.3, it becomes evident which fields various generators utilize during different phases to incorporate license information into their SBOMs. The *Declared Licenses* field emerges as the most frequently employed option. Notably, MST and ScanCode were unable to append any license information to their SPDX SBOMs. In contrast, all other generators offer support for the *Declared Licenses* field and successfully enriched certain dependencies with licensing details. An exception to this pattern is observed with GDG, which opted for the *Conducted Licenses* field instead. This choice deviates from convention, as the field was not specified to contain information derived from automated processes. Trivy and the converted SPDX SBOMs from CdxGen also employ the *Conducted License* field, simultaneously recording the same information in both the *Concluded Licenses* and *Declared Licenses* fields.

While MST claims to analyze the files in some dependencies, this is only true for the dependency MST adds for the scanned project itself and therefore has no real value. No other generator claims to have analyzed the files. Also ScanCode is not claiming to have analyzed the files, while it would be able to claim it. ScanCode invests lots of resources while scanning to analyze each file. This information might not be relait by the conversion from CycloneDX to SPDX due to the fact, that CycloneDX does not store any information about files analyzed.

In the container phase, Syft was the sole generator that consistently included *download location* in their packages entries. In the sources phase, MST also attempted to provide *download location* for some of their packages; however, the quality of this information was relatively low. Specifically, MST's data contained references to RubyGems homepage 268 times, 'demo,' '.', or 'unknown' were added as download locations 15 times, and only four packages referred to actual GitHub repositories as download location.

CycloneDX Licenses Coverage										
	Container				Rel.	Source				
Languages	CdxGen	Syft	Tern	Trivy	Syft	CdxGen	GDG*	ScanCode	Syft	Trivy
alpine			99%					100%		
alpm		100%	100%							
apk		99%		99%						
autotools								90%		
bower								45%		
cargo						0%	26%	28%	0%	0%
clojars	0%					0%				
composer	99%	99%		99%		97%	36%	65%	96%	96%
conan									0%	
crane		100%								
dart								55%		
deb		88%	0%	83%				0%		
gem	0%	98%	66%	97%		0%	85%	93%	0%	0%
generic		0%							0%	
github	0%					0%				
gh-action							0%			
golang	0,08%	0%		2%	0%	0%	0%	42%	0%	37%
hackage						0%			0%	
hex						0%			0%	0%
maven	0,08%	41%		0%	29%	83%	32%	67%	0,4%	1%
none				100%		0%	0%		100%	
npm	0%	97%	99%	96%		0%	89%	70%	3%	0,3%
nuget		0%		0%		0%	75%	85%		0%
null		0%		0%						0%
osgi								0,3%		
pub						0%				
pypi	26%	93%	80%	92%		21%	73%	85%	2%	0%
rpm	83%	99%	100%	100%	100%					
suse			100%							
ubuntu			100%							

Table 9.2: *CycloneDX Licenses Coverage by Language*

Syft outperforms other generators in this regard, although it's worth noting that Syft sometimes relies on alternative identifiers instead of real download locations. Notably, 684 distinct download locations refer to what looks like a GitHub project name of the pattern [text]/[text]. 3164 distinct download locations refer to actual download locations that could be downloaded or pulled as a git repository by URL.

SPDX License Values							
	Generator	All	Declared	Concluded	Files analyzed	download location	copyright text
Container	CdxGen	63466	2365	2365	0	0	0
	Syft	72391	52368	0	0	14307	0
	Tern	50149	30933	0	0	0	41431
	Trivy	66025	42519	42519	0	0	0
Rel.	CdxGen	112	0	0	0	0	0
	Syft	5965	740	0	0	0	0
Source	CdxGen	72968	9560	9560	0	0	0
	GDG	92008	0	64841	0	0	0
	ScanCode	14767	0	0	0	0	0
	Syft	98891	2859	34	0	0	0
	Trivy	42595	3426	3426	0	0	0

Table 9.3: *SPDX License Values*

Tern stands out as the only SBOM generator that incorporates the full *license text* by default. However, it's essential to consider that including each license text significantly increases the size of the SBOM files. Additionally its worth mentioning, that Tern has detected more license texts than declared licenses.

CycloneDX License Values						
	Generator	Expression	License	License ID	License Name	License URL
Container	CdxGen	33256	2365	2202	175	2202
	Syft	240	52034	109385	60890	
	Tern		56	4783		
	Trivy	40271				
Rel.	CdxGen					
	Syft	0	740	32	708	
Source	CdxGen	61	9582	9706	783	4118
	GDG		62630	62542	88	
	ScanCode	12928				
	Syft	5	2839	2746	110	
	Trivy	3213				

Table 9.4: *CycloneDX License Values*

In examining the CycloneDX license values presented in Table 9.4, it is distinguished between listing *SPDX License Expressions* and listing *license objects* as designated by CycloneDX. Trivy and ScanCode exclusively utilize *SPDX License Expressions* in their CycloneDX SBOMs. In contrast, Tern, Syft in the release phase, and the converted GDG SBOMs solely employ CycloneDX *licensing objects*. The use of licensing objects by GDG is primarily attributed to the conversion of the SBOM format from SPDX to CycloneDX,

rather than being an inherent characteristic of how GDG generates license information. Both CdxGen and Syft employ these two methods to describe licenses in CycloneDX.

A detailed examination of the license information in CycloneDX, as illustrated in the left section of Table 9.4, reveals the values utilized within the License-type form of CycloneDX. All generators employing the licenses object utilize the *License ID* field within the object to refere to licenses. This *License ID* field also references the SPDX License Expression standard related to the SPDX license list. Additionally, the *License Name* field is employed by all generators except Tern. This field is designed to convey License information that cannot be encapsulated by an SPDX License Expression. Uniquely, CdxGen incorporates a download location in the license information. There are additional fields in the CycloneDX license object, as defined in the schema standard, which were not utilized by any generator. None of them employed the *Text*, *Licensing*, or *Properties* sections of the licenses object.

An analysis of the SPDX License Expressions used by the generators in SPDX SBOMs for concluded and declared licenses, as well as in CycloneDX SBOMs for the expressions and License IDs, indicates that all generators employed the expression logic. This logic allows to describe the interrelations of multiple licenses using expressions such as *AND*, *OR*, or *WITH*.

10 SBOM Relationship Insights

CycloneDX and SPDX both provide features to detail the relationships among components within an SBOM. However, their approaches differ. SPDX defines relationships, enabling the description of how various resources in an SPDX SBOM interconnect. It enumerates relationship types to better contextualize these connections. For instance, a package may relate to other packages or files. Similarly, the same dependency may be referenced by other packages containing it. CycloneDX, in contrast, implements a section dedicated solely to outlining the relationships between the components themselves. This focused approach eliminates the need to enumerate relationship types.

10.1 SPDX Relationships

SPDX Relationships								
	Generator	CONTAINS	DEPENDENCY_OF	DEPENDS_ON	DESCRIBES	GENERATED_FROM	HAS_PREREQUISITE	OTHER
Container	CdxGen*				202			
	MST			28355	205			
	Syft	1403954	2124		204			76797
	Tern	34994			177	30309	1210	
	Trivy	92740			202			
Release	CdxGen*				117			
	MST				117			
	Syft	2			117			5965
	Trivy				116			
Source	CdxGen*				163			
	GDG							
	MST			73121	174			
	ScanCode*				162			
	Syft				174			98891
	Trivy	64478			168			

Table 10.1: *SPDX Relationships*

In examining the aggregation of relationships listed in the SPDX SBOMs, it can be observed that only the SBOM generated by GDG does not support relationships at all.

In contrast, all other generators include at least one dependency of the type *DESCRIBES*, typically referencing the document itself to establish a root in the relationship graph.

During the container phase, Syft, Trivy, and Tern exhibit a consensus by depicting relationships with the *CONTAINS* type.

Apart from that there is no significant consensus on the types of relationships that should be used to describe the relationships within a SPDX SBOM.

It seems, that a more common practice is that a generator implements preferred relationship types to describe the relationship tree in the SBOM.

CdxGen and ScanCode, however, experience issues due to the conversion of the SBOM to SPDX format, resulting in the loss of potentially useful relationships.

10.2 CycloneDX Dependency Relationships

In analyzing the dependency relationships within the CycloneDX samples, it is observed that only CdxGen and Trivy support dependency relationship functionality. In the Sources phase, both enumerate a large part of dependencies in the relationship section. In the Container phase, CdxGen enumerated 11 dependencies while Trivy lists also a large portion of dependencies. In the release phase, only Trivy enumerates 117 dependencies in the relationship section.

This finding is particularly noteworthy given that most other tools facilitate the description of relationships in SPDX SBOMs but not in CycloneDX SBOMs. However, it is important to distinguish between SPDX also listing relationships to other resources like files or snippets, while CycloneDX focuses only on relationships between dependencies. Trivy consistently incorporates relationships in all its SBOMs. The container and source SBOMs from Trivy include meaningful relationship information. However, the release SBOMs primarily features a relationship root referring back to the SBOM itself but no other relationship description of listed components. CdxGen successfully generates meaningful relationship graphs exclusively during the source phase.

11 Results / Findings

Several findings can be derived from the investigation in this research. Depending on the reader's perspective of this paper, different findings might be addressed. Therefore, the results are grouped into five sections, addressing several perspectives.

11.1 General Results

While this thesis investigates several aspects of SBOMs the general results are grouped in the findings related to the different sections covered in this paper.

11.1.1 SBOM Standards

While both SPDX and CycloneDX implement the concept of a machine-readable SBOM specifications, they differ in their respective approaches. Each has unique methodologies and supports different use cases. Although tools exist to convert an SBOM from SPDX to CycloneDX and vice versa, this conversion is not risk-free; information loss is a potential issue. This is because, despite mapping values between the two, not all use cases of one SBOM format are necessarily represented in the other [5].

SPDX, as a specification, has been standardized as ISO/IEC 5962:2021 [2], with version 2.2.1 of SPDX. However, the current iteration is version 2.3, with version 3.0 forthcoming.

The current version of CycloneDX is 1.4 updated at the time of writing to 1.5.

CycloneDX offers not only the specification but also a diverse variety of tools for various programming languages, aiding the generation, processing, and consumption of SBOMs. Similarly, while SPDX directly provides tools pertinent to its specification, it is also integrated into a wide variety of tools and projects.

11.1.2 SBOM Generation

In summarizing the lessons learned from generating SBOMs, it becomes clear that this process is multifaceted. Notably, generating SBOMs during the Release phase did not yield meaningful results, in stark contrast to the more fruitful outcomes observed in both the sources and container phases. The methodologies for data extraction employed by tools in these phases vary significantly. For example, in the container phase, each tool must scan files within the container. However, in the sources phase, there is a divergence in strategies. Tools such as Trivy and Syft don't shell out to external build tools like Maven, opting to deploy their own parsers for extracting information. Conversely, tools like CdxGen interact directly with build tools such as Maven. This variation in approaches highlights certain security implications, particularly when SBOM generators invoke third-party applications, they need to be found trustworthy. Additionally, tools like ScanCode perform a comprehensive scan of each file in the repository, thereby providing deeper insights into the broader project architecture and extending beyond mere dependency inclusion.

Looking at the performance, the tools that implemented their parsers were the most stable. Syft had no crashes, and Trivy crashed due to issues reading pom.xml files. Followed by CdxGen, which crashed eleven times due to issues communicating with the build system. It could also be argued that the developers are responsible in this case for providing a working setup / build system CdxGen can work on which was different in this paper. ScanCode consumes a lot of processing power and time to scan all files.

Also, it is shown that the different generators support different phases and ecosystems in the software development lifecycle. All these differences in the investigated tooling make it hard to make statements about the expected coverage and depth of the enumerated results. It can not be answered easily if only direct dependencies were enumerated, or also traversing dependencies are listed. In the release or container phase, it is also not possible to differentiate between direct or traversing dependencies. For example, in Java projects, it is common to add all dependencies in a lib directory. Also, it would be necessary to access the Maven repository to enumerate traversing dependencies in Maven to resolve them. While tools like CdxGen do so natively by using the build tooling, others like Syft only do that if enabled or might not support listing traversing dependencies at all. Ecosystems like NPM, on the other hand, enumerate all packages in a package-lock file so all traversing information can be resolved. Scaling this complexity on all tools, phases, and ecosystems makes it challenging to make statements about the quality of the generated SBOMs. Zhao et al. give an excellent example in their paper working on SCA in Java projects, presenting the complexity of the Java build system and the limitations of the investigated tooling and their findings, which are partly reflected in this investigation [50].

11.1.3 SBOM Data Assessment

To generated data was assessed by applying current tools and metrics to it. The aggregation further shows that the generators implement different features and standards and point out differences between the different Standards, like diverging requirement mapping of the NTIA minimum elements and different support of the same features. E.g. more generators support mapping out relationships in SPDX than in CycloneDX. The assessment also shows that the results from the release phase are very competitive compared to the other phases, which is contradicted by the results of the dependency insights section.

11.1.4 SBOM insights

The results of the insights section show that the generated SBOMs in the release phase are not competitive with the results in other phases. Partly, they do not enumerate any dependencies in an SBOM and are therefore excluded from the assessment.

It is also shown that identifiers like PURLs are produced differently, which makes it challenging to compare the produced data. This can also make it difficult to consume the SBOM and compare the data to vulnerability databases or further enrich them.

While all other SBOMs enumerate dependencies and enrich them with different information, other use cases like License information or dependency relationships are only partly

supported. This is not surprising, considering that all of these use cases are structured around the enumerated dependencies, inheriting quality issues.

11.2 Findings from a Developer Perspective

Looking at the findings from a developer's perspective that needs to integrate an SBOM generator in a project, this result should be used with caution. While the sample projects used in this investigation represent various technologies, it is not sufficient to tell which generator is best. Based on the different features of each generator, supporting different ecosystems and phases, it might be best to look into generators that support the technologies used in the project and test them against the affected projects.

While this paper only investigates tooling that supports a wide variety of different technologies, it is noteworthy to mention that there are a lot of other tools available that are much more specialized on a single technology and integrate seamlessly with the build system of a project. For example, OWASP provides a wide variety of CycloneDX plugins that can be incorporated into a project.

Suppose the affected project is based on a single technology, for example, a project that produces a library published on a package repository. In that case, it might be best to integrate one of these plugins into the build system. Integrating tightly with the build system might produce the best results for an SBOM.

If the project is composed of a wide variety of different technologies, it might be challenging to integrate the necessary plugins into the build system and merge all the produced results into one overall SBOM. For example, a project that implements a backend server in Java, a frontend in vue.js, and incorporates some Python and C++ to implement the functionality in a performant fashion while also shipping a database for persistent would prove challenging to produce an SBOM based on plugins that integrate into each technology separately. While such projects often use build tools to manage the overall project composition, integrating SBOM plugins with them might also be challenging, and the results might be impaired. Also, the ongoing complexity of maintaining a complex setup to produce an SBOM at build time might be challenging. In such cases, it might be best to use tooling like the one investigated in this paper, which does not integrate with the build system but is able to scan the project repository and process the used technologies.

11.3 Findings from a Consumer Perspective

While this paper does not investigate the consumption of SBOMs but only the generation of such, it is worth mentioning that an SBOM can be generated based on different motives. While a consumer of an SBOM can use it for primary use cases like vulnerability or license scanning, the producer of the SBOM might have done so for other motives, like being compliant with government regulation. While incorporating the SBOM into an asset management platform, its quality should be checked.

- Does the SBOM comply with the NTIA minimum elements

- Does the SBOM enumerate packages
  - Does the packages incorporate the package version
  - Does the packages incorporate external identifiers like PURL or CPE
  - Does the packages incoperate sufficiant license information
- Are there missing packages enumerated by other tools

11.4 Findings for Generators

A wide variety of differences can be found in the SBOMs produced by the different generators. These differences represent a unique selling point to potential users or customers. This can be rooted in diverging tooling methodologies and different goals the tools try to achieve. However, they can also be based on the differences in supported platforms and environments. While all of these are legitimate reasons for differences in the produced SBOMs, differences might also be related to bugs that still need to be addressed or features not integrated yet.

Benchmarks like the NITA minimum elements, the SBOM Scorecard project, or the SBOMQS project can provide handy tools to get an idea of the quality of an SBOM. However, these tools might fail if the data is enriched with false positives. While it might be interesting to implement the generators so that all generated SBOMs comply with these requirements, the quality of the SBOM gets reduced by providing automatically generated data weakly- or unrelated to the information the field was intended for. Incorporating data based on references is optimal, but it can also be argued that providing data based on an educated guess might also be sufficient. Nevertheless, at some point, it is better to provide no information than data that is likely to be wrong and misleading.

All generators are producing an SBOM in any case. Even if no additional information is added to the SBOM. It could be argued that there should be a break condition where the generation fails due to a lack of information. This raises the question of when a generator should fail. Software developers in a competitive market prefer to produce tools that are easy to use and do not fail. Nevertheless, the user needs to know that he might have produced a low-quality SBOM. The SBOM generator MST is the only tool requiring the user to specify certain information in order to produce an SBOM.

To better address this issue, it might be sufficient to give the user the opportunity to configure a set of metadata that gets consumed while generating the SBOM so it could be enriched with this additional information that otherwise could not be retrieved. Currently, the only way to add such information is by manually manipulating the SBOM after generation.

11.5 Findings for Spesification Standardisation

SPDX and CycloneDX provide a wide range of optional use cases, making it hard to set expectations for an SBOM. It is not clear what information is contained and how detailed the description is. This paper shows the different methodologies implemented by the

generators investigated for the same standards. While most fields are optional, it should be reviewed if all these methodologies are complementary.

This issue gets even more concerning when looking at the vast differences in use cases an SBOM can incorporate. CycloneDX is not only capable of incorporating an SBOM but can also be used for other use cases like a Hardware BOM (HBOM), Software as a Service BOM (SaaSBOM), Machine Learning BOM (ML-BOM), Manufacturing BOM (MBOM), Operations BOM (OBOM), Vulnerability Exploitability Exchange (VEX), Vulnerability Disclousure Report (VDR), Bill of Vulnerabilities (BOV) or the Common Release Notes Fromat [26]. While the use case of an SBOM might still be the best known, the additional complexity makes it hard for implemented tooling to incorporate all of these use cases to process all provided information. While a BOM can incorporate all of these features, in the real world, it might only incorporate a single or several but not all of these use cases. To address these concerns, CycloneDX refers to the Software Component Verification Standard (SCVS) that implements the BOM Maturity Model. This is intended to provide a framework for specifying profiles that can used to validate a BOM to check if they meet the expected requirements. The BOM Maturity Model is still under development and was published at the time of writing. [47]

SPDX addresses this issue by introducing profiles in the upcoming SPDX version 3.0. With profiles, SPDX addresses conformance issues, workgroups that focus on developing profiles, and addresses namespaces derived from these profiles. A generator of a BOM can then indicate that a BOM supports a particular profile so a consumer of a BOM can adjust their expectations [25].

12 Limitations

This thesis presents a comprehensive analysis of various SBOM generators across different stages of the software development lifecycle. However, the investigation was conducted with certain limitations, as detailed in the following.

In this study, SBOM generators were compared from all phases, without taking into account the differences in features implemented by these generators or the variations in sample sizes provided in different stages.

Additionally, not all generators were capable of processing every sample project provided. While it could be a argued how to handle the absence of SBOMs in the evaluation, this issue further distorts the sample sizes.

When evaluating the generated SBOMs, mapping the PURLs as external identifiers across different SBOMs proved to be a significant challenge. These challenges primarily stemmed from duplicate entries created by some generators and variations in the PURL syntax, complicating the mapping process.

The selection of subject projects was based on their latest status of the projects. Although most projects adhere to a rapid publication cycle, this does not guarantee that the most recent changes in the git project's main branch align with the latest Docker container or build. This discrepancy may lead to variances in the results between Container, Release, and Source SBOMs.

Also, this thesis only compares the results from different generators against each other, but not against a single source of truth that is based on the sample projects. Therefore, it can only be argued about the consensus between the different generators. It can not be shown that the consensus aligns with the sources they are based on.

Furthermore, the relevance of this information may be transient, given the rapid pace of innovation in this field. The current version of CycloneDX is 1.5, with development for the forthcoming version 1.6 already underway. Similarly, the latest version of SPDX is 2.3, with substantial progress made towards the major 3.0 release, for which a release candidate is already available.

Although further tools emerged, that could also be considered in such an evaluation that were not available at the beginning of this thesis.

This thesis was conducted in consultation with several developers of the different SBOM generation tools investigated. They acknowledged that some behaviors reported are attributable to bugs in the generation process, with resolutions anticipated in future versions and were partly fixed already at time of writing.

13 Further work

To improve the quality of SBOMs throughout various stages of their generation in the software development lifecycle, it is proposed to implement a mapping strategy. This strategy would align versions based on commits or tags between the source code in the git repository, the software release, and the container version. Such an approach is anticipated to enhance data quality and potentially minimize the occurrence of duplicates, which often result from minor version discrepancies or changes due to the addition or removal of packages.

Furthermore, this capability could enable the creation of different SBOMs for various versions of a software product. This would provide valuable insights into the SBOM drift during the software's development lifecycle. Such insights are crucial for comprehending the evolution of software projects, their adaptation to emerging vulnerabilities, and could offer a novel approach to assessing open-source projects.

Although this thesis primarily focuses on SBOM generators supporting autonomous generation within diverse ecosystems, there is a notable deficiency in tools allowing for manual modifications of the generated SBOMs. While automation offers significant benefits, it is essential for developers or maintainers to have the ability to intervene manually. An SBOM should not only represent the automated tool's detection but also mirror the developers' comprehension of the software project. The goal is to distribute software with an accompanying SBOM that accurately depicts the project's components. Therefore, it is necessary to develop additional tools that empower developers to manually adjust an SBOM as needed. Ideally, such tools would integrate flawlessly into existing build pipelines or existing SBOM generators, applying automatically to new builds while permitting programmatic modifications by developers.

This thesis has explored the coverage of various generators by comparing their results and calculating the overlapping intersections. However, this approach is not without limitations, as there remains an inherent bias in the data due to the differences between the generators abilities. Therefore, a promising direction for future work would be to implement a 'white-box' approach. This would involve setting up a collection of sample projects with known dependencies, to which the generators can then be applied. In such an investigation, the sample set could be tailored to align with the identified capabilities of the generators, allowing for a more nuanced evaluation.

While at the beginning of this thesis the adoption of SBOMs in open source projects was marginal, since then more and more projects published SBOMs with their releases. While there are still a lot of projects that don't provide a SBOM, there are a lot more real world SBOMs available compared to a year ago. Collecting and investigating them could provide valuable insights into the current adoption of SBOMs and the challenges ahead.

14 Summary

The paper presents a comprehensive analysis of SBOM generators, focusing on their capabilities, compliance with standards, and effectiveness in enumerating and mapping software dependencies and relationships.

It highlights the pivotal role of SBOMs in today's software supply chain and their potential value. The paper also discusses regulatory compliance requirements as mandated by the American EO 14028 and the European Union's CRA, emphasizing the need for machine-readable SBOMs.

Standards for machine-readable SBOMs are investigated, namely the specifications for SPDX and CycloneDX along with their core features.

Open source projects capable of automatically analyzing a software product at different stages and generating an SBOM are presented. These generators are applied to sample projects at three different phases of the software development lifecycle, focusing on container images, release files, and the source code of the projects.

The generated SBOMs are then verified for validity and evaluated using various metrics and tools, such as the NTIA minimum elements for an SBOM, the SBOM Scorecard project, and the SBOMQS project.

Subsequently, the paper investigates the features of the generated SBOMs, examining enumerated dependencies, provided licenses, and other potential features, a SBOM can contain.

The SBOMs are mapped using the provided PURL as an external identifier to explore the overlap between different SBOMs. It was discovered that various generators produce SBOMs of differing quality and depth. Calculating a metrics to compare results was challenging and not without bias. However, the proposed metric managed to identify discrepancies in the data that might suggest potential bugs in the implemented generators by examining subsets of data whose performance was worse than the overall average.

While there are several aspects that should be addressed in this area, the multidimensionality of this problem should be acknowledged. SBOM is a top-down approach reflecting a standardized and machine-readable representation of a software product. While this representation will likely never be perfect, the development in this field is rapid, and the results improve daily.

15 Appendix

15.1 Details on SBOM Generation

The following is a detailed description of the generation process for each generator in the different phases to produce the samples for this thesis.

15.1.1 CdxGen

CdxGen, developed under the OWASP foundation and licensed under Apache-2.0, is a specialized tool for generating SBOMs conforming to the CycloneDX standard. It is a versatile tool capable of scanning a multitude of languages and supports 31 different platforms with a varying of supported package formats. Written in JavaScript, CdxGen operates independently but can integrate with package managers such as Maven or Gradle, enhancing its scanning efficacy. Additionally, it accommodates platform-specific plugins, further fine-tuning its scanning results.

Designed with flexibility in mind, CdxGen not only scans source code but is also adept at examining container images and various specialized environments, such as servers, virtual machines, or Java WAR files. While the tooling for CycloneDX format version 1.5 was published at time of writing, the SBOMs for this thesis were generated using version 1.4 to ensure compatibility with other analysis and conversion tools. Version 9.2.2 of CdxGen was employed for generating the sample data. [19]

Container Scan

CdxGen specifies the feature to scan a given container to create a CycloneDX SBOM. A set of 205 Docker containers were scanned, resulting in the successful generation of SBOMs for 202 containers. As expected, CdxGen did not generate a SBOM for the scratch Tag, claiming, the container is not available. Also CdxGen failed to generate a SBOM for the containers of pegasus and sonarqube due to internal issues reading the filesystem of the docker container. The analysis of the generated SBOMs revealed that 186 contained at least one dependency, 182 included more than ten dependencies, and 158 featured over one hundred dependencies. The following command was used to generate the samples (see listing 15.1).

```
1 cdxgen [container] --deep --spec-version 1.4 -r -o
2 [output.file] -t docker
```

Listing 15.1: *CdxGen container comand*

Release Scan

While CdxGen is not explicitly specified as being capable of scanning release files, it can be applied to such tasks by utilizing its ability to scan the files in a given directory.

Out of 117 projects providing release files in their GitHub repositories, CdxGen was able to generate an SBOM for each project. However, the quality of these SBOM may be questionable, as only 5 of them were found to actually detect dependencies within the release files. Specifically, for Jenkins, CdxGen identified 92 dependencies; for Convertigo, 17 dependencies were detected; and for Sentry, ArchiveBox, and Babashka, only one dependency each was found. The remaining 112 SBOMs did not contain any detected dependencies of the subject projects. The following command was utilized to generate the SBOMs (see listing 15.2).

```
1 cdxgen --deep --spec-version 1.4 -r -o [output.file]
2 [release/path]
```

Listing 15.2: *CdxGen Command for Release Scanning*

Source Scan

CdxGen can be applied to the source files of a project to extract information about dependencies from the build systems. CdxGen's documentation indicates that the presence of build tools, such as Apache Maven, Gradle, or SBT, can enhance the quality of the results. The tool relies directly on these build systems to determine dependencies, which can improve the accuracy of the generated SBOMs but may also introduce complexities or issues during the build process. Due to unforeseen responses from the underlying build systems, CdxGen's generation process failed in 11 out of 174 cases. Of the successful scans, 146 SBOMs contained at least one dependency, 113 contained 10 or more dependencies, and 73 listed over 100 dependencies. The following command was utilized to generate the SBOMs (see listing 15.3).

```
1 cdxgen --deep --spec-version 1.4 -r -o [output.cdx.json]
2 [source/path]
```

Listing 15.3: *CdxGen Command for Source Scanning*

15.1.2 GitHub Dependency Graph

GDG provides the functionality to generate an SBOM in the form of an SPDX JSON file. This feature is integrated within the Dependency Graph, accessible from the insights tab of a GitHub repository. GitHub enables users to export an SBOM for the current main branch of the repository, indicating that the generation process is entirely managed by GitHub. However, a notable limitation is the absence of an option to retrieve an SBOM for a different branch, commit, release, or tag. As of now, the GitHub Dependency Graph supports 13 different package managers across 17 different programming languages. [20]

Source Scan

Of the 174 GitHub repositories analyzed, SBOMs generated by GDG were available for 171 projects. GitHub has this feature enabled by default, but it can be disabled by the project maintainer, making the feature unavailable. Consequently, SBOMs were not available

for the projects TomEE, Friendica, and ClamAV. All available SBOMs contained at least one dependency. Furthermore, 126 SBOMs included 10 or more dependencies, and 75 contained over 100 dependencies.

15.1.3 Microsoft SBOM Tool

The MST SBOM Tool is an SBOM generation tool released by Microsoft under the MIT license. The tool is developed in C# [21] and is built upon the Microsoft Component Detection project, which predominantly utilizes C# and supports 11 different ecosystems. For component detection within Linux environments, such as Debian, Alpine, or Fedora, it leverages Syft [18]. The tool exclusively produces SBOMs in the SPDX format.

Unique to the MST is its requirement for specifying certain parameters that are incorporated into the resulting SBOM. These parameters include the package name, supplier, namespace base URI, and package version. For the production of the sample data, version 1.1.8 of the MST was employed.

Container Scan

The MST was successfully applied to all 205 subject projects. Notably, it was capable of generating an SBOM for the *scratch* tag, a special reserved tag in the Docker ecosystem that enables the building of a container from scratch. Each generated SBOM contains at least one dependency. Of these SBOMs, 159 included more than 10 dependencies, and 122 documented more than 100 dependencies. The following command was employed to generate the sample SBOMs (see listing 15.4).

```
1 sbom-tool generate -di [container] -m
2 [output/path]-pn [packageName] -pv [PackageVersion]
3 -ps [supplier] -nsb [namespace URI]
```

Listing 15.4: *MST container command*

Release Scan

The MST was capable of generating an SBOM for all 117 subject projects during the release phase. Although a SBOM was produced for every release project, it is notable that each SBOM documented only a single dependency referring to the scanned project and not a dependency. The following command was employed to generate the sample SBOMs (see listing 15.5).

```
1 sbom-tool generate -b [output/path] -bc
2 [input/path] -pn [packageName] -ps [supplier]
3 -nsb [namespace URI] -pv [PackageVersion]
```

Listing 15.5: *MST release command*

Source Scan

The MST successfully generated an SBOM for each of the 174 subject projects during the source phase. Every SBOM listed at least one dependency. Of these, 99 SBOMs identified more than 10 dependencies, while 64 SBOMs enumerated more than 100 dependencies. The following command was employed to generate the sample SBOMs (see listing 15.6).

```
sbom-tool generate -b [output/path] -bc
[input/path] -pn [packageName] -pv [packageVersion]
-ps [supplier] -nsb [namespace URI]
```

Listing 15.6: *MST source command*

15.1.4 ScanCode Toolkit

ScanCode Toolkit is a tool developed and maintained by NexB under an Apache-2.0 license. It is a part of a broader ecosystem offered by the company for products in the field of SCA and is primarily written in Python. The ScanCode Toolkit claims support for 116 different programming languages, build tools, and package formats. While ScanCode can generate SBOMs in SPDX and CycloneDX formats, it does not support the SPDX format in JSON. Unfortunately we were not able to implement a conversion from the RDF supported SPDX output generated by ScanCode to the JSON SPDX output. Therefore, for this thesis, the CycloneDX output was converted to SPDX. It is also noteworthy that ScanCode also produced an invalid metadata properties object, rendering it incompatible for use with other tools, for instance, in converting the SBOMs from CycloneDX to SPDX. This metadata object was fixed before conversion from SPDX to CycloneDx. ScanCode Toolkit version 32.0.6 was employed to generate the sample data. [23]

Source Scan

ScanCode Toolkit was successful in generating SBOMs for 162 out of the 174 subject projects. The generation process with ScanCode is resource-intensive, and the procedure was carried out using 8 threads. All of the 12 projects that did not yield an SBOMs reached the 1-hour timeout threshold, leading to the termination of the process. The affected projects include mono, node, mysql, open-liberty, openjdk, mongo, percona, amazoncorretto, arangodb, gcc, odoo, and sapmachine. Among the completed SBOMs, 126 listed at least one dependency, 33 contained 10 or more dependencies, and 10 SBOMs documented more than 100 dependencies.

```
scancode -clpi -n 8 --cyclonedx [output.cdx.json] [sources]
```

Listing 15.7: *ScanCode source command*

15.1.5 Syft

Syft is a SBOMs generation tool primarily written in Go and released under the Apache-2.0 license. Developed and maintained by Anchore, it is part of their suite of products aimed

at enhancing software supply chain security. As a CLI tool, Syft facilitates the generation of SBOMs from container images and file systems. The tool supports up to 23 different ecosystems and provides output in various formats, including CycloneDX and SBOM. While Syft can convert between different SBOMs formats, this functionality is currently marked as experimental [18]. Additionally, Syft has the capability to export SBOMs in multiple formats simultaneously, as demonstrated in the commands 15.8, 15.9, and 15.10. For the creation of the sample data in this study, version v0.85.0 of Syft was utilized.

Container Scan

Syft successfully generated SBOMs for 204 out of the 205 container images examined. The exception was the *scratch* tag, for which a pointer exception was logged instead of producing an SBOM. Among the 204 SBOMs produced, 196 list at least one dependency, 193 include at least 10 dependencies, and 170 feature more than 100 dependencies. The command used to generate these samples is shown in Listing 15.8.

```
1 syft [container] -o spdx-json=[output.spdx.json] -o
2 cyclonedx-json=[output.cdx.json]
```

Listing 15.8: *Syft container command*

Release Scan

Syft managed to generate SBOMs for all 117 releases from the selected projects. However, 98 of these SBOMs did not list any dependencies. Of the remaining SBOMs, 19 include at least one dependency, 10 document at least 10 dependencies, and 3 detail more than 100 dependencies. The command utilized for generating these samples is provided in Listing 15.9.

```
1 syft [release.path] -o spdx-json=[output.spdx.json] -o
2 cyclonedx-json=[output.cdx.json]
```

Listing 15.9: *Syft release command*

Source Scan

Syft successfully generated an SBOM for all 174 subject projects from their source code. Out of the generated SBOMs 118 contain at least one dependency. Of these, 103 SBOMs include more than 10 dependencies, and 74 have more than 100 dependencies. The command used to generate the samples is presented in Listing 15.10.

```
1 syft [sources.path] -o spdx-json=[output.spdx.json] -o
2 cyclonedx-json=[output.cdx.json]
```

Listing 15.10: *Syft source command*

15.1.6 Tern

Tern is a tool designed for inspecting software packages within containers that also has the capability to generate SBOMs. It is primarily developed in Python and backed by the community. The tool is distributed under the BSD-2-Clause License. Although its primary function is to scan container images, it can be enhanced with the ScanCode Toolkit to yield more detailed information. Tern supports several output formats, including CycloneDX and SPDX. For generating the sample data, version 2.12.1 of Tern was utilized. [24]

Container Scan

Tern succeeded in generating an SBOM for 177 of the 205 subject projects. Each SBOM contained at least one listed dependency. Among these, 164 SBOMs included more than 10 dependencies, and 145 listed more than 100 dependencies. The following command was used to generate the samples (see listing 15.11).

```
1 tern report -f spdxjson -i [container]
2 -o [output.spdx.json]
3
4 tern report -f cyclonedxjson -i [container]
5 -o [output.cdx.json]
```

Listing 15.11: *Tern container command*

15.1.7 Trivy

Trivy is a comprehensive security scanner that is published under the Apache-2.0 license and is developed and maintained by Aqua Security. Trivy has the capability to scan container images, filesystems, Git repositories, virtual machine images, Kubernetes clusters, and AWS environments. It is not limited to the generation of SBOMs but also facilitates the search for CVEs, infrastructure as code (IaC) issues, analysis searching for misconfigurations, and detecting sensitive information, in addition to identifying software licenses. Predominantly written in Go, Trivy supports twelve different programming languages, seventeen different operating system distributions, and six different IaC and configuration languages. Furthermore, Trivy offers support for output in various SBOM formats, including SPDX and CycloneDX. For the generation of the sample data in this study, Trivy version 0.43.1 was utilized. [17]

Container Scan

Trivy was capable of generating an SBOM for 204 of the 205 subject project containers. It appropriately failed to generate an SBOM for one container, specifically the *scratch* tag, providing an error message indicating that the image was not available. All the scanned containers contained at least one dependency. Among these, 194 SBOMs listed more than 10 dependencies, and 167 listed more than 100 dependencies. The commands used to generate the sample SBOMs are listed below (see listing 15.12).

```
trivy image --format spdx-json --output [output.spdx.json]
[container]

trivy image --format cyclonedx --output [output.cdx.json]
[container]
```

Listing 15.12: *Trivy container command*

Release Scan

Trivy was able to generate an SBOM for 116 of the 117 subject project releases. Trivy crashed after 46 minutes while generating a SBOM for the jetty projects release files without logging a reason. Each of the generated SBOMs contained only one dependency. The commands used to generate the samples are listed below (see listing 15.13).

```
trivy fs --format spdx-json --output [output.spdx.json]
[release.path]

trivy fs --format cyclonedx --output [output.cdx.json]
[release.path]
```

Listing 15.13: *Trivy release command*

Source Scan

Trivy successfully generated an SBOM for 168 out of the 174 subject project sources. It failed to generate an SBOM for the projects Keycloak, Silverpeas, TomEE, and Nuxeo due to a fatal error encountered while reading a pom.xml file. For the projects Xwiki and Geonetwork, the generation process exceeded the timeout of 1 hour. All of the SBOMs contained at least one dependency. Of the generated SBOMs, 89 listed more than 10 dependencies, and 58 listed more than 100 dependencies. The following commands were used to generate the samples (see listing 15.14).

```
trivy fs --format spdx-json --output [output.spdx.json]
[sources.path]

trivy fs --format cyclonedx --output [output.cdx.json]
[sources.path]
```

Listing 15.14: *Trivy Source Command*

15.2 Detailed Data on SBOM Assessment

NTIA minimum elements SPDX coverage								
	Generator	*Supplier Name*	*Component Name*	*Version of Component*	*Other unique Identifiers*	*Depencency Relationship*	*Author_of SBOM_data*	*Timestamp*
Container	CdxGen*	0%	100%	100%	100%	100%	100%	100%
	MST	100%	100%	100%	100%	100%	100%	100%
	Syft	0%	100%	99%	100%	100%	100%	100%
	Tern	65%	100%	100%	100%	100%	100%	100%
	Trivy	99%	100%	99%	100%	100%	100%	100%
Release	CdxGen*	0%	100%	100%	100%	100%	100%	100%
	MST	100%	100%	100%	100%	100%	100%	100%
	Syft	0%	100%	99%	100%	100%	100%	100%
	Trivy	0%	100%	0%	100%	100%	100%	100%
Source	CdxGen*	0%	100%	100%	100%	100%	100%	100%
	GDG	100%	100%	100%	100%	0%	100%	100%
	MST	100%	100%	100%	100%	100%	100%	100%
	ScanCode*	0%	100%	97%	100%	100%	100%	100%
	Syft	0%	100%	84%	100%	100%	100%	100%
	Trivy	97%	100%	96%	100%	100%	100%	100%
	Total	49%	100%	97%	100%	93%	100%	100%

Table 15.1: *NTIA minimum elements SPDX coverage*

Table 15.1 presents the coverage of the NTIA minimum elements on the SPDX SBOM samples. For the mapping between the NTIA minimum elements and the SPDX schema, the published mapping table by SPDX was used [46].

NTIA minimum elements CycloneDX coverage								
	Generator	*Supplier Name*	*Component Name*	*Version of Component*	*Other unique Identifiers*	*Depencency Relationship*	*Author_of SBOM_data*	*Timestamp*
Container	CdxGen	0%	100%	100%	100%	100%	100%	100%
	MST*	0%	100%	100%	100%	0%	0%	100%
	Syft	0%	100%	99%	100%	0%	0%	100%
	Tern	0%	100%	100%	100%	0%	0%	100%
	Trivy	52%	100%	100%	99%	100%	0%	100%
Release	CdxGen	0%	100%	100%	100%	100%	100%	100%
	MST*	0%	100%	100%	100%	0%	0%	100%
	Syft	0%	100%	100%	100%	0%	0%	100%
	Trivy	0%	0%	0%	0%	0%	0%	100%
Source	CdxGen	0%	100%	100%	100%	100%	100%	100%
	GDG*	0%	100%	96%	100%	0%	0%	100%
	MST*	0%	100%	100%	100%	0%	0%	100%
	ScanCode	0%	100%	97%	100%	100%	0%	100%
	Syft	0%	100%	84%	100%	0%	0%	100%
	Trivy	0%	100%	95%	95%	100%	0%	100%
	Total	20%	100%	97%	99%	46%	19%	100%

Table 15.2: *NTIA minimum elements CycloneDX coverage*

Table 15.2 presents the coverage of the NTIA minimum elements on the CycloneDX SBOM samples. For the mapping between the NTIA minimum elements and the CycloneDX schema, the published mapping table by CycloneDX was used [34].

SPDX SBOM Scorecard								
	Generator	*Supported*	*Total*	*Compliance*	*Package-Ident.*	*Package-versions*	*Package-licenses*	*Creation-info*
Container	CdxGen*	100%	59%	100%	0%	92%	3%	100%
	MST	100%	60%	100%	0%	100%	0%	100%
	Syft	100%	92%	100%	93%	95%	72%	100%
	Tern	100%	71%	100%	0%	100%	59%	100%
	Trivy	100%	72%	100%	0%	93%	66%	100%
Release	CdxGen*	100%	40%	100%	0%	4%	0%	100%
	MST	100%	60%	100%	0%	100%	0%	100%
	Syft	100%	47%	100%	15%	15%	8%	100%
	Trivy	100%	40%	100%	0%	0%	0%	100%
Source	CdxGen*	100%	60%	100%	0%	89%	15%	100%
	GDG	100%	59%	100%	0%	78%	31%	80%
	MST	100%	60%	100%	0%	100%	0%	100%
	ScanCode*	100%	48%	100%	0%	42%	0%	100%
	Syft	100%	66%	100%	66%	57%	6%	100%
	Trivy	100%	52%	100%	0%	52%	10%	100%
	Total	100%	61%	100%	13%	73%	21%	99%

Table 15.3: *SPDX SBOM Scorecard*

Table 15.3 presents the results of the sample SPDX SBOMs applied to the SBOM Scorecard project [31].

CycloneDX SBOM Scorecard								
	Generator	*Supported*	*Total*	*Compliance*	*Package-Ident.*	*Package-versions*	*Package-licenses*	*Creation-info*
Container	CdxGen	100%	84%	100%	85%	85%	57%	93%
	MST*	100%	76%	100%	99%	99%	0%	79%
	Syft	100%	87%	100%	88%	89%	65%	93%
	Tern	100%	79%	100%	89%	88%	21%	98%
	Trivy	100%	1%	5%	0%	0%	0%	0%
Release	CdxGen	100%	41%	100%	4%	4%	0%	100%
	MST*	100%	77%	100%	100%	100%	0%	80%
	Syft	100%	48%	100%	16%	15%	8%	100%
	Trivy	100%	0%	0%	0%	0%	0%	0%
Source	CdxGen	100%	69%	100%	72%	72%	14%	82%
	GDG*	100%	65%	100%	80%	59%	14%	64%
	MST*	100%	69%	100%	86%	86%	0%	68%
	ScanCode	100%	75%	100%	77%	41%	60%	99%
	Syft	100%	56%	100%	50%	41%	4%	82%
	Trivy	100%	1%	4%	0%	0%	0%	0%
	Total	100%	57%	81%	59%	55%	18%	69%

Table 15.4: *CycloneDX SBOM Scorecard*

Table 15.4 presents the results of the sample CycloneDX SBOMs applied to the SBOM Scorecard project [31]

SPDX SBOMQS, Qualtity Metrics for SBOMs								
	Generator	*Supported*	*Total*	*Structural*	*NTIA-min.*	*Semantic*	*Quality*	*Sharing*
Container	CdxGen*	100%	6.20	10.00	8.23	1.78	3.33	10.00
	MST	100%	6.37	10.00	8.89	3.33	2.56	10.00
	Syft	100%	8.31	10.00	9.40	6.22	6.91	10.00
	Tern	100%	7.42	10.00	9.38	5.63	4.40	10.00
	Trivy	100%	8.17	10.00	9.33	4.43	7.29	10.00
Release	CdxGen*	100%	4.40	10.00	4.47	1.67	1.49	10.00
	MST	100%	6.82	10.00	10.00	3.33	2.86	10.00
	Syft	100%	4.90	10.00	5.04	2.41	2.19	10.00
	Trivy	100%	5.68	10.00	7.14	1.67	2.86	10.00
Source	CdxGen*	100%	6.38	10.00	8.12	2.14	3.86	10.00
	GDG	100%	6.48	10.00	8.27	4.38	3.07	10.00
	MST	100%	6.54	10.00	9.14	3.33	2.86	10.00
	ScanCode*	100%	5.57	10.00	7.11	1.67	2.54	10.00
	Syft	100%	6.24	10.00	7.07	3.18	4.04	10.00
	Trivy	100%	6.57	10.00	7.89	2.00	4.75	10.00
	Total	100%	6.45	10.00	8.14	3.28	3.85	10.00

Table 15.5: *SPDX SBOMQS, Qualtity Metrics for SBOMs*

Table 15.5 presents the results of the sample SPDX SBOMs applied to the SBOMQS project [32].

CycloneDX SBOMQS, Qualtity Metrics for SBOMs								
	Generator	*Supported*	*Total*	*Structural*	*NTIA-min.*	*Semantic*	*Quality*	*Sharing*
Container	CdxGen	100%	7.00	10.00	7.15	5.35	6.86	0.00
	MST*	100%	5.36	10.00	7.14	3.33	2.56	0.00
	Syft	100%	7.64	10.00	7.14	5.17	8.94	0.00
	Tern	100%	5.95	10.00	5.85	4.38	5.26	0.00
	Trivy	100%	7.26	7.50	9.28	6.16	6.60	0.00
Release	CdxGen	100%	3.51	10.00	3.03	1.74	1.55	0.00
	MST*	100%	5.45	10.00	7.14	3.33	2.86	0.00
	Syft	100%	5.25	10.00	5.92	3.38	3.41	0.00
	Trivy	100%	4.55	7.50	5.71	3.33	2.86	0.00
Source	CdxGen	100%	6.48	10.00	7.45	4.81	5.14	0.00
	GDG*	100%	6.00	10.00	6.84	4.30	4.47	0.00
	MST*	100%	5.45	10.00	7.14	3.33	2.86	0.00
	ScanCode	100%	6.38	10.00	5.68	5.03	6.49	0.00
	Syft	100%	6.00	10.00	6.50	3.47	5.16	0.00
	Trivy	100%	5.70	7.50	7.31	3.66	4.76	0.00
	Total	100%	6.01	9.5	6.77	4.19	4.90	0.00

Table 15.6: *CycloneDX SBOMQS, Qualtity Metrics for SBOMs*

Table 15.6 presents the results of the sample CycloneDX SBOMs applied to the SBOMQS project [32].

15.3 Excurs Dependency insights

15.3.1 Differences in Dependency Enrichment

Comparing the results generated by the different tools, it becomes evident that while all tools adhere to the respective standardizations, the results vary significantly by how differently they enrich the dependencies. These variations are influenced by the software development lifecycle phase and the SBOM generator's capabilities in capturing all functionalities.

An illustrative example involves comparing SBOMs generated by different tools across various stages of the software development lifecycle based on the same sample project. Detailed comparisons of these findings are published separately in a blog post [6]. Notably, all tools across all development phases consistently identified no single dependency. The differences in the investigated tooling methodologies are too significant, so no overall example could be picked to investigate the differences. Consequently, different dependencies were selected for illustrative purposes. For instance, Apache Commons-Compress, used in the SBOMs generated for the Jenkins project, was not detected by ScanCode, Tern, and MST Container Scan. However, Apache Commons-Compress was detected by all other generators in all phases and is, therefore, one of the most extensive intersecting dependencies among all generated samples. Alternative examples were chosen for the missing generators.

The results generated by CdxGen show similar results for the detected Apache Commons-Compress dependency in both Container and Release scans. However, significant differences are observed compared to the results derived from the Source scan. This might be due to the fact that the sources provide a reference to the Maven repository, where the SBOM can be enriched with additional information. In the Sources, the license information, hashes, and description are the main advantages over the container and release scans. CdxGen does not support SPDX output. For this reason, the SBOMs were converted based on the CycloneDX SBOM.

GDG provides an SPDX SBOM with the basic information needed for the major use cases of checking the version of a dependency together with a valid PURL to identify the dependency and retrieve the license information.

Unfortunately, MST was not able to detect the Apache Commons-Compress dependency in the container scan but only in the sources. MST was also not able to detect any dependencies in the release files. Nevertheless, the provided information in the container and sources phase is semantically identical. While MST provides a valid PURL and Version for their enumerated dependencies, no license information is provided in the examples.

The ScanCode SBOMs were generated based on CycloneDX and converted to SPDX. ScanCode was also not able to detect Apache Commons-Compress in the source code. Therefore, another dependency was picked as an example. While the information looks well enriched, the data quality could be better. Optional values like comments or copyright are initialized with *null*, and the version information is provided by a variable that might be based on a Maven abstraction to define versions in a unified way across all POM files.

Also, the depth of the scan conducted to detect dependencies needs to be improved. Only Jenkins-related libraries are listed, but no maven-imported dependencies.

Syft provides a well-enriched experience in both SPDX and CycloneDx. It supports both output formats and provides lots of information, such as several external reference locators of type PURL and CPE. They also made transparent how they collected this information. The sources are based on the POM file, while the container and release are derived from the installed Java archives that could be discovered during analysis. However, while Syft can also read the POM file, it does not use this information, like CdxGen, to enrich the SBOM with additional information from the package registry, such as hashes or licensing information (This feature could be enabled in Syft, but is disabled in the standard configuration). Also, the container and release scan refer to the Apache 2.0 license by URL and not by identifier. This makes it especially challenging to process the SPDX results. While the URL to the license is readable in the CycloneDX output, it is malformed in the SPDX output.

While Tern supports both the output in SPDX and CycloneDx, the experience with SPDX is much better than the CycloneDX output. The SPDX output provides a comment, supplier, and copyright Text, all of which are missing in the CycloneDX output. The CycloneDX result only contains name, PURL, type, and version of the dependency.

Trivy successfully identified dependencies in the containers and source phase but not in the release phase. Although it detected the majority of fundamental details, the license information was the only aspect missing in this case. It was also the only tool that generated the field *primaryPackagePurpose*. While it makes the information source transparent in the CycloneDX files based on the dependency detected, such as the JAR or POM files, this information is not added in the SPDX file.

15.3.2 Jenkins Example

In Table 15.7, the distribution of detected resources within the Jenkins sample project is presented. This distribution is segregated according to the different phases of the software development lifecycle during which the SBOM was generated. It is further categorized by the type of resources listed as dependencies. For instance, Java resources based on Maven were detected in all phases, with a predominant presence in the source phase. This discrepancy can be attributed to several factors. During the Container and Release phases, SBOM generators are required to analyze each JAR file to extract dependency information. In contrast, during the source phase, the build system, such as Maven, can be directly queried for this information, leading to different methodologies in extracting dependency data. Moreover, source files may reference dependencies that are not included in the final product, such as test dependencies like JUnit.

The observed variance in the detected quantity of dependencies is largely explainable by the depth at which tools search for dependencies. ScanCode only listed packages produced by the project itself, excluding imported dependencies. Tools like Syft and GDG disclose first-level dependencies, whereas CdxGen also enumerates transitive dependencies. However, the distinction between transitive and non-transitive dependencies becomes blurred in Java projects when analyzing release files or container images, as all

Jenkins SPDX PURL grouped by type																
Type	Container					Release				Source					
	CdxGen	MST	Syft	Tern	Trivy	CdxGen	MST	Syft	Trivy	CdxGen	GDG	MST	ScanCode	Syft	Trivy
apk				14											
deb		156	156	156	156										
generic			1												
github										8					
ghactions											11				
golang			26												
maven	94		300		167	92		277		252	104	240	8	174	80
npm										662	721	662	1	661	13
oci					1										
rpm	229							2							
swid		1					1					1			

Table 15.7: *Jenkins SPDX PURL grouped by type*

utilized libraries are stored collectively. As a result, tools like Syft and Trivy inadvertently catalog transitive dependencies in the release and container SBOMs, whereas they are not included in the sources.

While several scanners successfully identified Java dependencies in release or container scans, none were able to detect npm packages in the container or release files.

Despite JavaScript not being compiled into binary, it remains challenging to ascertain precisely which libraries were utilized in building a JavaScript application.

Additionally, Syft was the only scanner that detected Go packages in the container image. Upon manual verification of the asserted references, binaries based on Go were indeed present in the container's base image.

Both CdxGen and GDG listed GitHub actions in their results. Although they refer to identical actions, such as *githubaction/checkout*, they employ different typenames to describe them in the PURL.

While all container-based SBOMs list the packages that are installed on the base image, tern is the only one to add apk / debian packages to the SBOM. While all the generators found the same DEB packages, only CdxGen referred to them as RPM packages. While DEB is related to the Debian ecosystem and therefor to the base image of the container,

Jenkins SPDX PURL grouped by type cleaned of duplicates																
Type	Container					Release				Source						
	CdxGen	MST	Syft	Tern	Trivy	CdxGen	MST	Syft	Trivy	CdxGen	GDG	MST	ScanCode	Syft	Trivy
apk				14											
deb		156	156	156	156										
generic			1												
github										8					
ghactions											11				
golang			26												
maven	94		227		144	92		217		252	104	240	8	132	80
npm										662	721	662	1	661	13
oci					1										
rpm	229							2							
swid		1					1					1			

Table 15.8: *Jenkins SPDX PURL grouped by type cleaned of duplicates*

RPM refferes to the RedHad package manager for RedHad Enterprice Linux. Validating the results CdxGen lists the same image ID hash as the other SBOMs do, which refers to the Debian jenkins container. Looking into the list of dependencies, CdxGen listed to packages like *pkg:rpm/apt*. Packages like apt do not exist in the RPM ecosystem but in the Debian world. Therefor this behavior must be related to a bug in CdxGen. It was also checked, that this error was not introduced by the convertion from CycloneDX to SPDX, but the PURL with the rpm types are also listed in the original CycloneDX SBOMs.

Comparing the list in Table 15.7 with the list in Table 15.8, where duplicate dependencies are removed by PURL, it is noteworthy that dependencies with different versions were not eliminated in this tables. It can be seen that, while all dependencies found remain exactly the same, the Maven dependencies from Syft in all phases and the Trivy dependencies from Maven in the container phase are reduced due to the removal of duplicates. This highlights the different methodologies that can be applied regarding duplicate dependencies in a SBOM.

All SPDX PURLs grouped by type																
Type	Container					Release				Source					
	CdxGen	MST	Syft	Tern	Trivy	CdxGen	MST	Syft	Trivy	CdxGen	GDG	MST	ScanCode	Syft	Trivy
clojars	✓									✓					
github[1]	✓									✓	✓				
gem	✓		✓	✓	✓					✓	✓	✓	✓	✓	✓
pypi	✓		✓	✓	✓	✓				✓	✓	✓	✓	✓	✓
composer	✓		✓		✓					✓	✓		✓	✓	✓
golang	✓		✓		✓			✓		✓	✓	✓	✓	✓	✓
npm	✓		✓	✓	✓					✓	✓	✓	✓	✓	✓
maven	✓		✓		✓	✓		✓		✓	✓	✓	✓	✓	✓
rpm	✓	✓	✓	✓	✓			✓						✓	
swid		✓					✓					✓			
deb		✓	✓	✓	✓								✓		
cran			✓												
alpm			✓	✓											
generic			✓											✓	
nuget			✓		✓					✓	✓	✓	✓		✓
apk			✓	✓	✓										
oci					✓										
none					✓										
hex										✓				✓	✓
pub										✓					
hackage										✓				✓	
cargo										✓	✓	✓	✓	✓	✓
bazel													✓		
haxe													✓		
opam													✓		
jar													✓		
ha1e													✓		
bower													✓		
autotools													✓		
dart													✓		
osgi													✓		
alpine													✓		
conan														✓	
pkg:														✓	
| ✓ < 10 | ✓ < 100 | ✓ < 1.000 | ✓ <10.000 | ✓ < 100.000 |															
[1] Github and Githubactions were consolidated in this table.															

Table 15.9: *All SPDX PURLs grouped by type*

15.3.3 Generalising to all Samples

Generalizing the findings from individual examples to an evaluation across all samples, Table 15.9 presents a comprehensive list of PURL types identified across all projects. These types are categorized based on the different phases and the generators that detected them. The check icons in the table are color-coded to represent the quantity of packages identified by each generator for a particular type.

The majority of the findings were obtained during the sources phase. While ScanCode may not delve as deeply into the dependency chain, as previously demonstrated, it boasts support for the broadest range of diverse ecosystems. Beyond ScanCode, there is a general consensus among the different generators regarding the tooling supported in the sources phase.

Although there are tools that stand out in the data for their support of a specific programming language or build system that is not covered by any other tool.

Investigating the Container phase, it becomes evident that MST's primary focus lies in the detection of packages associated with the base image of the container. On the other hand, Tern is also centered around analyzing the setup of the base image but exhibits the ability to identify several related packages such as Python, Ruby, and NPM. Tern's success is attributed to its capability to identify packages that were not only shipped with the project but also installed within the container itself. Taking this ability a step further, tools like CdxGen, Syft, and Trivy delve deeper into the examination, enabling the detection of packages spanning various programming languages and tools.

In examining the outcomes of the release phase, the most sobering findings can be observed. MST supplied only a SWID tag, which was not detected but defined by the MST. This tool generates a SWID tag in the form of a dependency for each project. However, these tags refer back to the project itself rather than to a related dependency. Only CdxGen and Syft yielded meaningful results. The most reliable findings pertained to the Java-related Maven ecosystem.

List of Figures

List of Listings

List of Tables

Bibliography

[1] 14:00-17:00. *ISO/IEC 19770-2:2015*. ISO. URL: https://www.iso.org/standard/65666.html (visited on 11/24/2023).

[2] 14:00-17:00. *ISO/IEC 5962:2021*. ISO. URL: https://www.iso.org/standard/81870.html (visited on 03/24/2023).

[3] *Are SBOMs Any Good? Preliminary Measurement of the Quality of Open Source Project SBOMs*. URL: https://www.chainguard.dev/unchained/are-sboms-any-good-preliminary-measurement-of-the-quality-of-open-source-project-sboms (visited on 03/18/2023).

[4] *Are SBOMs Good Enough for Government Work?* URL: https://www.chainguard.dev/unchained/are-sboms-good-enough-for-government-work (visited on 03/18/2023).

[5] M. Biebel. *Converting SBOMs between SPDX and CycloneDx*. Sept. 27, 2023. URL: https://mariuxdeangelo.gitlab.io/website/#/post/20230925-SBOM-Convertion-Tools (visited on 11/04/2023).

[6] M. Biebel. *SBOM dependency semantics in SPDX and CycloneDx*. Sept. 24, 2023. URL: https://mariuxdeangelo.gitlab.io/website/#/post/20230924-SBOM-dependency-semantics-SPDX-and-CycloneDx (visited on 11/04/2023).

[7] *chainguard-dev/bom-shelter*. Feb. 13, 2023. URL: https://github.com/chainguard-dev/bom-shelter (visited on 03/19/2023).

[8] *Convertion Bom-squad SBOM Convert CLI*. Aug. 18, 2023. URL: https://github.com/bom-squad/sbom-convert (visited on 09/03/2023).

[9] *Convertion cdx2spdx*. Aug. 29, 2023. URL: https://github.com/spdx/cdx2spdx (visited on 09/03/2023).

[10] *Convertion CycloneDx CLI*. Aug. 31, 2023. URL: https://github.com/CycloneDX/cyclonedx-cli (visited on 09/03/2023).

[11] *CRA EU*. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:52022PC0454 (visited on 11/24/2023).

[12] *CRA EU Cyber Resilience Act | Shaping Europe's digital future*. URL: https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act (visited on 11/24/2023).

[13] *Cybersecurity Mobilisation Plan Open Source Security Foundation (OpenSSF)*. Open Source Security Foundation. May 13, 2022. URL: https://openssf.org/oss-security-mobilization-plan/ (visited on 11/24/2023).

[14] *CycloneDX project History*. URL: https://cyclonedx.org/about/history/ (visited on 05/05/2023).

[15] *CycloneDX v1.4 Spesification JSON Reference*. URL: https://cyclonedx.org/docs/1.4/json/#bomFormat (visited on 05/05/2023).

[16] *Documentation around using PURLs as unique identifiers · Issue #242 · package-url/purl-spec*. GitHub. URL: https://github.com/package-url/purl-spec/issues/242 (visited on 11/29/2023).

[17] *Generator aquasecurity/trivy*. July 10, 2023. URL: https://github.com/aquasecurity/trivy (visited on 07/10/2023).

[18] *Generator Conversion anchore/syft*. July 10, 2023. URL: https://github.com/anchore/syft (visited on 07/10/2023).

[19] *Generator CycloneDX (cdxgen)*. July 8, 2023. URL: https://github.com/CycloneDX/cdxgen (visited on 07/08/2023).

[20] *Generator github dependency graph*. GitHub Docs. URL: https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph (visited on 07/10/2023).

[21] *Generator Microsoft SBOM Tool*. July 3, 2023. URL: https://github.com/microsoft/sbom-tool (visited on 07/10/2023).

[22] *Generator Scancode Output Formats — ScanCode-Toolkit documentation*. URL: https://scancode-toolkit.readthedocs.io/en/stable/cli-reference/output-format.html?highlight=spdx#spdx-rdf-file (visited on 11/25/2023).

[23] *Generator scancode-toolkit*. URL: https://github.com/nexB/scancode-toolkit (visited on 07/10/2023).

[24] *Generator Tern*. July 9, 2023. URL: https://github.com/tern-tools/tern (visited on 07/10/2023).

[25] goneall. *SPDX Understanding SPDX Profiles – SPDX*. Oct. 9, 2023. URL: https://spdx.dev/understanding-spdx-profiles/ (visited on 12/02/2023).

[26] C. C. W. Group. "CycloneDx Authoritative Guide to SBOM". In: (June 5, 2023).

[27] M. Hatta. "The Nebraska problem in open source software development". In: *Annals of Business Administrative Science* 5 (Oct. 15, 2022). ISSN: 1347-4456, 1347-4464. DOI: 10.7880/abas.0220914a. (Visited on 03/13/2023).

[28] S. Hendrick and V. Research. "Software Bill of Materials (SBOM) and Cybersecurity Readiness". In: (Jan. 2022).

[29] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch. *Log4j The Race to the Vulnerable: Measuring the Log4j Shell Incident*. June 7, 2022. DOI: 10.48550/arXiv.2205.02544. arXiv: 2205.02544[cs]. URL: http://arxiv.org/abs/2205.02544 (visited on 11/01/2023).

[30] T. W. House. *Executive Order on Improving the Nation's Cybersecurity*. The White House. May 12, 2021. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/ (visited on 03/18/2023).

[31] *Metrics SBOM Scorecard*. Oct. 6, 2023. URL: https://github.com/eBay/sbom-scorecard (visited on 10/29/2023).

[32] *Metrics sbomqs: Quality metrics for SBOMs*. Oct. 16, 2023. URL: https://github.com/interlynk-io/sbomqs (visited on 10/29/2023).

[33] J. S. Meyers. *How to Make High-Quality SBOMs*. Open Source Security Foundation. Mar. 2, 2023. URL: https://openssf.org/blog/2023/03/02/how-to-make-high-quality-sboms/ (visited on 03/18/2023).

[34] *NTIA Minimum Requirements OWASP CycloneDX Authoritative Guide to SBOM*. URL: https://cyclonedx.org/guides/sbom/bom/ (visited on 08/25/2023).

[35] *NTIA Minimum Requirements The Minimum Elements For a Software Bill of Materials (SBOM) | National Telecommunications and Information Administration*. URL: https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom (visited on 03/18/2023).

[36] *NTIA Minimum RequirementsAnnex K: How To Use SPDX in Different Scenarios - specification v2.3.0*. URL: https://spdx.github.io/spdx-spec/v2.3/how-to-use/ (visited on 08/25/2023).

[37] M. Ohm, H. Plate, A. Sykosch, and M. Meier. *Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks*. May 19, 2020. DOI: 10.48550/arXiv.2005.09535. arXiv: 2005.09535[cs]. URL: http://arxiv.org/abs/2005.09535 (visited on 03/22/2023).

[38] *purl-spec/README.rst at master · package-url/purl-spec*. GitHub. URL: https://github.com/package-url/purl-spec/blob/master/README.rst (visited on 11/29/2023).

[39] *Report on the 2020 FOSS Contributor Survey*. URL: https://www.linuxfoundation.org/resources/publications/foss-contributor-2020 (visited on 03/19/2023).

[40] *SBOM Everywhere OSSF SIG Github*. Mar. 18, 2023. URL: https://github.com/ossf/sbom-everywhere (visited on 03/19/2023).

[41] *SBOM Everywhere Tooling Ecosystem working with CycloneDX*. Google Docs. URL: https://docs.google.com/document/d/1biwYXrtoRc_LF7Pw10TO2TGIhlM6jwkDG23nc9M_RiE/edit?usp=embed_facebook (visited on 07/13/2023).

[42] *Snyk Terms of Service*. Snyk. URL: https://snyk.io/policies/terms-of-service/ (visited on 11/04/2023).

[43] *SOFTWARE BILL OF MATERIALS | National Telecommunications and Information Administration*. URL: https://ntia.gov/page/software-bill-materials (visited on 03/18/2023).

[44] "Software Security in Supply Chains: Software Bill of Materials (SBOM)". In: *NIST* (May 3, 2022). (Visited on 03/18/2023).

[45] *SPDX About*. Software Package Data Exchange (SPDX). URL: https://spdx.dev/about/ (visited on 05/01/2023).

[46] *SPDX specification v2.3.0*. URL: https://spdx.github.io/spdx-spec/v2.3/ (visited on 05/01/2023).

[47] S. Springett, D. Russo, G. Fick, J. Herz, J. Scott, et al. *BOM Maturity Model*. URL: https://scvs.owasp.org/bom-maturity-model/undefined/bom-maturity-model/urn/owasp/scvs/bom/resource/software/evidence/identity/method/value/ (visited on 12/04/2023).

[48] D. Waltermire, B. A. Cheikes, L. Feldman, and G. Witte. *Guidelines for the Creation of Interoperable Software Identification (SWID) Tags*. NIST IR 8060. National Institute of Standards and Technology, Apr. 2016. DOI: 10.6028/NIST.IR.8060. URL: https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8060.pdf (visited on 12/04/2023).

[49] 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*. Feb. 7, 2023. DOI: 10.48550/arXiv.2301.05362. arXiv: 2301.05362[cs]. URL: http://arxiv.org/abs/2301.05362 (visited on 03/13/2023).

[50] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, et al. "Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 30, 2023. ISBN: 9798400703270. DOI: 10.1145/3611643.3616299. URL: https://dl.acm.org/doi/10.1145/3611643.3616299 (visited on 12/11/2023).

Glossary

CPE Common Platform Enumeration.
CRA Cyber Resiliance Act.
CVE Common Vulnerabilities and Exposures.
CWE Common Weakness Enumeration.
EO Executive Order.
EU European Union.
GDG Github Dependency Graph.
ITAM Information Technology Asset Management.
JSF JSON Signature Format.
MST Microsoft SBOM Tool.
NTIA National Telecommunications and Information Administration.
OWASP Open Worldwide Application Security Project.
PURL Package Uniform Resource Locator.
SBOM Software Bill of Materials.
SCA Software Composition Analysis.
SPDX Software Package Data Exchange.
SWID Software Identifier.