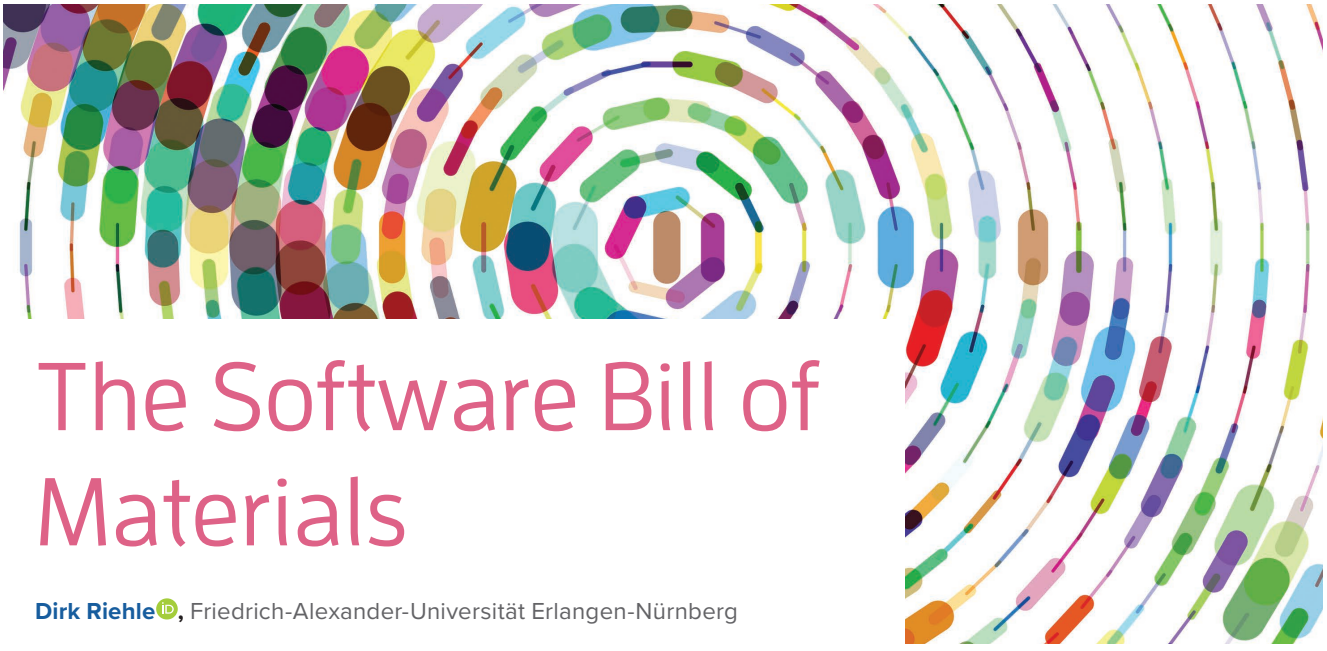
EDITOR **DIRK RIEHLE**
Friedrich Alexander-University of Erlangen Nürnberg;
dirk.riehle@fau.de



The Software Bill of Materials

Dirk Riehle, Friedrich-Alexander-Universität Erlangen-Nürnberg

Cybersecurity threats and software supply chain attacks are at an all-time high. Customers and agencies keep tightening the requirements for their software. An important recent development is the practical use of software bills of materials.

A bill of materials (BOM) is a list of components ("materials") that make up some artifact. A software BOM (SBOM) is a BOM where all the components are software components. It is important to have an SBOM for a software project or product that is as complete and correct as possible, for two main reasons.

- *Critical data structure:* Complete and correct SBOM data are critical for a host of engineering functions. The three most important functions that require an SBOM are
  - *Open source governance:* To decide which components are acceptable to a project or product, you first need to know whether they are included and then what licenses and other conditions they come with.
  - *License compliance:* To deliver your project to clients or your product to customers, you need to comply with the licenses of any open source code included in the software. The SBOM tells you what those are.
  - *Security and vulnerability management:* To manage operational risk, you need to understand what components are doing their job in the given software, whether there are known vulnerabilities, and whether new vulnerabilities have been discovered.
- *Non-functional requirement:* A complete and correct SBOM has become a purchasing requirement of many customers.

Originally, large customers in a software supply chain would require from their suppliers that they provide SBOMs together with any software they were supplying. In the case of custom software projects, large customers would even request to receive signoff authority on the use of open source components before they were incorporated into the software being built.

Digital Object Identifier 10.1109/MC.2025.3530276
Date of current version: 27 March 2025

FROM THE EDITOR

Welcome back! This month's "Open Source" column continues discussing open source use in and by organizations. We turn to a fundamental data structure that anyone developing or using project or products built from open source needs: the software bill of materials, that is, an inventory of components in software. This data structure has become so important that governments have made it a requirement of professional software.—*Dirk Riehle*

In 2020, the European Union (EU) announced the Cyber Resilience Act (CRA).[a] This regulation complements the previous NIS-2 legislation to improve product security across the EU. The CRA entered into force in 2024. Vendors of products that include software are required to provide an SBOM to customers as well as to proactively track and respond to any vulnerabilities that become known about their products.

In 2021, the U.S. American government issued an executive order requiring, among other things, that any U.S. federal purchaser of software be provided an SBOM for the software being purchased.[b] While previous motivations for an SBOM were mostly about license compliance, the U.S. government cares more about cybersecurity and the risks from vulnerabilities in software. It is safe to assume that other governments will follow suit.

For any given software, the SBOM needs to list the original code of the supplier, presumably with their proprietary license, as well as any third-party components. A third-party component is any code, including open source code, not owned by you. Such third-party components come in two main forms.

1. Standalone components are the traditional libraries and components you are including in your project or product.
2. Code snippets are chunks of source code that have been copied and pasted into your code or into the third-party code you are using.

The two prominent (and competing) specifications for representing SBOMs are the SPDX and the CycloneDX specifications. These specifications allow the presentation of an SBOM in a linear format (list) of records with each record representing a component and some of its metadata.

There are different types of SBOM, created for different purposes. The most common SBOM is the one given to customers as a part of selling a product. Other types of SBOMs add tooling information to document how the software is being built or include verification information to comply with regulatory requirements.

To create an SBOM, you need to identify and gather all third-party components your code is using, whether a standalone component or a code snippet. For any such component, you need to gather the necessary metadata for each component.

It is impossible to create a complete and correct SBOM for a nontrivial piece of software. Too much copy and paste without tracking lineage in both open and closed source software has ruined this opportunity.

THE DEPENDENCY GRAPH

The process of creating an SBOM is called *software composition analysis* (*SCA*). An SCA first creates the so-called dependency graph of your software and then derives the SBOM from it.

A dependency is a software component that some other component depends on. A component depends on another component if the component can't perform its function without the depended-on component. In the common case, this is a code dependency, like being able to call the functions of the depended-on component.

A dependency graph is a graph of software components as the nodes connected by depends-on (dependency) relationships as the edges (links). In any modern software, most of these components will be third-party components, including open source components, which are components owned and licensed to you by someone else.

There are many different types of components that can become nodes in a dependency graph, depending on how broadly or narrowly the dependency graph is to be used.

- In the original narrow sense, the components in a dependency graph are all code components. There are two types of components.
  - *Traditional standalone components or libraries:* These are components that have a clear boundary with their context (they come as their own package, ideally with a well-defined interface).
  - *Code snippets:* Code snippets are pieces of code that have been copied and pasted into your code by your developers or into open source dependencies by the open source developers. Legally speaking, such code snippets are components separate from the embedding component because they usually have a different copyright holder and a different license.
- In a more recent broader sense, with the goal of completely documenting everything that goes into the building of

[a] https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act.
[b] https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity

software, components can also be tools that build the software, resources that provide the necessary information, etc.

A dependency graph is a directed graph; incoming links to a component originate from other components that depend on this component, and outgoing links from a component go to the other components that this component depends on. As a matter of good software architecture, the graph is ideally also an acyclic graph.

Dependencies have levels. The level number is the number of steps removed from the root of the graph. This leads to the following definitions:

- The root component of a dependency graph has the level zero and is usually your own original code. There may be one or more root components.
- The first-level dependencies are the immediate dependencies of the root component. They are noteworthy because they are present in the minds of your developers and they are explicitly specified in your build system instructions. They are also often called the *direct dependencies*.
- Second- and higher-level dependencies are the dependencies of your first-level dependencies. They are also called *indirect dependencies*. They are noteworthy because they are not present in the minds of your developers and they are not very visible in their day-to-day work. Yet they constitute the largest part of the code that your project or product is built from.

The two prominent (and competing) specifications for representing SBOMs are the SPDX and the CycloneDX specifications.

As a rule of thumb, the size relationship between your original code, your direct dependencies, and your indirect dependencies is one to nine to 90 in parts. In other words, 90% of your vulnerabilities stem from code you are not thinking much about. The indirect dependencies are the proverbial iceberg under the waterline.

SBOMs are created from a dependency graph. The nodes of a dependency graph correspond to the component entries in the SBOM. While the dependency graph remains a graph structure, the SBOM drops the relationships and is (mostly) a flat list of components. For this reason, the dependency graph and SBOM are not the same. Figure 1 shows a dependency graph, including our term definitions.

SCA

SCA is the analysis of your project or product's source code to identify the component structure of the software, also known as its *dependency graph*. As discussed, components may be standalone components, or they may be code snippets. The code of a component may be owned by you or by a third party, then called *third-party code*. Open source code is the most prominent example of third-party code.

The main motivation for SCA, originally, was to ensure license compliance. Any third-party code is legally separate code that comes with its own licenses. You need to comply with these licenses when you are delivering your projects to clients and delivering your products to customers.

Legally separate does not necessarily mean technically separate. Most notably, source code snippets that have been copied into your source code or into your dependencies are

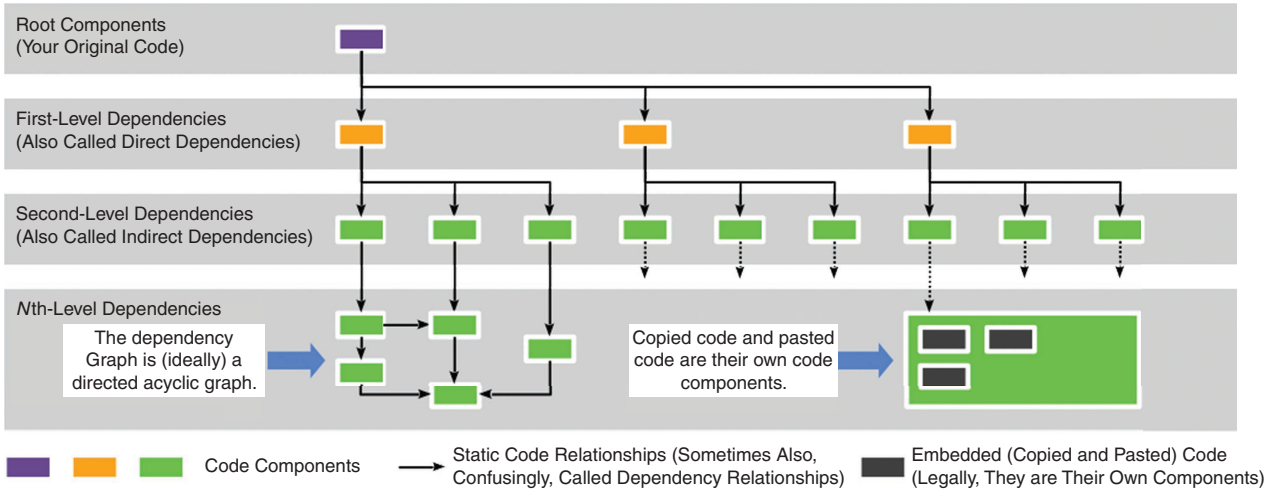


FIGURE 1. Illustration of a dependency graph.

legally separate code components even though they are embedded in your code or third-party code. You still need to identify these snippets, even in your dependencies, if you want to deliver license-compliant software.

SCA is typically performed using specialized tools. These tools read through the whole source code base of the software and try to identify any third-party code. An SCA tool needs access to the full source code, so in addition to providing your original code, you also have to either download the dependencies yourself or direct the tool on how to do so.

> An SCA first creates the so-called dependency graph of your software and then derives the SBOM from it.

Examples of open source SCA projects are FOSSology,[c] a complete solution, and ScanCode,[d] a scanning tool to be embedded into a larger custom toolchain. An example of a free but comprehensive service is SCA Tool.[e] In addition, there are many commercial tools on the market.

For an SCA tool, the software consists of a hierarchical folder structure with files and code snippets in files. Source code outside this folder structure is not considered. An SCA tool does not and should not make assumptions about the folder structure mapping to the dependency graph in a particular way. As the result of an SCA, you will be presented with the folder, file, and snippet structure rather than the dependency graph. An export of this information in SBOM form will provide a flat list (rather than a graph).

SCA is not a fully automatic process. Existing SCA tools will analyze the source code and present their findings to their users for signoff. The key findings presented to users are

- *Component identification:* For a given software component, an SCA tool will suggest a specific origin component, ideally using a unique component identifier like a package URL (PURL).

For a given code snippet, an SCA tool will also suggest the origin component and to this add the location of the source code within the component that the snippet may have been copied from.

- *Legal information:* Originally designed for license compliance, SCA tools will try to determine the component's legal information: which licenses, which copyright holders, and any other notices that a user needs to know about.
- *Vulnerabilities:* More recently, SCA tools started adding known information on vulnerabilities, though this is often considered a follow-on step in a toolchain and not part of SCA.

In addition to source code analysis, binary analysis tools let you analyze the software composition of binary files. Binary files can be found anywhere; they might be hiding in a source code folder or be part of a container image. Like source code, they need to be found, identified, and analyzed.

WORKING WITH SCA TOOLS

SCA is a tool-based process that cannot be fully automated. An SCA tool expects or downloads a hierarchical structure of all relevant artifacts. Typically, this is a folder hierarchy of source code files pulled from version control, but it can also be container images with nondescript binary files included.

Different SCA tools naturally provide different functionalities. At its core, however, there are three different types of source code analysis results that an SCA tool might provide to its users (not all tools do).

- *The dependency graph:* A core output is the actual dependency graph (not just the SBOM). This includes correctly identifying both components and their linkage (forming the components into a graph).

Modern package managers have made it easy to determine a software's dependency graph, but many older software systems written in languages without established package managers resist any automation of creating the dependency graph. Package managers help SCA tools identify a component. The metadata provided by package managers, for example, component licenses and owners, is more often incorrect than not.

- *Meta-data from source code analysis:* Another core output of an SCA tool is the analysis of the source code. Most commonly, SCA tools look for legal information to help users ensure license compliance.

Identifying legal information is commonly performed in a simple and straightforward way by using regular expression matching against defined terms and databases like license text databases. Code quality analysis and identifying unknown vulnerabilities are also useful analysis functions available in some tools.

- *Snippet matching of your and third-party source code.* The final core output of some SCA tools is the identification of code snippets that may have been copied from the web into your code or any third-party code, including open source components.

Free-to-use open source SCA tools usually don't offer a snippet matching

[c]https://github.com/fossology/fossology.
[d]https://github.com/aboutcode-org/scancode-toolkit.
[e]https://scatool.com.

feature because to perform this function, the tool needs to compare any code snippet against the whole wide world of third-party code. This requires the creation and continuous updating of a large database of such third-party code, which can become rather expensive.

A tool like SCA Tool works through the artifact hierarchy and collects its findings for review and signoff by its users. It is not advisable for users to just accept what an SCA tool is suggesting. More often than not, the findings will be wrong. To this end, SCA tools provide users with a workflow in which they can review each finding for correctness.

There are many challenges to a human review.

- *Erroneous data:* An SCA tool may pull in erroneous data, for example, from package managers. Users need to review and correct these data.
- *Laborious process:* The developers of an SCA tool typically don't want to be on the hook for overlooked third-party code. Hence an SCA tool is set to be highly sensitive, often suggesting third-party code, in particular copied and pasted snippets, where there is none. This leads the tool to declare a large number of findings, many of which, if not most of them, will be false positives. Working through all these findings is a significant time sink for SCA tool users.
- *Error-prone process:* The review process is highly error prone because it is mind-numbingly boring. Reviewers have to work through a large set of findings, many of which are similar and repetitive yet may vary in minor but important details. As humans work, attention may wane, and a desire to move forward will get its way, leading to sloppy work and, ultimately, errors in the analysis and review process.
- *Expensive review:* The review is often delegated to the original developers, who would rather be writing new code and shipping features than reviewing old code and cleaning up legal debt. Using your developers to review SCA tool findings is rather expensive labor and often better to be delegated to third parties.

Creating a dependency graph and deriving the SBOM for the first time, therefore, is often a laborious, expensive, and error-fraught process. Ideally, changes to your project and product lead only to an incremental adjustment of the dependency graph and SBOM data.

BASIC SBOM REQUIREMENTS

An SBOM captures which code components are included in the software. There are two original uses.

- The first use of the SBOM information is to ensure that only code components that both the developer and any recipient would find acceptable were included; most notably, developers generally prefer to keep copyleft-licensed components out of their products.
- The second use of this information is to create proper legal notices for the third-party code in the software. A developer, when distributing the software, has to provide these legal notices about the included open source components to comply with their licenses.

Customers in a supply chain often make the provision of an SBOM a purchasing requirement, as discussed before. Governments have followed suit, mostly driven by the need to make software more secure.

A report by the U.S. Department of Commerce details the basic requirements for an SBOM.[f] Any SBOM should name its author and the time it was created. Each component (material) in an SBOM should provide the component's name, its version number, and the supplier of the component. Interestingly, the report also states that the component should list its relationship to other components, which I would have considered helpful but not critical.

The report sees SBOMs as hierarchical structures. At the root is the SBOM for the software being described. The components in the SBOM can then have their own SBOM, potentially creating a hierarchical structure. You cannot, however, map the dependency graph into a hierarchy, at least not without creating significant redundancy. I argue that the components in an SBOM should simply be captured as a flat list; if preserving the dependency graph is important, each component can reference the components it depends on.

Also, an SBOM should be machine readable for automated processing. The report lists SPDX, CycloneDX, and SWID tags as established format specifications for capturing SBOM information. The report notes that the industry so far has failed at providing unique identifiers for components and that the supplier and component names should therefore be human readable, for human interpretation, but not necessarily machine interpretable.

The grassroots PURL effort is offering help to uniquely identify components.[g] The supplier of the component and its name (and version number, etc.) are encoded into one heterogeneous name value, the PURL. It consists of seven components structured using the following syntax:

```
scheme:type/namespace/name@
version?qualifiers#subpath
```

While not directly a traditional URL, a PURL uniquely nevertheless identifies a location. The location then becomes the supplier of the component.

[f] https://www.ntia.doc.gov/report/2021/minimum-elements-software-bill-materials-sbom.

[g] https://github.com/package-url/purl-spec.

Therefore, identical copies of the same code base in different locations are treated as different components.

An SBOM that fulfills these basic requirements can already be delivered with the software to its users to fulfill a purchasing requirement. That said, there are many more types and uses of SBOMs.

TYPES AND USES OF SBOMs

The original and still primary use of an SBOM is to list what components are included in the software when provided to a customer. As explained, there are two main uses.

1. *Governance and compliance:* Large companies in a supply chain wanted and still want to know what components they have to deal with, often in advance of a delivery. The primary reasons are open source governance (ensuring that only desired components are included) and license compliance (making sure that the licenses can be complied with).
2. *Security:* More recently, fueled by worries about cybersecurity, governments, including the United States and the EU, have put forth requirements that any product is to come with an SBOM. This way, users can identify any security issues with the product as vulnerabilities of the included components become known. SBOMs have become a purchasing requirement.

This is not the only type of SBOM; there are several more. The most prominent classification of SBOM types is provided in a 2023 white paper by the U.S. Cybersecurity & Infrastructure Security Agency (CISA).[h] CISA identifies six different types of SBOM, which can be broken down into two sets of three depending on how the SBOM is created.

- The first set is SBOMs created from the supplier's development process.
  1. *Design:* A Design SBOM is created from planning documents like prospective product architectures. As a consequence, a Design SBOM may not be an accurate reflection of what will be shipped eventually. It may be helpful to buyers in a supply chain to prepare for what's to come their way.
  2. *Source:* A Source SBOM provides a static picture of the supplier's source code and its dependencies, as found in the repositories. It can be helpful to identify vulnerabilities but does not provide a complete picture as it omits any build or runtime dependencies.
  3. *Build:* A Build SBOM is created from the build process of the supplier as it compiles source code and assembles the final package for delivery to customers. Aimed at operations, it does not include components needed for building and testing. It may still miss dynamic dependencies, though.
- The second set is SBOMs created by the buyer (or others) through analysis.
  1. *Analyzed:* An Analyzed SBOM is created from SCA of the static delivered software. This is almost always a binary analysis of the artifact. As such, an Analyzed SBOM will miss much, but it may discover components that the suppliers may have overlooked.
  2. *Deployed:* A Deployed SBOM is created by analyzing the deployed software. After deployment, additional components may have been loaded or may have become visible that were not identifiable before. Like Analyzed, Deployed SBOMs complement the supplier's SBOMs.
  3. *Runtime:* A Runtime SBOM is created from observing the running software (often requiring instrumentation). Of the SBOMs created through analysis, a Runtime SBOM provides the most comprehensive picture, but it will miss components that have not been activated and are not visible yet.

The original type of SBOM mentioned in the beginning corresponds to the Build SBOM created by the supplier. Other SBOM types have other uses; for example, the developer may want to track and document details of testing and staging their products for various reasons—for example, debugging, auditability, or certification.

There is a logic of progression in the two classes of SBOMs. A Build SBOM is by and large more comprehensive and more accurate than a Source SBOM than a Design SBOM, and a Runtime SBOM is by and large more comprehensive and more accurate than a Deployed SBOM than an Analyzed SBOM.

Both classes complement each other; SBOMs created by the supplier may miss some dynamically loaded components, knowingly or unknowingly, and SBOMs created by user analysis may miss some or many of the components that the system has not yet run into or that were obscured otherwise.

Taken together, a Build SBOM and a Runtime SBOM can provide a comprehensive picture, one that is needed for safe and secure operations. of software by organizations of any size.

DIRK RIEHLE is the professor for open-source software at Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany. Contact him at dirk@riehle.org.

[h]https://www.cisa.gov/sites/default/files/2023-04/sbom-types-document-508c.pdf.