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

Free and Open Source Software License Compliance: Tools for Software Composition Analysis

Philippe Ombredanne, nexB Inc.

Modern software is assembled from thousands of open source components, each with its own provenance and license, meaning that automation is the only practical way to comply with license conditions. We review the domain of software license compliance tools.

Digital Object Identifier 10.1109/MC.2020.3011082
Date of current version: 5 October 2020

In an ideal world, the provenance and licensing of third-party software components would be available as easy-to-discover structured data. But a recent study[1] on license documentation found that fewer than 5% of approximately 5,000 popular free and open source software (FOSS) packages contained complete and unambiguous license documentation. Modern software products and applications are assembled like LEGO blocks from FOSS components because reusing existing code is a faster and more efficient way to create solutions. Provisioning FOSS components from the Internet is frictionless since it takes only a few seconds of a programmer's time to download and install a new element. Updated component versions may be released several times per year, and each version may have a different provenance and various licenses and dependencies.

This means that a typical software team needs to continuously

FROM THE EDITOR

This month's column kicks off the license compliance tooling and standards section of the "using open source" theme arc. Before you can deliver a software product with correct legal notices to customers, you have to understand what open source code is included in your software. You have to create the so-called bill of materials. Does this sound simple? It is not. In this article, Philippe Ombredanne, author of the widely used license text scanner ScanCode, introduces us to the challenges of analyzing a software code base for all the open source code that knowingly or unknowingly was added to it. More than ever, happy hacking, and stay safe! —*Dirk Riehle*

track a very large number of provenance and license combinations. Finding thousands or tens of thousands of FOSS components in a single application code base is now commonplace. Since a significant number of these components may have poor or missing license and provenance documentation, software tools are essential to tackle the data volume, fill the information gaps, and automate most FOSS license compliance activities. The focus of this article is on tools and techniques for identifying FOSS components that you use and how you use them because your FOSS compliance obligations depend on both.

Finding thousands or tens of thousands of FOSS components in a single application code base is now commonplace.

There are very few FOSS license conditions that apply when you only use or run software "internally." In general, you are obliged only to provide attribution and redistribution for FOSS components that you distribute. In the FOSS license context, *distribution* means that you provide software to a third party as a software package (via a download or through a medium such as a DVD) or deployed on a device (a smartphone, computer, Internet of Things device, and so forth).

If you build applications rather than software products or deploy software products in a hosted software-as-a-service environment, you may think that you do not distribute any code. But distribution occurs more often than you may think. Do you publish a mobile app? This is software distribution. Do you publish a dynamic website? This may include software distribution if your website used client-side JavaScript code that is downloaded to run in a browser.

In a typical software development environment, a significant proportion of the FOSS components used by developers is for internal use only, for example, compilers, integrated development environments, build tools, test tools, and so on. Therefore, if you distribute a product, it is critical to know exactly which FOSS components (including compiled binaries) you circulate so that you understand your attribution and redistribution obligations.

SOFTWARE COMPOSITION ANALYSIS TOOLS

There are several possible areas where tools may help with license compliance activities. In this article, we focus on the compliance tools that are essential for identifying the provenance and licenses of FOSS components as they are used in running software. We are providing only a limited summary for important yet secondary activities, such as interaction and binary analysis, attribution notice generation, and corresponding source code redistribution obligations.

Which FOSS components do you use?

The first step for any FOSS compliance program is to identify the FOSS components you use. The current industry analyst (for instance, Gartner or Forrester) term for identifying third-party software components is software composition analysis (SCA). SCA is broader than FOSS license compliance because it also includes the identification of security vulnerabilities and quality attributes, although those topics are beyond the scope of this article. FOSS component identification is a complex and time-consuming activity, but this is the area where you can get the biggest benefits from tools and automation.

Scanning versus matching

The set of techniques to identify which FOSS components you use and their provenance and licenses is generally known as *scanning*. This term can be confusing because there are actually two different techniques to consider:

- Scanning is when you directly extract information from source and binary files.
- Matching is when you search for the provenance of files based on matching file content and attributes to an external index of FOSS components.

Scanning does not require an external database. Matching requires a preindexed database of known FOSS components, including metadata and code (source and binaries).

SCANNING

With scanning, you can detect material that includes the following:

- structured information from package manifests and build scripts
- explicit license notices, license tags, license mentions, and license texts
- other provenance clues, including emails, uniform resource locators, and specific code constructs, such as programming language imports, include statements, namespaces, and code tree structures.

Parsing declared licenses from a manifest

The simplest scanning technique is to collect the data from a FOSS component that comes with structured provenance and license information. It is important to consider the following:

- When installed from a package repository, a package has a manifest that contains structured provenance data (such as Java with Apache Maven and JavaScript with Node Package Manager), including references to source code and version control repositories.
- Code and documentation files may contain other structured data, such as a Software Package Data Exchange (SPDX)[2] document or a notice file.

In practice, only a subset of the packages may contain declared provenance and license data.

For the ClearlyDefined project,[7] we evaluated the clarity of the license documentation for roughly 5,000 of the most popular FOSS packages. In this study,[1] a package with a high license clarity has a top-level declared license (for example, in a package manifest or a copying or readme file) and consistent file-level license notices in most of the code files where the licenses are well-known FOSS licenses listed by the SPDX project[3] and where the package contains complete license texts. The license clarity scores achieved with these criteria are lower than we expected. The overall median and average license clarity scores are approximately 45/100. Only 194 of the 4,892 packages had a license clarity score of at least 80/100. While it is encouraging that roughly 75% of these packages have a "declared" top-level license, such a declaration may be inconsistent with file-level license notices. Fewer than 15% of the packages provided a top-level license consistent with file-level license notices.

FOSS package management tools and some external tools (such as the ScanCode toolkit[3]) provide a way to collect FOSS package data. Yet, providing a unified view of software metadata is challenging because each package manager has its own unique way to deliver structured metadata. To better understand the complexity of the problem, the CodeMeta Project[4] publishes evolving mappings and cross-references of the data attributes, names, and definitions of more than 10 different package ecosystems. Yet for some package managers, such as those for Go, the package manifest contains no license information.

Scanning will not help you if there is no provenance and license information in the code you analyze.

Scanning to detect license texts, notices, and mentions

Because a license declaration may not be present in a package manifest or may be incomplete or ambiguous, you also need to detect and normalize other license references in text and notice files in a code base. Before some recent housekeeping, we found that there were approximately 800 different ways to state that a file was licensed under the GNU General Public License (GPL) in the Linux kernel sources. These license notices can be short: a few words, such as "license: MIT," or one word, such as "GPLv2," are considered by some authors to be a sufficient license declaration. A notice can be very long, such as the full text of the GNU Affero GPL 3.0 (approximately 37,000 characters). The challenge is to account for thousands of texts with many small and large variations. Each variation can be detected using different text and string comparison techniques. There are three main approaches used to detect licenses:

1. pattern matching, where small text patterns are handcrafted and used as proxies to search for licenses
2. probabilistic text matching, where a similarity metric (typically the edit distance) is used to find the closest matching license or notice text
3. exhaustive pairwise comparisons to find similar licenses using text sequence alignments (which is also known as *diff*).

The most popular FOSS tools for license detection include Fossology[5] (using approach 1), GitHub licensee[6] (using approach 2), and ScanCode Toolkit[5] (using approaches 1 through 3). Most tools use only the first and second approaches, which are only approximate. For more details, a good list of license detection tools is maintained by the Debian project.[7]

MATCHING

In contrast to scanning, the goal of matching is to find code borrowed from FOSS projects, based on the detection of code similarities (for example, duplicates, near duplicates, and clones).

Why matching?

Scanning will not help you if there is no provenance and license information in the code you analyze. Therefore, the primary use case for matching is to analyze code base files that do not have any clear provenance and license information. A secondary matching use case is to verify that FOSS files identified from the scanning data match the original files from the corresponding FOSS project. For matching, the basic approach is to find textual similarities between the code under analysis and other source and binary code files. If you think of matching as a search problem, the search index would be much smaller than a typical Internet-scale search engine: several terabytes for code matching are still many orders of magnitude fewer than the petabytes used by the search indices of Google Search and Microsoft Bing. Yet the size of the "query" to this smaller index can be gigabyte-size, as a whole code base is under analysis. This is in contrast with a search on Google Search, which is limited to a 32-word query of only a few bytes.

There are few FOSS tools available today for code matching, making this an attractive area for new development.

How much FOSS code is there?

The volume of FOSS code to index in a matching database is very large. The Software Heritage project[8] has already archived more than 8 billion unique source code files coming from more than 120 million projects representing several hundred terabytes of code. A typical Linux distribution maintains roughly 30,000 binary packages. Libraries.io[9] reports tracking more than 5 million FOSS packages (ignoring versions), and ClearlyDefined.io[10] tracks 10 million-plus FOSS package versions. GitHub claims to host more than 100 million code repositories.

Finding similar code

The simple and correct solution would be to perform an exhaustive pairwise comparison between a code base that is under analysis and the indexed code. But the data volume makes this approach impractical because it would take too much time to compute or would be too expensive when using cloud computing resources. A solution is to reduce the dimension of the problem to something smaller. Fixed-size file and code snippet checksums as well as "fuzzy" fingerprints and sketches[11] are used as smaller proxies to search for file similarities in a more cost- and time-effective way. For instance, a 128-bit checksum can be used to find code that is several thousand times bigger than it is. This reduced length helps keep the index size smaller and the lookup times and cost more practical with minimal loss of accuracy.

The main weakness of the matching approach is that with a large index, matching tools tend to provide many false positives that require expert review. FOSS components tend to be reused extensively by other FOSS projects, so any large FOSS index suffers from the presence of many duplicates and near duplicates. This duplication introduces many ambiguities into the match results.

Matching tool options

There are few FOSS tools available today for code matching, making this an attractive area for new development. Existing tools have been primarily provided by commercial companies, such as BlackDuck Software (acquired by Synopsys) and Palamida (acquired by Flexera), and new entrants, such as FOSSID. One emerging FOSS solution may be the Software Heritage project,[8] which provides a checksum lookup application programming interface that can be used for large-scale file matching against open data.

Scanning and matching summary

At first glance, matching seems to be a better way to detect code provenance than scanning. In practice, however, scanning is typically faster and more accurate than matching. Scanning and matching are ultimately complementary, but it is usually most efficient to start with scanning. Beyond choosing scanning and matching tools and techniques, you need to plan your SCA activities according to how you use FOSS components, that is, which components you distribute versus those that you use only internally for development, testing, and continuous integration/continuous delivery.

Which FOSS components do you really distribute?

Only a subset of all third-party packages that you identify may be used when running a program. There could be tools and testing utilities as well as documentation that may not be part of the code that is distributed and deployed, and these often have a different license than the code. Another factor is that many dependent FOSS packages ("dependencies") are not part of the source code stored in a version control system and used in development. They are usually downloaded at build time from a shared public or private repository. If you apply only scanning and matching techniques to the source code base, you will usually miss the dependencies, which may constitute hundreds of packages. Overall, the distribution package (the "binaries") for a product is often the best place to identify the set of FOSS components that require attribution and redistribution. If you do not do this, you need to "resolve" the dependencies of FOSS packages, as stated in the development code.

Resolving dependencies related to third-party packages

When using an application package repository (such as RubyGems and NuGet), a software programmer will state the direct dependencies in a dependency manifest file. At software build time, these first-level dependencies are fetched and installed by a package manager or a build tool (such as Python pip and Gradle). This process applies, in turn, to each dependency recursively and "all the way down." It is not uncommon to have deeply nested package dependency hierarchies that contain 1,000 packages or more, even though only a few first-level direct dependencies are stated by the programmer. With a Docker containers and virtual machine "images," which are both popular formats to deploy software in the cloud, an application may routinely embed 10,000-plus application packages and Linux binary system packages, each with its own provenance and license.

To identify resolved dependencies, you need to perform one of the following steps:

- Collect a preresolved list of package dependencies, which is called a *lockfile*.
- Run a software build to collect the list of dependencies that are or would be installed.
- Analyze the set of dependencies found in the software as deployed or distributed.

FOSS tools, such as the Open Source Review Toolkit,[12] provide a way to resolve, collect, and fetch the code of application dependencies by imitating the build process.

Beyond

There are also other tools and techniques that may be required to determine your FOSS compliance obligations, including the following:

- Analyzing the content of C/C++ and Go compiled binaries involves reverse-engineering techniques, such as symbol parsing and decompilation, and will require a tool such as BANG (Binary Analysis Next Generation).[13]
- Analyzing the content of mobile applications archives, such as an Android .apk file and an iOS .ipa file, also requires specialized tools.

After you have identified the FOSS components that you use and how you use them, you should be ready to focus on the second stage of FOSS compliance activities, where you create the attribution and redistribution artifacts. This part of compliance is typically somewhat easier than the SCA activities explained in this article, but it merits a separate, future article.

FOSS license compliance tools are an emerging domain with surprisingly complex requirements and a wide range of options. The tools that are FOSS themselves offer many opportunities for community collaboration and a foundation for an organization to assemble a bespoke and efficient toolchain to support its needs. C

REFERENCES

1. P. Ombredanne and D. Clark, "What is the state of open source license clarity?" ClearlyDefined, Apr. 26, 2019. [Online]. Available: https://github.com/clearlydefined/license-score/blob/master/ClearlyDefined%20-%20ClearlyLicensed%20clarity%20report-2019.pdf
2. SPDX. Accessed on: July 23, 2020. [Online]. Available: https://spdx.org
3. AboutCode, "ScanCode," 2020. [Online]. Available: https://www.aboutcode.org/projects/scancode.html
4. The CodeMeta Project. Accessed on: July 23, 2020. [Online]. Available: https://codemeta.github.io
5. Fossology. Accessed on: July 23, 2020. [Online]. Available: https://www.fossology.org/
6. Licensee. Accessed on: July 23, 2020. [Online]. Available: https://github.com/licensee/licensee
7. Debian, "Copyright review tools," 2020. [Online]. Available: https://wiki.debian.org/CopyrightReviewTools
8. Software Heritage. Accessed on: July 23, 2020. [Online]. Available: https://www.softwareheritage.org/
9. Libraries.io. Accessed on: July 23, 2020. [Online]. Available: https://libraries.io
10. ClearlyDefined, "Stats," 2020. [Online]. Available: https://clearlydefined.io/stats
11. Wikipedia, "Locality-sensitive hashing," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Locality-sensitive_hashing
12. OSS Review Toolkit. Accessed on: July 23, 2020. [Online]. Available: https://github.com/heremaps/oss-review-toolkit
13. Binary Analysis Next Generation. Accessed on: July 23, 2020. [Online]. Available: https://github.com/armijnhemel/binaryanalysis-ng

PHILIPPE OMBREDANNE is the chief technology officer at nexB, San Carlos, California; the maintainer of the ScanCode toolkit project; and a lead maintainer for AboutCode.org projects. Contact him at pombredanne@nexb.com.