OpenChain Compliance Automation

Generating compliance artefacts

Alexander Murphy - Orcro Limited

2023-09-19

OpenChain

International standard (ISO/IEC 5230) lists requirements for a quality open source licence compliance program.[1] The specificaiton and workgroup manage the development publicly on GitHub.[2]

The standard provides the following definition:

> *2.1 - Compliance artifacts*
>
> *A collection of artifacts that represent the output of a compliance program and accompany the supplied software*
>
> *Note: The collection may include (but is not limited to) one or more of the following: attribution notices, source code, build and install scripts, copy of licenses, copyright notices, modification notifications, written offers, open source component bill of materials, and SPDX documents.*

Production of these data (the artefacts[3]) are what we will show via this demonstration. Attribution notices and licence texts specifically, although other artefacts may be required. This is not an exhaustive exercise, regardless, it is highly unlikely that the steps in this example will map exactly to your own use-cases.

Tools

OpenChain is a non-prescriptive standard. To use computer science terminology, OpenChain is declarative, it tells you *what* to do, but you decide *how* to go about it. This demonstration is one generalisation of a method (rather, a collection of tools and processes) used frequently at Orcro to generate compliance artefacts.

There are numerous other tools which you may consider using to manage licence compliance, and, specifically, to generate the artefacts. Some of these are listed in Appendix A: Tools and Projects.

Scancode toolkit

Orcro's favorite OSS SCA tool (https://github.com/nexB/scancode-toolkit) is fully featured, well supported, and has an excellent community. It is straightforward to integrate into existing pipelines (run Python script direct from the repo).

The functionality scancode provides is essentially leveraging a comprehensive battery of regex patterns to `grep` source code for licence/copyright matches. The huge test suite means that it is has a high reliability.

R

Statistical computing programming language. This is the author's personal preference although Python and bash scripts would be more than suitable. No libraries are required, just "Base R". All the scripts used in

[1] OpenChain project homepage: https://www.openchainproject.org/

[2] Specification available on GitHub: https://github.com/OpenChain-Project/License-Compliance-Specification

[3] Artefacts - British engligh, artifacts, american english. Former used throughout.

this demonstration are R scripts.

Git

The code we review will be located in a Git repository. *this* repo (where this pdf is stored) contains a submodule to the project used in this demonstration, OpenBLAS (more on this project below). Git may be used to track generation of materials and help to pin them to versions.

Overview

We begin by identifying what data we need to automate. This step is usually resource-intensive, so an arbitrary (and pre-vetted by me) software product will be used for this example.

Then we break-down the automation pipeline into "components", and semi-manually run these to illustrate what the tools are doing. In other words, we run through production of the compliance artefacts step-by-step, and see what would happen behind the scenes if we were to run a "generate compliance artefacts" command in CI.

The output of the process (the compliance artefacts) will then be shown. These will be in the form of plain text files, and these can be manipulated like any regular file, such as compressed or serialised, if required.

Setting expectations

- Implementing the process on a CI system is not covered.
- This only covers generation of compliance artefacts for a single library at the application layer. It is highly likely that additional FOSS is used throughout any shipped product (Docker base images, GPU drivers, etc.) but this process is for the individual library only.
- We won't consider legal requirements, such as how the jurisdiction your code is shipped to may affect your obligations.
- Engineering considerations, such as caching SCA results.

OpenBLAS

BLAS (basic linear algebra subprograms) is a widely used library for matrix and vector calculations. The source is freely available[4] but isn't licensed per se, certainly not under an open source licence, rather it has some unusual wording about being free to use but to add notices if anything is modified and to provide acknowledgement for the authors.

On the other hand, OpenBLAS uses BSD-3-Clause for its out-licence, and the full source (which we look at here) is available on GitHub[5]. We will consider the compiled code where no additional third-party libraries have been included via cmake.

Identifying the requried artefacts

There are some typical questions you will *ask* when designing the automation system such as:

- What is *distributed*?
- Are there any *dependencies*?
- *How* will the code be shipped?
- Are there any *snippets* present?

In this case, all of the source code is provided in a monorepo. We can run our tools here without requiring any additional *source code* downloads[6].

[4]The reference implementation https://www.netlib.org/blas/
[5]Repository for OpenBLAS: https://github.com/xianyi/OpenBLAS
[6]We will download some information, but we'll cover that later

Note that build tools don't end up in the distributed binary (typically), therefore there is no need to be concerned with licences they may be under, although cmake is distributed under BSD-3-Clause so this wouldn't be a significant issue anyway as it matches OpenBLAS.

As the main licence is BSD-3-Clause, in fact, all the licences are permissive, the obligations are straightforward: Provide copyright notices (all of them!) and the licence text(s).[7]

We also see that it doesn't matter whether the code is distributed as source or in binary form, the artefacts are required. Note that if distributed *as* source code, then the compliance materials *will be provided automatically as they are contained within the source itself*! But we consider the compiled case here, so we must provide the artefacts separately.

> Providing the full source code for components licensed permissively is not *required* but it may be one way to comply! Although, if you have more than one component with different obligations then things can get messy quickly, hence, automation...

Identifying the source

Once the required artefacts are identified, the next step is to identify *what source code* relates to the binary code. For this demonstration it is very straightforward, we are producing the OpenBLAS binary, compiled with no other third-party libraries, therefore, the source we must analyse (our data) is the raw monorepo source code for OpenBLAS.

```
git clone --depth=1 https://github.com/xianyi/OpenBLAS.git
```

It's not necessary to clone the history of the source code, so we can lighten the load on the CI system simply by specifying only the most recent source with `--depth=1` (which we will assume our build is from). If we were using a specific version, then after the download completes we might want to track back to a specific commit with something like:

```
git reset --hard ###
```

Replacing `###` with the commit hash. But we will use the most recent version.

Initial SCA

With the required source code (input data to scancode) metaphorically in-hand, we can run our software composition analysis, or SCA scan. We're not scanning for anything other than OSS licensing metadata.

Scancode can be run from the command-line. So a developer, who is working inside the development repo, can simply use (or include the following in CI scripts):

```
scancode -clp -n 18 --csv ScanOut.csv path/to/OpenBLAS
```

Which will run scancode on the OpenBLAS source repository that we previously cloned. This assumes that scancode has been installed (added to path et cetera), but see Appendix B: Scancode Installation for simple installation instructions.

As this is the first time that we have extracted copyright information using scancode, we should take a look at the data it has provided us with. It is typically high quality, but there are sometimes things that deserve particular attention. Anyway, it is easy to do this.

There are various ways of checking, for example, if scancode was run with the `--csv` output format then it can even be loaded up into Excel. Here though, we'll use R.

```
# From the R cli, or this can be placed into an Rscript
copyright_data <- read.csv("ScanOut.csv") # load the scan results
```

[7] Another obligation is to *not* use the name of the project in marketing, but this does not impact the compliance artefacts r.e. openchain

```
# summarise 'unique' licences in the dataset (so we only see one of each result)
licences <- unique(copyright_data$detected_license_expression_spdx)

# display them
licences
```

```
 [1] ""
 [2] "BSD-2-Clause-Views"
 [3] "BSD-3-Clause"
 [4] "BSD-3-Clause AND BSD-2-Clause-Views"
 [5] "LicenseRef-scancode-blas-2017"
 [6] "BSD-3-Clause AND (BSD-3-Clause AND BSD-2-Clause-Views)"
 [7] "BSD-2-Clause"
 [8] "BSD-2-Clause-Views AND CC0-1.0"
 [9] "BSD-3-Clause-Open-MPI"
[10] "LicenseRef-scancode-blas-2017 AND BSD-3-Clause"
[11] "LicenseRef-scancode-proprietary-license"
[12] "MIT"
[13] "Apache-2.0"
```

R has spat out this list of SPDX IDs[8] and provided us with licence references for further information on certain files (`LiceseRef` where there is an unknown detection).

There doesn't appear to be too much to review here, these are all Permissive licences, or variants of one. Except for that result of `LicenseRef-scancode-proprietary-license`. You should not be too alarmed at these results (at least in the first instance) because Scancode is quite *greedy*. It will generate more false positives than omissions, and it's better this way as you can manually review the additional results (you cannot manually review what you do not know exists)!

Let's have a further look at this result, to see how often it appears, and what is triggering this result in the scan.

```
# selecting results that match the expression in brackets [expression]
# and then from the results, extract the path of the file (*where* the scan found it)
copyright_data[copyright_data$detected_license_expression_spdx ==
                 "LicenseRef-scancode-proprietary-license", ]["path"]
```

```
                                      path
15408 OpenBLAS/lapack-netlib/LAPACKE/README
```

So the proprietary license result came from the LAPACK docs. We don't need to worry about this because we're assuming we're compiling without third-party programs.[9]

It would be good practice to verify all of the licence results, at least by sample, but as everything is permissive here there's unlikely to be any major issues, *and this aligns with our expectations and understanding of OpenBLAS*, so, we'll proceed with generating the artefacts.

Generating artefacts

Now that the scan results have been collected and reviewed for issues, we can use whatever means at our disposal to generate appropriate artefacts. There are various ways in which artefacts *could* be provided for

[8] Where one exists, the eagle-eyed amongst you may notice the empty quotation marks " " which indicate a file has no licence information. This is fine, as we *know* it is part of the OpenBLAS source code. However, metadata, such as REUSE (see Appendix A) may alleviate these null results.

[9] You may be concerned here how we know this, and this is what was mentioned earlier, you can automate the heavy stuff but some domain knowledge is essential for these edge cases. Regardless, LAPACK is also under a semi-OSS licence, so even if we were compiling with it, we would not be concerned. Therefore, let's not attempt to remove it, we may have a rogue copyright notice but who doesn't like extra credit?

any particular licence, so what follows is appropriate for permissive licences, OpenChain compliance, and where no source code provision is required.

Some scripts have been developed by Orcro for this particular task, and are available under Apache-2.0 for you to freely use yourselves, or to learn more about generating artefacts if you wish to[10] and they are also as a submodule in this demo repository.

We're going to use a simplified script, which takes raw scancode output and produces the artefacts. It has no user-interface to speak of however, so let's see how it works step-by-step. This is what you would run at a CLI, or put into a script:

```
Rscript ScanToArtefacts.R ScanOut.csv
```

We use the `Rscript` command line tool to run the script `ScanToArtefacts.R` on the SCA data file `ScanOut.csv`. Briefly, before we look at the script, let's once again look at the variable which scancode is providing us with:

```
# view the variables that scancode extracts)
names(copyright_data)
```

```
 [1] "path"                                "type"
 [3] "detected_license_expression"         "detected_license_expression_spdx"
 [5] "percentage_of_license_text"          "scan_errors"
 [7] "license_expression"                  "detection_log"
 [9] "license_match__score"                "license_match__matched_length"
[11] "license_match__match_coverage"       "license_match__matcher"
[13] "license_match__license_expression"   "license_match__rule_identifier"
[15] "license_match__rule_relevance"       "license_match__rule_url"
[17] "author"                              "start_line"
[19] "end_line"                            "copyright"
[21] "holder"
```

You can see there is quite a lot going on under-the-hood with scancode. In particular look at those `license_match*` variables. These are columns in the `.csv` output file. They report how *sure* scancode is that it has found a match. We'll simplify this analysis and assume that results are reliable (and this is typically the case).

Here's a snippet of the controlling logic of the script:

```
if (length(args) != 1) { # incorrect number of cli arguments?
    cat(errorArgsNumber)
} else if (any(artefactFileNames %in% dir())) { # any pre-existing output files?
    cat(errorFileExists)
} else if (file.exists(args)) { # does the input data exist?
    generateArtefacts()
    cat(successMessage)
} else {
    cat(errorInputMissing) # if input data doesn't exist
}
```

It is not important to know what is going on here, but note that there is some error checking, and see the statement `any(artefactFileNames %in% dir())`, this is checking that there are no existing artefacts so that we don't accidentally overwrite any data. If all goes well with the checks, the `generateArtefacts()` function is called, shown here:

```
generateArtefacts = function() {
    sca_data <- dataPrep(read.csv(args[1])) # tidy the data
```

[10] Scripts: git reset –hard 9a36b9e79tbb9132c7

```
    ...
```

First, some effort is spent preparing the raw scancode data into something more reasonable, with `dataPrep()`, this is because scancode output is still pretty raw (only one-step above literal source code!). It also contains various fields we're not interested in, so those get cleared out:

```
dataPrep <- function(raw_sca) {
    # subset only relevant fields (1, 4, 20)
    # then delete empty rows (will have contained now-unnecessary data)
    out <- raw_sca[!(raw_sca$detected_license_expression_spdx == "" &
                  raw_sca$copyright == ""), c(1, 4, 20)]
    out[out == ''] <- NA
    aggregate(. ~ path, data = out, FUN = na.omit, na.action = "na.pass")
}
```

Next we use a helper function `outputFile()` a bunch of times, (once for each of the artefacts).

```
    ...
    # generate the artefacts - there is some unnecessary moving of data around
    # in memory, but this simplifies the script a bit (and it's sufficiently
    # performant anyway)
    outputFile(generateOverviewText(sca_data), "Licensing_Overview.md")
    outputFile(generateAppendixA(sca_data), "Licensing_Appendix_A.md")
    outputFile(generateAppendixB(sca_data), "Licensing_Appendix_B.md")
}
```

We noted earlier that scancode has a huge battery of tests, and the resulting data, while valuable, is messy. Tidying that up makes the remaining artefact generation quite simple. This will be the last code snippet we look at, but here's an example of generating the licence text artefacts:

```
generateAppendixA = function(tidy_data) {
    # tidying list of licences
    l <- unlist(strsplit(unlist(tidy_data$detected_license_expression_spdx), " AND "))
    l <- unique(l)
    l <- l[l != "N/A"]
    l <- l[-grep("License", l)]
    l <- gsub("[\\(|\\)]", "", l)

    # setup urls for downloading
    first <- rep("https://raw.githubusercontent.com/spdx/license-list-data/master/text/", length(l))
    last <- rep(".txt", length(l))
    dls <- paste0(first, l, last)

    # local "out" is a buffer, if one dl fails, the script will abort and may
    # leave artefacts
    out <- appendixAText

    # useful info for the user
    cat("\n\n Licence texts will now be downloaded...\n\n")
    Sys.sleep(2)

    for (l in dls) {
        download.file(l, destfile = "licence.tmp",
                      method = "wget",
                      quite = TRUE)
        out = paste0(out, "\n\n--------------------\n\n",
                     readChar("licence.tmp", file.info("licence.tmp")$size))
```

```
        file.remove("licence.tmp")
    }

    cat("\n\n Removing temporary files...\n\n")
    Sys.sleep(2)

    out
}
```

Notice that some further preparation of the data is done (which is specific for this artefact, so it makes sense to do it here) before raw licence texts are downloaded from a reputable repository[11]. There is then some logic to concatenate all the licences in a readable format into the single artefact file. There is also some cli output to let a user know that the internet is being accessed, this can be removed if the script is used in an automated pipeline.

The entire script is only a couple hundred lines long, and, in essence, we are simply extracting relevant data from the SCA results, and then wrangling it into a suitable output format. There are some quality of life-checks (ensuring that no data is overwritten). The file output is handled by a helper function elsewere in the script.

After running the script the working directory will contain the following files:

- Licensing_Overview.md
- Licensing_Appendix_1.md
- Licensing_Appendix_2.md

And we can view the contents of each of these files - they are in plain text. Here's the `Overview` in its entirety:

```
# Overview

This software contains a number of open source components. For a summary, see
below. For the relevant licence texts, see Appendix A. For the relevant notices
and attributions et cetera, see Appendix B.

Not all components listed may be incorporated in, or may necessarily have
generated derivative works which are incorporated in the firmware, but were
used during the build process.

Where a range of dates is given after a copyright notice, this should be taken
to imply that copyright is asserted for every year within that range, inclusive
of the years stated.

## Components and Licences

OpenBLAS - BSD-3-Clause
```

This is mostly boilerplate, but at the bottom there is a list of the components in the software. There's only one here, as we're reviewing a library with no transitive dependencies (OpenBLAS has some which you can link to, but we are assuming these are not used).

There's also the licence `Appendix`, whose generating function we looked at earlier:

```
# Appendix A: Licence Texts

--------------------
```

[11] There is some risk in relying on a third-party source for data like this (it may be taken down without your knowledge). The script will fail gracefully, but this is more of a cop-out and should be handled strategically.

```
Copyright (c) <year> <owner>.

Redistribution and use in source and binary forms, with or without modification, are permitted provided

1. Redistributions of source code must retain the above copyright notice, this list of conditions and tl

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED W


--------------------
```

Here's the BSD-3-Clause licence[12], there are others in the file. And lastly, here's the head of the copyright notice artefact:

```
# Appendix B: Notices and Attribution

---------------

File: OpenBLAS/benchmark/amax.c
Copyright statement(s): Copyright (c) 2016, The OpenBLAS Project

---------------

File: OpenBLAS/benchmark/amin.c
Copyright statement(s): Copyright (c) 2016, The OpenBLAS Project

---------------

File: OpenBLAS/benchmark/asum.c
Copyright statement(s): Copyright (c) 2014, The OpenBLAS Project
```

We can see that each file (demarkated by lots of dashes) is noted, alongside the copyright statements that were extracted for that file. There are

```
wc -l Licensing_Appendix_B.md
```

```
25296 Licensing_Appendix_B.md
```

... quite a few notices! Divide by five (each statement has surrounding whitespace, dashes, filename) and we calculate there are approximately five thousand copyright statements in OpenBLAS. [13]

Conclusion

We've looked at an end-to-end process for generating compliance artefacts from raw source code. Obviously this is a simplified process for this demonstration, there are countless nuances which could not possibly be documented [14]. This is the challenge of automating compliance.

[12] Apologies for the formatting, it is verbatim the BSD-3-Clause with long lines (which have no newline characters) and it's hard to wrangle that in a fixed width doc.

[13] Note that any files which lack copyright statements are not listed. This is acceptable as BSD-3-Clause obligates you to reproduce the statements, *not* the source code.

[14] Every time I do an analysis, there is something new

The heavy lifting can be left to SCA tools and scripts, as we've described above. But the devil is very much in the detail with compliance, and this requires some diligent analysis, preferably in the policy stage to prevent rampant OSS code from causing licensing issues right before a deployment.

Appendix A: Tools and Projects

ORT The open source review toolkit. This is a substantial project and is the closest an open source project has come to an "off-the-shelf" complete solution to compliance. That said, setup is considerable as it provides a kitchen-sink-first approach.

https://oss-review-toolkit.org/ort/

REUSE The REUSE specification is a simple amalgamation of a CLI tool and instructions for labelling the copyright attribution in source files. Integrating a REUSE compliant project into your projects will be straightforward from a compliance perspective. It also provides a framework for tracking the IP inventory of your code.

https://reuse.software/

Syft A good attempt to generate SBOMs and materials from Docker containers. It is not yet plug-and-play ready, however, containers are notoriously difficult for licence compliance so this tool may provide some benefit.

https://github.com/anchore/syft

Open Source Tooling Group Provides copious materials for adding knowledge to a compliance program. The materials are made available under an open source licence.

https://github.com/Open-Source-Compliance/Sharing-creates-value

Appendix B: Scancode installation

`cd` to the installation directory, then

```
git clone --depth=1 https://github.com/nexB/scancode-toolkit.git
cd ./scancode
./scancode --help
```

Running `./scancode --help` will setup scancode for first use. There are other methods in the documentation (https://scancode-toolkit.readthedocs.io/en/stable/) but the above is, clearly, very simple. The above can be copied into a `.sh` script, Docker `RUN` directive, et cetera, as required.

Appendix B: Intel's `proprietary-license` result

Here's a snippet of the LAPACKE readme

```
head -n 30 OpenBLAS/lapack-netlib/LAPACKE/README
```

```
--------------------------------------------------------------------------------
                            C Interface to LAPACK
                                  README
--------------------------------------------------------------------------------
Introduction
--------------------------------------------------------------------------------

This library is a part of reference implementation for the C interface to
LAPACK project according to the specifications described at the forum for
the Intel(R) Math Kernel Library (Intel(R) MKL):
http://software.intel.com/en-us/forums/showthread.php?t=61234

This implementation provides a native C interface to LAPACK routines available
at www.netlib.org/lapack to facilitate usage of LAPACK functionality
for C programmers.
This implementation introduces:
```

```
- row-major and column-major matrix layout controlled by the first function
  parameter;
- an implementation with working arrays (middle-level interface) as well as
  without working arrays (high-level interface);
- input scalars passed by value;
- error code as a return value instead of the INFO parameter.

This implementation supports both the ILP64 and LP64 programming models,
and different complex type styles: structure, C99.

This implementation includes interfaces for the LAPACK-3.2.1 Driver and
Computational routines only.

--------------------------------------------------------------------------------
```

This would normally require further analysis, and potentially the involvement of legal teams despite it being highly likely in the "INTEL TERMS OF SERVICE" that such code is licensed freely with the disclaimer.