Is Open Source Eating the World's Software? Measuring the Proportion of Open Source in Proprietary Software Using Java Binaries

Julius Musseau*
Mergebase
Canada
julius@mergebase.com

John Speed Meyers
Chainguard
USA
jsmeyers@chainguard.dev

George P. Sieniawski
IQT Labs
USA
gsieniawski@iqt.org

C. Albert Thompson
Ford Motor Company
USA
cthom409@ford.com

Daniel German
University of Victoria
Canada
dmg@uvic.ca

ABSTRACT

That open source software comprises an increasingly large percentage of modern software applications has become conventional wisdom. The exact extent to which open source software constitutes today's applications is indeterminate, however, at least by the standards of the academic software engineering research community. This paper proposes a methodology and associated tool that can analyze Java binaries and determine the proportion of open source that comprises them. This paper also presents empirical measurements of 5 commercial Java software systems, reporting OSS proportions between 76.2% to 99.9% among these 5 systems, including a historical analysis covering 6 versions and 12 years for one of the subject systems.

KEYWORDS

Open Source Software, Measurement, Methodology, Java, Binaries

ACM Reference Format:
Julius Musseau, John Speed Meyers, George P. Sieniawski, C. Albert Thompson, and Daniel German. 2022. Is Open Source Eating the World's Software? Measuring the Proportion of Open Source in Proprietary Software Using Java Binaries. In *Proceedings of Mining Software Repositories 2022: International Conference on Software Engineering (MSR '22).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

That modern software developers often use open source components to build applications and that open source software components have become pervasive is not in doubt. At the same time, the extent to which modern software applications contain open source components is indeterminate, at least for analysts who expect a rigorous and repeatable method and answer.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '22, May 21–29, 2022, Pittsburgh, PA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

Existing analysis on the prevalence of open source components in software applications largely derives from the commercial grey literature, along with a handful of academic studies [1–3] which we summarize in Section 2. As a prominent example of grey literature claims, a 2021 report by Synopsys titled "Open Source Security and Risk Analysis" includes an analysis of approximately 1500 client codebases and concludes that "75 percent of all codebases were composed of open source." [12] Sonatype, in a 2020 analysis, found that "80% to 90% of a typical application is composed of [open source] components." [11] These and other analyses, some survey-based (e.g. one done by Tidelift), have also examined the percentage of applications that contain any open source components and the average number of components found in typical applications. [14]

While these analyses are often based on large numbers of observations and use actual industry codebases, they frequently neither reference any tool for measuring the amount of open source content in a codebase nor describe a particular methodology. Additionally, these analyses often use proprietary customer data unavailable to outside researchers, which complicates generalizing the results.

Our goal is to measure the extent of open source code in a codebase, this paper proposes a methodology to measure the proportion of open source code in Java software by analyzing the fully-qualified class names of the classes that compose it and their corresponding sizes. We have developed an open source tool, called `contains-oss`,[1] that implements this methodology, enabling the researcher to calculate what percentage of a codebase is open source.

The next section discusses related work. Section 3 summarizes challenges associated with measuring the open source content of a codebase. Section 4 describes the methodology, followed by section 5, which covers the associated tool in detail. The sixth section applies the tool to several industrial Java codebases. Finally, Section 7 describes the limits and ambiguities of measuring the percentage of a codebase that is open source and proposes future research related to this topic.

2 RELATED WORK

Previous software engineering studies have sought to analyze open source code reuse using a wide variety of approaches. For instance, Dann et al. [3], examine how forking, patching, and "re-bundling"

[1] `contains-oss` is available at https://github.com/mergebase/contains-oss

existing code can have an adverse impact on the performance of OWASP Dependency-Check, Eclipse Steady, GitHub Security Alerts, and three additional commercial vulnerability scanners. Following analysis of over 7,000 Java codebases developed at a major European multinational, the authors present a four-part typology of code reuse as well as a novel test suite called Achilles. Importantly, in analyzing those codebases' direct and transitive dependencies, Dann et al. [3] rely on each software project's bill of materials (BOM), as generated by Eclipse Steady. As we discuss in Section 4, our methodology does not require the analyst to have access to a BOM. Instead, we focus on the fully qualified names (FQNs) of Java classes, which allow us to trace the origins of a given class to open source ecosystems like Maven-Central. Furthermore, although Dann et al. "checked how many classes on Maven have identical FQN[s] as ... 254 vulnerable classes and equivalent bytecode,"[3], their FQN analysis centers on Java re-compilation.

In addition, Bavota et al. [2] investigate how the Java subset of the Apache ecosystem evolved from 1999 to 2013. They focus on the change history of 147 Java codebases, including refactoring operations, license changes, developer discussions, and the stated rationale for bugfixes. Instead of analyzing the FQN of Java classes, however, Bavota et al. crawled "the SVN repository and identifie[d] the folder containing each of the project releases identified by the crawler" to find inter-project dependency listings [2]. In codebases that lacked explicit listings (37% of the total), Bavota et al. instead took the Levenshtein distance between Jar archive names and Java release names.

Turning to another related study, Bauer, Heinemann, and Deissenboeck distinguish between important third-party Java libraries, which "play a central role and thus have a significant impact on [Java project] maintenance"[1] and unimportant third-party Jars. This distinction, in turn, hinges on the number of distinct API method calls to a library, the percentage of Java classes "affected" by a call when traversing the abstract syntax tree (AST) of the codebase, and the ratio of API utilization to the total number of API methods available, among other static analysis metrics [1]. The authors then proceed to explain how to implement these AST traversal methods and subsequently visualize the results in ConQAT, a software quality assessment toolkit. However, Bauer, Heinemann, and Deissenboeck acknowledge their method does not consider "libraries that are indirectly used via other libraries," instead putting the onus on the analyst to consider transitively referenced Jars.

Heinemann et al. have previously investigated the extent of code reuse in open source Java codebases.[5] Ruiz et al. in two separate studies analyze the Android marketplace, especially class reuse, finding widespread evidence of reuse.[8][9]. German and Di Penta propose a method for open source license compliance.[4] Scwittek and Eicker analyze third party component reuse in Java open source software by measuring the number and provenance of open source components in nearly forty open source Java web applications.[10] Ishio et al. propose and evaluate a technique for detecting third-party components in Java releases.[13]

3 METHODOLOGY

Measuring the percentage of open source code within a proprietary codebase presents two fundamental hurdles. First, researchers often only have access to a compiled binary, not source code. Accordingly, any potential methodology for measuring open source content must deal with this severe limitation. Second, partitioning a codebase into mutually exclusive proprietary versus open source sections is challenging. Modern applications, no matter the language, often do not explicitly mark particular files or modules as "open source," though some informal conventions associated with select programming languages and programming practices can aid such a partitioning. Our approach is based on the following observations regarding binary Jar files:

(1) Every entity in a binary Jar has a fully qualified name (FQN), which, in general, uniquely identifies the entity within the universe of Java software. In most cases, an analyst can use this FQN to trace back the origin of the entity. For example, the class PatternLayout, part of Apache's log4j-core library, has the FQN *org.apache.logging.log4j.core.layout.-PatternLayout*. The first two components indicate that it originated in the Apache organization, the next 3 components indicate the specific library. Thus one can use the FQN of an entity to determine if such entity is OSS or not.
(2) With few exceptions, every class and method present in the source code of a Java file is compiled into its Java binary, and using its FQN, it is possible to map one to the other.
(3) In addition, most Java binaries are distributed with debug information[2]. This debug information contains line number information. It is therefore possible to quantify the number of lines of a source code file. It is also possible to know which lines of this source code file are actually lines of code (as opposed to comment or empty lines) since every line of code will be referenced in the debug information of a binary.

Our method has two steps, the first of which is optional.

3.1 Preliminary step (optional): create a list of open source entities and their FQNs

Using Maven-Central it is possible to create a list covering a very large proportion of all FQNs in the known OSS Java universe[3]. Note that, in some cases, the name of the organization might be sufficient to identify whether the entity is OSS or not (e.g. *org.apache*). However, this is not always the case; for example, Jira (*com.atlassian.jira)* publishes some entities in Maven-Central, but it also uses this prefix for its own proprietary software. The process to create this list is straightforward:

- For every binary Jar in Maven-Central:

(1) Extract all the .class files in the Jar
(2) Add each extracted FQN from the Jar to the list of known OSS FQNs

Note that this method assumes that all versions of a library are OSS. For example, if earlier OSS versions of the library were published to Maven-Central, while later private proprietary versions were not, then no assertion can be made regarding whether a

[2]A developer must explicitly configure Maven or Gradle to not include debug symbols. In addition, because developers find Java stacktraces with line numbers much easier to work with, developers are further incentivized to leave debug symbols enabled
[3]Additional public Maven artifact repositories could be added (such as spring-central) to increase coverage, but due to its popularity we suspect Maven-Central likely contains at least 90% of all known OSS Java FQNs

given FQN is OSS or not, at least not without also further assessing version information. However, to simplify our methodology, within this study we assume that if one FQN is identified as OSS, then all versions of that same FQN are also OSS, despite some rare situations where this assumption does not hold. Finally, truly determining whether a library itself is OSS is complicated and can depend on numerous factors and even different definitions of "OSS" (e.g., OSI-approved, SPDX-designated or other criteria). However, for this study, we considered a library's current presence (circa December 2021) and availability for download from Maven-Central as sufficient indicia of it being OSS. For example, we considered Oracle's "ojdbc10-19.3.0.0.jar" file to be OSS, despite the fact it uses a relatively OSS-incompatible license: "Oracle Free Use Terms and Conditions (FUTC)". There are two reasons for using this definition of OSS: First, the fact that Maven-Central themselves are permitted to redistribute the Jar file signifies a certain degree of permissive reuse. Historically, Maven-Central has removed many Jar files from their archive upon request from copyright holders. Second, since the goal of this study is to measure how much of a given software system is internally developed by its development team vs. externally imported and re-used as OSS, we believe that categorizing any Jar file that successfully downloads from Maven-Central as OSS (e.g., "ojdbc10-19.3.0.0.Jar") made sense for analysis purposes.

The complete list of 3,600,000+ distinct FQNs we extracted from Maven-Central is included in our replication package inside the file "names.uniq.gz". This list also includes all FQNs observed within several JDK versions and variations (JDK 1.6.0 to JDK 17, Oracle and OpenJDK), since it would not make sense to categorize these as internal in cases where they appeared inside software systems. This list also included FQNs derived from inner-classes, as well as Java bytecode generated from JVM-compatible languages such as Scala, Clojure, Kotlin, etc. Since our technique counted OSS proportion using bytecode analysis, our technique was not limited to only Java. Any language that compiles to JVM bytecode could be analyzed.

3.2 Processing the Java binary to be analyzed

For input, this method requires a Jar file. Crucially, this method does not require access to the source code or to a Software Bill of Materials (BOM), as in [3].

(1) For each class in the binary Jar:
   (a) Determine if it is open source or not using its FQN.
   (b) Measure it (e.g., by observing the largest line number present within its debug symbols)

Classifying a class as OSS or not can be easily done with a pre-computed set of FQNs that are known to be OSS. Alternatively, an analyst can look up the main prefixes present in the Jar; depending on the number of different libraries present in the Jar, it might be straightforward or time-consuming to classify the FQN prefixes into OSS and not. Sometimes developers copy OSS libraries into their source code tree to simplify dependency management and deployment. In most cases (and to avoid costly renames), the developers maintain the same file structure of the library. Thus, the FQN of an embedded library has a suffix equal to the FQN of the library. E.g. a log4j class might have the FQN *com.myorg.depend.org.apache.-logging.log4j.core.layout.PatternLayout.* In cases like this using a human analyst will be more accurate, since a pre-computed set of known-OSS FQN's is unlikely to contain these (artifacts like this are unlikely to be published in Maven-Central).

Once the FQN has been classified (as OSS or proprietary), measuring the class file can be done in various ways. A class file always corresponds to exactly one source code file (though not the other way, because of inner-classes). It is also easy to measure the number of methods in such a class. If debug information is present, it is possible to extract, at the very least, the largest line number referenced, which will correspond to the number of lines in the original .java file. It is also possible to extract every line number referenced in the binary Jar (if the line is referenced in the binary, it must be a SLOC in the original source code–and not an empty line), however this method will under-count certain types of lines of code, such as those that split a long statement, those with only braces, and many others. Nonetheless, research has shown all these metrics tend to correlate strongly [7].

Since our goal is to measure the proportion of OSS in proprietary software, we are less interested in an exact measure of size, and more on a comparison between the two subsets. Thus, any threat to validity in the measurement of size is likely to be the same for both subsets.

Our replication package includes a sub-directory called "replication" where we stored the raw data results in JSON files from running the `contains-oss` tool against the subject systems. Note: The *Ford Motor Company* ran the `contains-oss` tool, but was not willing to share replication data beyond the total number of lines observed for each of their subject systems. All subject systems were analyzed using the pre-computed set of OSS FQNs (based on Maven-Central), except for the Ford systems where a human analyst manually conducted the partioning into OSS and non-OSS based on names within the FQNs themselves (e.g., those that contained "com.ford" vs. those that did not).

4 TOOL: CONTAINS-OSS

```
1  "atlassian-oauth-plugin-1.0.8.1.jar":{
2    "lines.internal":2466,
3    "lines.external":4588,
4    "breakdown.internal":{
5      "com.atlassian.oauth.bridge":386,
6      "com.atlassian.oauth.provider":1826,
7      "com.atlassian.oauth.shared":254
8    },
9    "breakdown.external":{
10     "net.oauth":1417,
11     "net.oauth.client":505,
12     "net.oauth.http":356,
13     "net.oauth.server":245,
14     "net.oauth.signature":1503,
15     "net.oauth.signature.pem":562
16   }
17 }
```

Listing 1: Sample output after invoking the contains-oss.jar tool - in this case both internal (proprietary) and external (open-source) code is mixed inside a single Java binary Jar file found within Atlassian Jira 4.1.1.

This paper introduces `contains-oss`, a GPL-licensed software found at https://github.com/mergebase/contains-oss. `contains-oss` is a command-line tool that measures the proportion of OSS software in a binary Java (a Jar file). It implements the methodology

described in the preceding section. It takes as input a directory containing Jar, Zip, and War files (at any sub-directory depth) and a listing of known OSS FQNs. From these inputs it computes the proportion of lines of code (if debug information is available) or proportion of methods (otherwise) that are OSS in the Jar. An example of its output is presented in Listing 1.

5 RESULTS AND ANALYSIS

As a test of this methodology, we selected various industrial applications from 3 different organizations: 3 from *Ford Motor Company*[4], 1 from *Mergebase*[5] and 1 from Atlassian[6]. In all cases we only used the binary Jars of each application.

Figure 1 shows the results of `contains-oss` applied to five different applications. The percentage of the applications that are open source ranges from 76.2 percent to 99.9 percent.

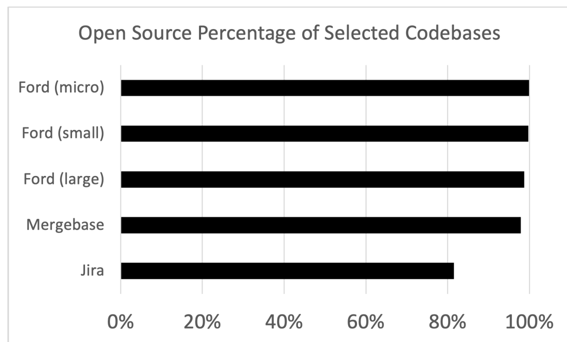

Figure 1: Percentage of OSS across selected applications
Note: All *Ford* applications results were provided courtesy of *Ford Motor Company*.

We observed that, for the *Ford* applications, the string "*Ford* " was sufficient to determine which FQNs where proprietary (i.e., authored by *Ford* developers). For the *Mergebase* and the Atlassian applications we used our pre-computed set of Maven-Central FQNs to determine whether an artifact was proprietary or OSS.

Table 1 report the results of an analysis of different versions of Jira. The percentage of the codebase that is open source declines from 86 percent in version 4.1.1 to a low of 76 percent in version 8.0.2. These measurements span from April 2010 (4.1.1) to October 2021 (8.20.0). The dip in lines of code between version 8.0.2 and 8.20.0 is due to the removal of unnecessary OSS dependencies.

Figure 2 shows open source measurements of Jira before and after installing and setting up Jira. We performed this analysis to examine whether Jira itself brings in additional logic during its installation and initialization processes. The measurements are nearly the same: 78 percent pre-setup (after downloading and extracting 2021-10-18-atlassian-jira-software-8.20.0.tar.gz) compared to 79 percent post-setup (after getting Jira to launch and confirming its correct operation).

These analyses suggest that the proportion of open source in proprietary Java applications is very high, but it also varies a lot:

Release	Date	LOCS	Proportion of OSS
4.1.1	April 2010	6,988,990	0.86
5.0.1	March 2012	10,581,126	0.83
6.0.1	May 2013	11,626,264	0.82
7.0.2	November 2015	15,887,765	0.77
8.0.2	March 2019	15,550,557	0.76
8.20.0	October 2021	19,339,034	0.78

Table 1: Different releases of Jira, their size and proportion of OSS in them.

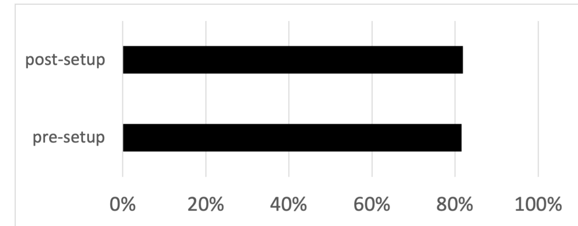

Figure 2: Open Source in Jira Pre- and Post-Set Up

from around 76% to 99%. Over time, the proportion of OSS can vary due to many factors: in the case of Jira, cleaning up dependencies; in others it might be adopting new libraries as the features of the application grow.

6 DISCUSSION

The methodology and the associated tool described in this paper (`contains-oss`) is preliminary and its creation and proposal leaves several open questions. The creation and application of `contains-oss` to our own products has reinforced our belief in the sore need for more conceptual clarity and associated measurements for repeatedly and transparently measuring the extent to which an application is open source. This section forthrightly describes the limitations of this methodology and `contains-oss` and then makes a call for a research agenda and tool development to overcome the current limitations of this methodology and tool.

First, what is the best definition of proportion of open source in a given application? As we noted before, Sonatype reports that "75 percent of all codebases were composed of open source"[11]. Sonatype does not publish its methodology, thus we do not know how it came to this conclusion.

However, most OSS used in proprietary software are libraries. And most software does not use all the functions/methods/class in a library. In fact, a library may exist in the version control repo of an organization, along with the proprietary software. Without understanding the build process, it is difficult to know whether the library would be part of the binary or not. The lack of clarity of Sonatype's and similar reports make it difficult to assess.

Our method does not perform static nor binary analysis to determine which proportion of the OSS is actually used. Any small application will be dwarfed by the size of any OSS libraries that it uses; thus, it is unsurprising to see reports that OSS comprises the majority of current proprietary software.

[4]Note: All *Ford* application results were provided courtesy of *Ford Motor Company*.
[5]http://mergebase.com
[6]Atlassian.com

6.1 Regarding threats to validity of our proposed method

There are five notable limitations to `contains-oss`. First, the tool currently only applies to Java codebases. Second, in the absence of a list of OSS FQNs, the user must designate a set of strings that denote which files were authored by the party that produced the software. This requires expert knowledge and cannot be easily applied at large scale. This design choice also reflects an engineering practice used by Java developers and so will not be applicable across all programming languages. Third, the line counting procedure is itself approximate, relying on breakpoints introduced by the build tool, and undercounts the total number of lines in a file. Fourth, for files not built with the debug flag set to `true`, `contains-oss` measures and compares other binary entities. Fifth, should a developer copy open source code into their codebase, there exists the possibility that the Java Jar file labeled as proprietary will, in fact, include open source code.

That said, we do not know of a tool with similar functionality and offer `contains-oss` as a first step toward a broader research agenda. Ideally, a tool designed to measure the extent to which a codebase contains open source code would work across languages and would contain a number of approaches to calculating the extent to which a codebase is open source. Such approaches would include counting the number of:

- instructions
- lines (as `contains-oss` currently does)
- components
- functions
- open source lines actually executed (versus any open source code which is bundled but never used)

6.2 Regarding support for other languages

Each programming language has its own idiosyncrasies regarding how to build and bundle dependencies. Our method assumes no dynamic linking, thus both open source and proprietary entities are included in the binary. Even though not all languages have the notion of a universal FQN, it might still be possible to create listings of identifiers (at different levels of granularity) that might be identifiable in the binary. The level of information available in the binary will also vary from language to language and from developer to developer: C/C++ have been usually stripped of debug information; Javascript is often obfuscated. Accordingly, more research is needed to determine information analysts can use to identify OSS entities in a given application (such as the use of strings to identify origin or source code entities[6]).

There is ample room for future researchers to select a larger and representative corpus of codebases so that researchers can attain an accurate global estimate of the prevalence of open source code in software applications. In the meantime, these analyses demonstrate the possibilities and complications of such an approach. Finally, this research agenda should include a methodology and data source for selecting a large set of representative industrial applications (even if in binary form only) so that analysts can apply such a tool and generalize results.

7 CONCLUSIONS

Measuring the extent to which the world's software is composed of open source software is possible, but contrary to appearances, is more difficult to calculate than commonly understood. The challenges include creating measurement tools that can accept binaries, not just source code, as input and partitioning a software artifact into open source and proprietary sections. This paper describes one methodology and associated tool for those who to undertake such a measurement project.

Our development of this methodology and tool suggests that the task is possible, even at a large scale.

From a purely empirical point of view, this paper contributes a small collection of important data-points for software engineering researchers, especially those focused on open-source software and its reuse. We observed a high percentage of OSS present in a mix of commercial Java systems, ranging from 76% to over 99%. We also observed how smaller systems appear to use a higher percentage of OSS. While most software engineering researchers and developers rightly recognize that OSS constitutes the majority of software present in most modern systems, we believe this paper is the first to provide an empirical repeatable measurement.

REFERENCES

[1] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. A structured approach to assess third-party library usage. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 483–492. IEEE, 2012.

[2] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[3] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. IEEE, 2021.

[4] Daniel German and Massimiliano Di Penta. A method for open source license compliance of java applications. *IEEE Software*, 29(3), 2011.

[5] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. On the extent and nature of software reuse in open source java projects. In *International Conference on Software Reuse, 2011.(ICSR)*, 2011.

[6] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 63–72. ACM, 2011.

[7] Israel Herraiz and Ahmed E. Hassan. Beyond lines of code: Do we need more complexity metrics? In Andy Oram and Greg Wilson, editors, *Making Software - What Really Works, and Why We Believe It*, Theory in practice, pages 125–144. O'Reilly, 2011.

[8] Bram Adams Israel J. Mojica Ruiz, Meiyappan Nagappan and Ahmed E. Hassan. Understanding reuse in the android market. In *20th International Conference on Program Comprehension*, 2012.

[9] Meiyappan Nagappan Steffen Dienst Thorsten Berger Israel J. Mojica Ruiz, Bram Adams and Ahmed E. Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2), 2014.

[10] Widura Schwittek and Stefan Eicker. A study on third party component reuse in java enterprise open source software. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, 2013.

[11] Sonatype Inc. The 2020 State of the Software Supply Chain. https://www.sonatype.com/2020ssc, Aug 2020.

[12] Synopsys Cybersecurity Research Center. Open Source Security and Risk Analysis Report. https://www.synopsys.com/software-integrity/resources/analyst-reports/2020-open-source-security-risk-analysis.html, May 2021.

[13] Tetsuya Kanda Daniel M. German Takashi Ishio, Raula Gaikovina Kula and Katsuro Inoue. Software ingredients: Detection of third-party component reuse in java software release. In *IEEE/ACM 13th Working Conference on Mining Software Repositories*, 2016.

[14] Tidelift. How to make open source work better for everybody. https://tidelift.com/about/2018-tidelift-professional-open-source-survey-results, July 2018.