IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 August 2023, accepted 27 September 2023, date of publication 5 October 2023, date of current version 13 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3322362

RESEARCH ARTICLE

Assessing the Real Impact of Open-Source Components in Software Systems

ANDY MOLIN, ANDREI MARIO RIVIŞ, AND RADU MARINESCU
Faculty of Automation and Computers, Politehnica University Timişoara, 300223 Timişoara, Romania
Corresponding authors: Andrei Mario Riviş (mario.rivis@upt.ro) and Andy Molin (andy.molin@upt.ro)

ABSTRACT Open-source libraries form the backbone of modern software systems, making software composition analysis (SCA) a vital part of the software development cycle. Despite its importance, current SCA methods, primarily focusing on open-source component issues, lack comprehensive analysis of these components' integration into the software system. This paper proposes an advanced SCA approach that simultaneously considers open-source component issues and their integration into a software system. We introduce a novel meta-model that links a library with its source code dependencies and enables a unified analysis, irrespective of the originating package manager or open-source repository. The proposed approach, instantiated through a code analysis tool and adapters for major package managers and repositories, was applied to over 200 popular GitHub projects. Results confirm that the impact of open-source component issues largely depends on their integration level in the software system, validating our assumption that effective risk management requires understanding of the open-source component use within the system. Our work, therefore, provides an enriched methodology for SCA.

INDEX TERMS Inspect, dependency, library, age, vulnerability.

I. INTRODUCTION

Open-source libraries and frameworks are at the core of any modern software system. However, the software engineering community has growingly become aware that the usage of open-source components comes with significant risks, the most important one being vulnerabilities that allow attackers to harm the systems by using vulnerable components. Many vulnerabilities are caused by a project relying on an outdated version of an open-source component, which is sometimes caused by negligence, but other times it is caused by being stuck with an old library that was modified and on which many other modules depend, which makes it hard to upgrade the library, an issue known as the *Evolution Trap*. Licensing is another concern when using open-source libraries. Licenses can be more permissive or more restrictive. The restrictive ones, copyleft licenses as they are known, might bring problems in a system. Copyleft is a method for making a software program free, requiring that all modified and extended versions of the program to be also free and released under the same terms and conditions.

In this context, *Software Composition Analysis* (SCA) is increasingly playing an important role in the development cycle of software systems, as it can detect important issues like security vulnerabilities, licensing problems or outdated library versions. Although there are tools which offer solutions, in most cases, each concern is treated separately, leaving a gap within the information about the analyzed libraries. Moreover, in order to accurately assess the risks incurred by these issues on a specific software system, it is equally important to understand the extent to which the system depends at the code level on these open-source components. Unfortunately, current composition analysis approaches are too much focused on the issues found in the open-source component, and neglect to analyze where these components are actually used in the software system that is inspected.

In this paper we introduce an enhanced, tool-supported, approach to *Software Composition Analysis* that aims to contribute at addressing the aforementioned shortcomings. The presented approach takes into account both the issues

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

associate with the open-source component *and* the degree to which an inspected software system relies on that component. At the core of the approach is a meta-model for composition analysis with two novel traits: (i) it associates a library with its source code dependencies and (ii) it provides a unified way of analyzing the wealth of information coming from the different package managers, open-source repositories or composition analysis tools that a software system may use.

To instantiate the model with data from a broad range of projects we developed a code analysis tool (*Inspector-Lib*) which contains adapters for fetching meta-data about open-source components for several major package managers. *Inspector-Lib* also contains a component that detects library usages in source code files.

We applied this approach to over 200 popular open-source projects from GitHub to test the feasibility of the data extraction tools on heterogeneous projects. We also conducted a more in-depth study on a selected subset of 28 projects to assess how much projects depend on open-source components that present significant risks, which translates into the maintenance effort that would be required to address the risks from these open-source components.

The results show that the real impact of issues in open-source components varies significantly depending on the extent of the component's integration into each software system. This validates the assumption that in order to act efficiently on the risks incurred by open-source components, we must go beyond the analysis of the component itself, and understand better how the component is used in a software system. Hence, this paper contributes to software composition analysis by providing a broader and more comprehensive method for assessing and mitigating the risks associated with the use of open-source libraries.

The remainder of this paper is structured as follows: after presenting alternative approaches and their limitations in Section II, we discuss our library assessment approach in Section III and illustrate it on a running example. In Section IV we present our two evaluation studies, and then we present our conclusions and future work direction in Section V.

II. RELATED WORK

A. APPROACHES

As previous research, which will be presented in this section, has shown, knowing and understanding a system is very important when it comes to maintaining and evolving it. To help the system, we must first understand it, how it is build, what dependencies it has and what the risks are of having them.

The biggest threat when it comes to third-party libraries are vulnerabilities, which is a frequently discussed topic [1], [2], [3], [4], [5], [6], [7], [8]. They are one of the most pressing problems in open-source libraries. It may be difficult to find and fix them. Also, they may propagate to other packages, making them vulnerable too. Zapata et al. [1] offer an interesting study of vulnerable dependency migrations for npm packages. Developers are encouraged to maintain and update any outdated dependency to remain safe from potential threats including vulnerabilities. In their study they manually inspected 60 client projects from three cases of high severity vulnerabilities and investigated whether or not clients are safe from these threats. Plate et al. [2] propose a pragmatic approach to facilitate the impact assessment, describe a proof-of-concept only for Java and examine vulnerabilities as a case study. In their study they analyze a Java system to see whether the affected class (referenced to in the description of the vulnerability) is used. Pashchenko et al. [3] present a methodology for a correct allocation of development resources. They discovered that a majority of vulnerable dependencies may be fixed by simply updating to a new version, while only a small percentage of vulnerable dependencies require a costly mitigation strategy. Decan et al. [4] study vulnerabilities in npm packages. They analyze how and when they are discovered and fixed, and to which extent they affect other packages, providing guidelines for package maintainers and tool developers to improve the process of dealing with security issues.

Combining different libraries may also be a problem. Some libraries may not be compatible with a certain version of other libraries, as presented in [9], [10], [11], and [12]. In large software systems, it is common practice to adopt third-party libraries. Therefore incompatibility between library dependencies may occur and complicate the system's development and maintainability. Yano et al. [9] present a tool that offers an interactive data visualization of popular library version combinations. Using the ''wisdom of the crowd'', Yano et al. created a tool to assist system maintainers by offering data visualization of the best library version combinations and to help them make the best library usage decisions by showing the ''best-fit'' result of library links. Decan et al. [10] use a data set to carry out an empirical analysis of the similarities and differences between the evolution of package dependency networks for seven packaging ecosystems of varying sizes and ages. They observe that the dependency networks tend to grow over time, both in size and in number of package updates (only a minority of packages are responsible for most of the package updates), and that the majority of packages depend on other packages. With their analysis they provided important information about the update tendency and the relations of different packages.

Also, when working on large software systems, developers may be unaware of the risks the libraries hide in their system [13], [14]. Studies show that developers don't look at the version of the library that they are using, leaving them with an old version which may be vulnerable. This is an unnecessary risk, because the majority of vulnerable dependencies may be fixed by simply updating to a new version.

Always having the latest version can help to get rid of vulnerabilities. The system must be ''fresh'' to avoid security risks. Systems using outdated dependencies are four times as likely to have security issues as opposed to

systems that are up-to-date. Cox et al. [15] introduce a metric to quantify the use of recent versions of dependencies, referring to it as the system's ''dependency freshness''. They validate the usefulness of the metric using interviews and investigate the relationship between outdated dependencies and security vulnerabilities. Their results show that systems using outdated dependencies are four times as likely to have security issues as opposed to systems that are up-to-date.

Wang et al. [16] conduct a library risk analysis, which aims to quantify the potential risk of using outdated libraries and the developer response to the risk. They conduct a comprehensive study on third-party library usages, updates, and risks in the Java ecosystem, offering practical insights for developers, proposing a bug-driven alerting system for informed decisions on library updates, and highlighting the importance of addressing maintenance challenges in software development.

B. TOOLS

The concerns presented in the previous subsection also impact the industry. To detect such concerns, tools have been developed for industrial use, to analyze the dependencies of a system [17], [18], [19], [20]. There are several ones which treat some parts of Software Composition Analysis, such as vulnerabilities, licensing and version updates, but each independent and being focused on particular issues.

They find security vulnerabilities, eliminate the risk of open-source license noncompliance, inspect the source code but also the binaries and automatically monitor for new vulnerabilities [17]. Or find and fix security vulnerabilities in the code [18]. They keep your dependencies up-to-date by looking in the dependency files and searching for any outdated dependencies [19]. And they can automate the process of open-source component selection, approval and management, including security detection and remediation, and compliance issues [20].

A summary of the tools and their functionalities can be seen in Table 1.

TABLE 1. Tools.

Tools	Vulnerabilities	Licensing issues	Version Updates
Black Duck	X	X	
Snyk	X		
Dependabot	X		X
WhiteSource	X	X	

C. LIMITATIONS OF THE STATE-OF-THE-ART

As presented earlier, the concerns are treated in isolation (only vulnerabilities, only age), there is no unified way and synergies between different parts of Software Composition Analysis are not possible. It is very hard to make a decision without having all the information of the dependencies in one place. To better understand and to have a bigger picture of the dependencies in a system, it is important to have all the information together.

Even more, it is not only about risks and vulnerabilities, it is also about productivity, about efficiency and how developers spend their time managing dependencies. The emphasis is on the libraries themselves, and less on their context in the system. That is why the effort of maintaining the dependencies is unknown, because it is difficult to know something about it without looking at the actual ''footprint'' of the libraries in the system. To act properly, one needs to understand the dependencies within the context of the system.

Knowing the age, or ''the freshness'', and the vulnerabilities of the used libraries is very important, but this is not enough insight. To better understand a system and its structure we have to look where these libraries are used, which components of the system are vulnerable, which file contains the vulnerable dependency, which parts of the system are the ''old'' ones. The idea is to connect the different worlds of Software Composition Analysis and reduce the gap between them through an ''end-to-end'' approach to analyzing the dependencies of a software system and providing a better decision-making process regarding the system's dependencies by understanding the ''footprint'' of each dependency.

TABLE 2. Limitations in cross-examination.

Limitation	Description
Isolation of Concerns	Current approaches treat concerns in isolation, focusing solely on vulnerabilities, age of dependencies, without considering synergies.
Lack of Unified Information	There is no unified way to gather and present all dependency information in one place, making it challenging to make informed decisions regarding dependencies.
Neglect of Context	The emphasis is mainly on libraries themselves, overlooking their context within the system. This results in an unknown level of effort required to maintain dependencies, as their "footprint" in the system is not adequately understood.
Insufficient System Insight	While knowledge of library age and vulnerabilities is crucial, it offers limited insight. Understanding where libraries are used, identifying vulnerable components, and pinpointing files with vulnerable dependencies is necessary for comprehensive analysis.
Limited Decision-Making Support	Without a complete understanding of the "footprint" of each dependency, decision-makers lack the necessary information to act effectively and make informed choices regarding a software system's dependencies.

III. INSPECTOR-LIB: THE ASSESSMENT APPROACH

This section will present *Inspector-Lib*, our tool-driven approach that allows for a context-aware analysis of open-source components. As shown in Figure 1, the approach consists of three elements that will be introduced next: (i) the *meta-model* which is the core of our approach as it specifies the pieces of information that we analyze and how these pieces are interconnected; (ii) the *open-source component profiler* that identifies the open-source components used by the analyzed system, fetches key information about these components from the various package managers, repositories

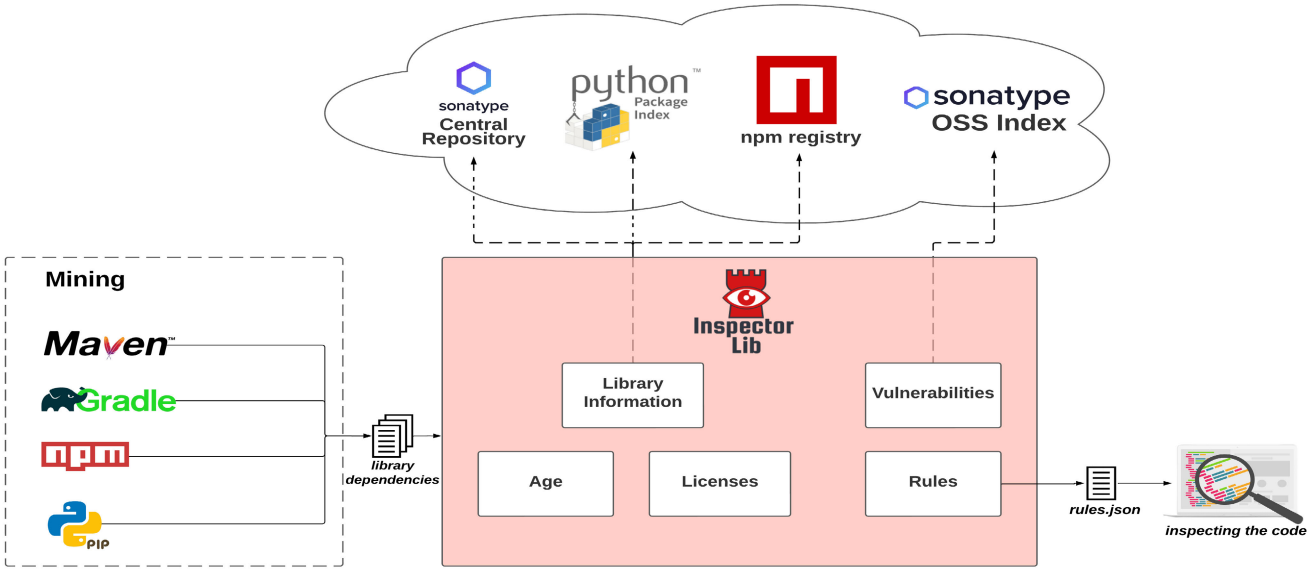

FIGURE 1. Inspector-lib system overview.

and/or composition analysis tools, and stores it in our model; and (iii) the *open-source code scanner*, which searches the entire codebase for ''traces'' of open-source components, using a set of rules based on regular expressions.

A. MODELING OPEN-SOURCE COMPONENTS

Most software projects are implemented nowadays using several programming languages; for instance a language like Java for the backend, another language like JavaScript for the frontend and a scripting language like Python as ''glue'' code. Each of these languages is managing its dependencies in a different way, using a different package manager. This raises a challenge when it comes to analyzing efficiently open-source dependencies in multi-language projects. Therefore, we designed a *unified meta-model* that will allow the analysis of all the open-source dependencies of a project in a unitary manner.

As depicted in Figure 2 this meta-model is based on the idea of capturing the *key elements* that are relevant for any open-source dependency regardless of the programming language that it serves or the dependency manager that is used by the project. Our meta-model captures four types of information about each dependency used by the project:

1) *Library Identifier*: the dependency name, version identifier, and a reference to its origin including the providing package manager and the *purl* which identifies each open-source package uniquely.
2) *Library Meta-data*: detailed descriptions, licensing information, complete release histories (encompassing version numbers and release timestamps), relevant keywords, homepage URLs, and more. All these are extracted from the package managers, based on the basic information. Moreover, using the basic information about the library we store in the model information about the vulnerabilities that the dependency may have. The details about how the meta-data is fetched are described in Section III-B2.
3) *Code References*: the list of fully qualified names of source files that reference the open-source component. This information is extracted using the automated technique described in Section III-C.
4) *Derived Metrics*: By using the meta-data described above we automatically compute several key metrics that are used in the risk analysis described in Section III-D. For the age of a library, we store the time from current used library version to latest released version, time from current used library version to current date, and time from latest released version to current date. Each duration is computed in months. For vulnerabilities, we count how many vulnerabilities each library has, compute the total and the average vulnerability score for each library and save the references for each found vulnerability.

The meta-model decouples the analysis part from the heterogeneity of fetching the data from multiple sources, allowing us to define any type of analysis in a unitary manner. Moreover, this approach allows us to extend *Inspector-Lib* towards more languages and more package managers, as the extension would just require writing the adapters that extract the meta-data, without requiring any changes to the risk analyses themselves. This makes the approach easy to adapt and scale.

B. PROFILING OPEN-SOURCE COMPONENTS

1) IDENTIFYING THE OPEN-SOURCE COMPONENTS

The dependencies must first be extracted from the analyzed system. The used libraries can be found in the configuration files of the package managers. The configuration files and the way the dependencies are declared differs from one package manager to another. For example, the configuration file for

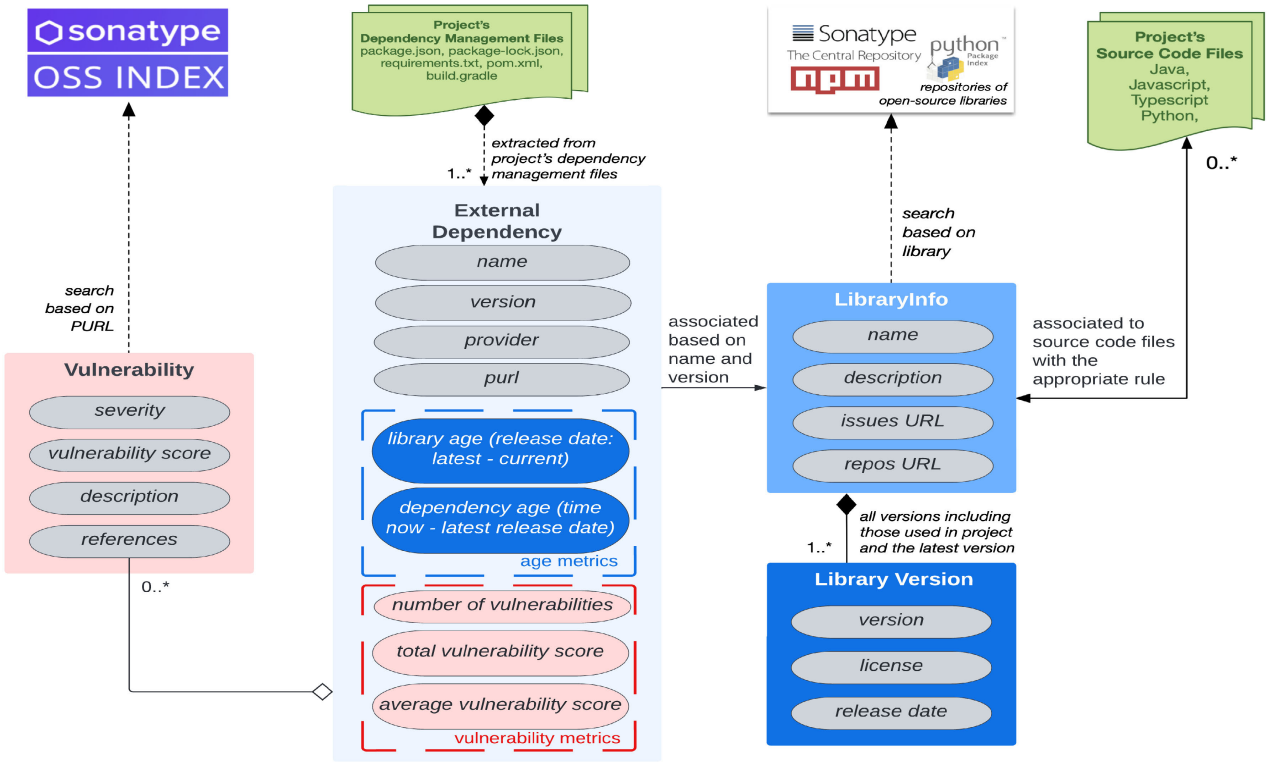

FIGURE 2. Inspector-lib model.

Maven is a XML file named *pom.xml*, and for *Gradle* it is the *build.gradle* file, each package manager having its own configuration file. The dependencies are extracted by parsing these files. After the extraction is completed and the dependencies are found, the model population step begins. Here is an example of declaring dependencies in a *pom.xml* file:

```
<dependencies>
  <dependency>
    <groupId>me.tongfei</groupId>
    <artifactId>progressbar</artifactId>
    <version>0.7.2</version>
  </dependency>
</dependencies>
```

After parsing this `pom.xml` file the presence of the `me.tongfei progressbar` library, version `0.7.2` can be observed. The next step is extracting information about the libraries found in the configuration files and populating the model with data.

2) FETCHING OPEN-SOURCE META-DATA

Information about the found libraries must be extracted. This is done by making requests and getting the data as a response, which is parsed and added to the model. Each third-party software repository has its own API. To get the information, the correct URL must be created to obtain the data for each found library.

The response obtained contains information about the specific library which was requested, structured as a JSON. The response is parsed and the model is populated with the information from the request. For example, let us assume we need information about `tensorflow`, an open-source software library for high performance numerical computation used in Python. To obtain the information about this library a request to the Python Package Index (PyPI),[1] the official third-party software repository for Python, has to be made. This will result in getting a response, as a JSON, containing information like: description of the library, license, all releases of the library (version and release time of each version), keywords, homepage, name of the library, and so on. After parsing the JSON response, the model will be populated with the obtained information. The same approach is used for libraries of different programming languages: make a request to the third-party software repositories, obtain the response, parse it and populate the model with the obtained information.

To get details about vulnerabilities, *Sonatype's OSS Index API* is used. The OSS Index contains plenty of identified vulnerabilities of open-source components from the NVD (National Vulnerability Database) and other open-source vulnerability databases. *Sonatype OSS Index* uses a *package-url* specification to describe the coordinates of the components / packages. So, to find a library, its *package-url* must first be made to use it as a coordinate to get information. The information is then parsed, the vulnerability metrics (presented in the section above) are computed and then added to the model. For example, the *package-url*

[1] https://pypi.org/pypi/tensorflow/json

(purl) for the *npm* library `foobar` version `12.3.1` is: `pkg:npm/foobar@12.3.1`.

C. OPEN-SOURCE CODE SCANNER

Generating rules helps in detecting where in the code and, in which files, the libraries are used. The idea of the rules was inspired from Microsoft's Application Inspector.[2] To find the usage of the libraries, different patterns are generated in the rules. For example, let us take a look at the rule generated for the *junit* library.

```
{
  ''tags'': [ ''testing'' ],
  ''patterns'': [ {
    ''pattern'': ''junit'',
    ''type'': ''substring'',
    ''scopes'': [ ''all'' ]
  } ],
  ''applies_to'': [ ''java'' ]
}
```

The rules are generated in JSON format and contain three attributes: `tags`, `patterns` and `applies_to`. The tags attribute is an array of strings representing keywords or categories which correspond to the library. The patterns attribute is an array of one or more pattern objects and contains attributes like: `pattern`, `type` and `scopes`. The pattern attribute is a string representing the pattern to match. It is usually a regular expression, but can also be a simple string. The format of the pattern is specified by the ''type''. A `type` can be

- `regex` - pattern is a regular expression
- `string` - searches for the raw text on a word boundary
- `substring` - searches the raw text but does not look for word boundaries

Also, `scopes` is a string array that specifies what part of the file to search in. The scopes are:

- `code` - searches only the sections that represent executable logic
- `comment` - searches in the non-executable documentation that is found in the file to explain the code
- `all` - searches everywhere in the file

Finally, `applies_to` is a string array containing what languages a rule should be run on. If the applies_to attribute is not defined or empty, then the rule is applied to every file type.

So, for the example above, the rule has as a pattern the name of the library, `junit`, it searches for this pattern in the entire code (comments as well) and only in Java files. If the pattern matches, then the file or files where it matched will be returned and the user will know where the library is used inside the system.

To see where exactly the library is used, adds a new dimension to the Software Composition Analysis, by combining the information about the library with the knowledge of being aware where the library is used in the system, making a transition from offering information ''just about the library'' to offering a more comprehensive analysis about ''my library'', about the concrete library used, to determine which actions or which future steps are best for the system, in the context of its libraries.

[2]https://github.com/Microsoft/ApplicationInspector

D. RISK ASPECTS

The meta-data collected from the various sources and stored in the model can be then used to identify a set of *risk aspects* related to an open-source component that must be assessed. To ensure an objective assessment for each, a *metric* must be defined to measure the level of risk; also, a classifier must be used to assign a metric value with a discrete level of risk, based on a set of user-defined risks.

In this paper, we decided to focus on the following three aspects:

1) **Lag**: a project using an outdated version of an open-source component. In order to measure this, we define the *Age of Used Version* metric as the number of months between the dates of the latest version and that of the version used in the analyzed project, whereby a larger value will indicate a case of a more severe case of *Lag*.
2) **Operational Risk**: a project depends on open-source component that is not anymore maintained by its developers. To quantify this aspect, we define the *Age of Latest Version* metrics as the number of months between the current date and the release date of the latest version; again, the higher the value for this metric, the higher the *Operational Risk* associated with that open-source component.
3) **Vulnerability**: a project is depending on an open-source component that has one ore more known security vulnerabilities. To quantify the severity we will rely on the *Severity Level* metric defined by *Sonatype's OSS Index*.

E. ILLUSTRATING THE APPROACH

After having described the major components of our approach we will illustrate in this section the way it can be used step by step to assess a system's open-source dependencies, based on a made-up example. However, to make the example close to reality, we imagined the case of a system that is using Java on the backend, managed using *Maven*, and JavaScript on the frontend, managed using *npm*.

The first step in the assessment is to extract the dependencies from the system as described in Section III-B1. *Inspector-Lib's* dependency scanner will search for the specific configuration files of each package manager. In this example, it will parse the `pom.xml` files, for the *Java/Maven*, and the `package.json`, `package-lock.json` files for the *JavaScript/npm* side. Each configuration file is parsed and dependencies are extracted. The result is a list of third-party open-source libraries that are used in the system. Let's assume that the dependency extraction step has found the five dependencies summarised in Table 3, specifically the library's name and

TABLE 3. Library meta-data for the running example.

Provider	Library	Used Version	Latest Version	Age of Used Version	Age of Latest Version	Vulnerabilities	License
maven	mysql:mysql-connector-java	8.0.12	8.0.30	48	2	1H, 4M	GPL 2.0
maven	junit:junit	4.7	4.13.2	138	18	1M	CPL 1.0
maven	com.fasterxml.jackson.core:jackson-databind	2.13.3	2.13.4	3	0	2H	Apache 2.0
npm	minimist	1.2.0	1.2.6	79	6	1M	MIT
npm	exit	0.1.2	0.1.2	0	105	-	MIT

its version number. This information is stored in the model that instantiates the meta-model. Note that the information is stored in a uniform manner although it comes from two different dependency managers: *Maven* and *npm*.

The second step of the assessment is to collect relevant information about each of these libraries, namely: the date of the library's *latest release*, the open-source license of that library, and the set of *known vulnerabilities* associated with the version of the library that our project relies on. This information is fetched by making requests to the various online databases that store and constantly update key facts about open-source dependencies, as shown in Figure 1. Again, the information is stored in our model in a manner that is independent of the specific database from where the information was fetched.

As a result of feeding the information into the model, and computing the *Age of Used Version* and *Age of Latest Version* metrics, as described in Section III-D, we end up with the results summarised in Table 3.

By analyzing the results, we discover several concerning facts: `junit 4.7` is outdated by more than 10 years and has a known vulnerability; `minimist 1.2.0` is also outdated by more than 6 years and has one vulnerability. Next, we discovered that `mysql-connector-java 8.0.12` is outdated by more than 4 years, and has five vulnerabilities, one of them being of high risk; moreover, the library is released under a copyleft license, which means that depending on how it is used in the analyzed project, the project's codebase might need to be released under a similar copyleft license. This is a concerning finding as we keep our project closed-source, meaning that we must either replace or remove the library.

Despite the valuable insights found about these libraries, we still don't know much about the effort that will be required to address these issues. The uncertainty has a clear cause: declaring an open-source dependency in a build configuration file does not mean that it is actually used, nor does it indicate how widespread that usage is. This is where the differentiation aspect of our approach comes into play: we use regular expressions to match each library to certain strings that are specific to that library (e.g. full-path package names) and then search for these library "fingerprints" across the project's codebase.

By doing this, in our example, we found that the `mysql-connector-java` library was declared in the *pom.xml* file, but it is currently not used anymore in the code. This means that despite all the concerning findings that we outlined earlier, there is no *real* risk for our system; and all we need to do is to remove the declaration. By contrast, we discovered that the `minimist` library is used widely across our codebase, which means that updating it is a high priority especially considering the vulnerabilities.

IV. EVALUATION OF THE APPROACH

The previous example has illustrated the need to know how an open-source dependency is used in a codebase, and the *Inspector-Lib* approach to address this need. However, going beyond a running example, the point that needs to be validated is this: does the case from our example occur in *real* projects? In other words, how frequent are the cases when open-source components with comparable issues (e.g. vulnerabilities, lack of support, copyleft licenses etc.) have a radically different impact in terms of the cost of remediation, based on how much the analyzed project actually depends on that component.

In this context, we will validate our approach by investigating the following research questions:

- **RQ1**: Does *Inspector-Lib* provide the right level of automation to handle a large variety of projects that are written in wide range of languages and rely on multiple dependency management systems?
- **RQ2**: Across real-life projects, how broad is the range of maintenance effort that is required to address open-source components that appear to pose a comparable level of risk to the project?

A. FEASIBILITY OF DATA EXTRACTION

The first step of the evaluation was focused on addressing RQ1, namely to asses if the *Inspector-Lib* tool-set is able to deal with a wide range of projects, written in multiple programming languages and having their open-source dependencies managed by multiple systems. To this end we selected 200 projects,[3] namely 50 projects for three languages and four major dependency management systems: Java (*Maven* and *Gradle*), JavaScript (*npm*) and Python. We selected only open-source projects for two reasons: to ensure that we have access to a large number of projects, and also make the results of this study easier to replicate. We selected the

[3]The projects can be found in following dataset [21].

projects from those that have the highest ratings on the GitHub platform, namely the projects with the largest number of stars, whereby a star is a sign of appreciation meaning that projects with many stars can be regarded as being very important to a wider community. Many of the selected projects belong the major organizations like Google, Spring, Apache, Facebook, Angular, VueJS, and Microsoft, but we have also included project from many other organizations and individual developers.

For the projects containing Java code, we extracted 19,064 dependencies from 50 *Maven* and 1,152 dependencies from the projects managed using Gradle. Due to the large number of dependencies in the *Maven* projects we analyzed in more detail 1,171 dependencies of five projects coming from Google (3) and Apache (2). We found 42 libraries having known security vulnerabilities, and 231 libraries for which an outdated version was used (high *Lag*); also, several used libraries have a high *Operational Risk* as they were not been active in more than five years. The 50 *Gradle* projects had 585 dependencies that were outdated; and we detected that these dependencies contained 77 vulnerabilities.

For the 50 JavaScript projects, we successfully extracted 46,762 dependencies from the *npm* configuration file. Due to the very large number of dependencies, similarly to the *Maven* projects, we chose 15 projects to analyze their dependencies thoroughly. We selected the projects so that they are widely spread across the most mature organizations (Facebook, Angular, Google, VueJS) to ensure that we potentially capture a diverse range of patterns in handling open-source dependencies. These 15 projects have a total of 2,482 dependencies. The number itself is surprisingly small: although we selected 30% of the initial 50 project, they only have 5% of all the dependencies. This may indicate that more mature companies tend to have as few dependencies as possible, although this is an assumption that would require more analysis before it can stated as a fact. When extracting the meta-data for these dependencies we discovered that almost half of them (1,147) are outdated. Also, 41 libraries have known vulnerabilities.

For the 50 Python projects, we extracted 721 dependencies. From these dependencies only 12 have vulnerabilities and 291 are outdated.

This part of the analysis has revealed that even in highly popular projects managed by mature companies the usage of open-source components poses significant challenges. Moreover, we noticed that the JavaScript dependencies tend to be more affected by software composition issues, while the Python projects tend to be affected less. For now this is just a first impression that requires further investigation before it can be validated.

As a conclusion, returning to the questions posed at the beginning of the subsection, with our approach we were able to extract dependencies from different projects and get information about them, as well as computing their age, but also to detect the usage of the libraries in the code with the generated rules leading to an enhanced way of looking at the dependencies of a system. The last question will be answered in the next subsection.

B. RELEVANCE OF LIBRARY'S CODE "FOOTPRINT"

The second part of the evaluation is focused on answering the second research question, namely the extent to which across real projects the risk and maintenance effort associated with an open-source component varies significantly depending on how much the assessed project depends on that component. The evaluation was performed based on 28 open-source projects from GitHub, selected from the larger set of projects analyzed during the first evaluation stage. For this second stage, we selected a set of the ''highest rated'' projects, based one their number of stars/up-votes on GitHub, coming from three organizations: *Facebook*, *Google* and *VueJS*. We selected 3 *Maven*, 3 *Gradle*, 4 *Python* and 18 *npm* projects. We based our decision to select a higher number of *npm* projects on the observation made during the first evaluation stage that npm projects tend to have more dependencies compared to other types of projects.

The evaluation is based on the following methodology: we detected and classified automatically the cases of risks associated with an open-source component itself across the 28 project, as presented in Section III-D. For the sake of uniformity we decided to use a four-level classifier on all risk aspects, based on the following set of three thresholds:

1) **Lag**: risk is considered *high* when *Age of Used Version* is at least 5 years (60 months), else it is *medium* if the value is at least 2 years, and otherwise it is *low* if the value if at least 12 months.
2) **Operational Risk**: risk is considered *high* when *Age of Latest Version* is at least 3 years (36 months), else it is *medium* if the value is at least 2 years, and otherwise it is *low* if the value is at least 12 months.
3) **Vulnerability**: a *high* risk corresponds to a *Severity Level* of at least 7, else the risk is *medium* if the *Severity Level* is at least 4, and, otherwise, it is *low* if the metric has a non-zero value.

Then, we broke down those cases along a second, orthogonal dimension, namely how much each of the analyzed projects depends on that open-source component with some intrinsic risks. To quantify this, we simply counted the files that directly depend on an open-source component, based on the results retrieved by the *Open-Source Code Scanner* described in Section III-C; and we used the following thresholds to classify the values: a dependency on an open-source component is considered *widespread* if at least 100 files depend on it, else the dependency is considered *scattered* if at least 40 files depend on that OS component; if not, the dependency is considered *ghost* if no files depend on the open-source component, or else it is classified as *sparse*.

Findings numbers in cells count the total numbers libs in a category across all the 28 analyzed projects		Code Dependency extent to which the code relies on the library			
		widespread >= 100 files	scattered >= 40 files	sparse >= 10 files	ghost only declared
Lag used version is sensibly behind a library's latest version	high >= 5 years	5	5	28	1
	medium >= 2 years	19	27	74	36
	low >= 1 year	24	21	34	33
Operational Risk latest version is very old	high >= 3 years	13	9	18	30
	medium >= 2 years	1			9
	low >= 1 year	21	9	20	20
Vulnerability used version has a known security vulnerability of a certain severity level	high >= 7.0		1	2	1
	medium >= 4.0	5		5	1
	low >= 0.1				

FIGURE 3. Library concerns and their impact in the code.

1) DATA INTERPRETATION

The results are summarised in the matrix depicted in Figure 3.

For the cases of *Lag* we notice that while most cases are *sparsely* used in the analyzed projects, there are nevertheless 24 cases of OS components (5 *high Lag* and 19 *medium Lag*) that have a *widespread* presence in projects. Additionally, there are 32 cases of OS components that are used in over 40 files per project. The distribution of severe *Lag* cases across the entire spectrum of project dependency indicates that the level of maintenance effort required to address high risk issues in open-source components can differ based on the extent to which a project depends on that component.

While there are fewer severe cases of OS components with *Operational Risk*, the distribution is similar to the *Lag* risk: there are many cases where OS components that are inactive for many years have a wide "footprint" in a project's codebase. This means that some projects heavily rely on potentially unsupported libraries, which pose significant operational risks. At the same time it is worth noting that a higher number of severe cases (i.e., *high* and *medium*) are in *ghost* dependencies, namely the library is declared in the project's configuration file but it is not used anymore in the code. This is to a large extent a good sign suggesting that in these projects developers were active in removing the dependencies on these unsupported components and just forgot to remove the declared dependency.

In terms of vulnerabilities, the overall number of cases is significantly lower compared to the previous two types of risks, which is expected given the high popularity of these projects, and the very solid reputation of the three organizations that have created them. Nevertheless, the number of severe cases (i.e. *high* and *medium*) is almost evenly distributed between the projects where the component was used directly in at least 40 files (*widespread* and *scattered*) compared to the projects where components with severe vulnerabilities are only *sparsely* used in the code. This demonstrates that also for the case of vulnerability risks, estimating the actual risk and the effort required to address the risk varies widely based on how deeply embedded in the code the dependency is.

In conclusion, the data tends to validate the assumption that measuring how much a codebase depends on an OS component offers valuable insights regarding the actions that must be taken, the priority in which open-source related issues must be addressed, and the level of effort required to remediate the identified concerns; for instance, when an OS component with *high Lag* has a *widespread* code dependency, we will know that a major part of the system relies on outdated libraries; and therefore a comprehensive update strategy targeting these widely used libraries becomes imperative to ensure overall system integrity.

2) MANUAL INSPECTION OF FINDINGS

In addition to the high-level, quantitative analysis of the data, we manually inspected a large number of cases, starting from the most relevant ones, namely the cases of OS components that have one or more severe risks and have a wide dependency in certain projects. For projects where such strong cases exist, we also searched complementary cases, namely components of relatively similar risk, which are less widely used in the same project; or cases where the same open-source component is less widely used in other projects. We discuss next the most interesting discoveries.

1. *vue-cli* (VueJS): The project has a *widespread* dependency on library `ejs 3.1.6`[4] and has one high vulnerability (score 9.8). This means that although the low *Lag* would suggest that an upgrade to the latest version should be easy and would remove the vulnerability, due to its scattered usage, updating the library may impact the system's functionality as it may have been relied upon in specific ways or for specific functionalities.

2. *react-native* (Facebook): the `typescript 4.1.3`[4] component exhibits one high vulnerability, but the project only uses the component *sparsely*. By contrast, in another project (*jest*) from the same organization (Facebook), there is a widespread dependency on `typescript 4.7.3` but the version used here has no vulnerabilities. This suggests that organizations tend to pay more attention in upgrading their OS dependencies when the project has a wider dependency on a library. We also noted that in the *jest* project, multiple versions of `typescript` are used in parallel in different parts of the system, likely due to the modular nature of the components.

3. *react-native* versus *flux* (Facebook): the library `react 15.4.1`[4] has a high *Lag* and has a *widespread* usage, which means that upgrading it to the latest version may introduce functional changes in the system or impact the components where it is utilized. By contrast, in *flux* an even older version of the same library is used (`15.0.2`) but this time the library is only *sparsely* used; therefore, an upgrade is expected to have less impact compared to the system where *react* is heavily used.

4. *react* (Facebook): The `electron 11.1.0`[4] has three high and three medium vulnerabilities. It is used in a limited number of files, indicating its relatively low importance in the system. Considering the number and severity of vulnerabilities, it prompts an evaluation of the necessity of this dependency in the system.

5. *guice* (Google) vs. *stetho* (Facebook): The library `junit 4.11`[4] is *widespread* throughout the *guice* system, while in *stetho* `junit 4.12` is only *sparsely* used. Although both versions of `junit` have a medium vulnerability, the effort of upgrading to the latest version where the vulnerabilities are solved is much smaller in the case of *stetho*.

6. *guice* (Google): The `servlet-api 2.5`[4] library is *scattered* throughout the system, impacting more than 40 files, it has a *high Operational Risk* and is licensed under a GPL 2.0 license, which is copyleft. While this may not pose a problem for an open-source system like this, it could be problematic for commercial or private systems, as the copyleft license would require the system's parts utilizing the library to be made public.

7. *ExoPlayer* (Google): There are two **high** severity vulnerabilities in `protobuf-javalite 3.19.1`[4]; however, at a close inspection we noted that it is declared as a dependency without being used at all in the code (*ghost* dependency). This raises questions about why the library is declared but not utilized, especially considering the presence of vulnerabilities. It suggests either a potential oversight or the library's previous usage that was not removed from the configuration files.

8. *Druid* (Apache): The project uses two libraries containing vulnerabilities: `jetty-server`[4] and `netty`[4], but analyzing the spread of their usage in the code reveals differences: `jetty-server` was used in 34 files, out of which 11 contain test code; and `io.netty:netty` is accessed from 16 files, two of which contains test code. This finding shows the importance of taking into account also the nature of the files where these dependencies occur (e.g. test vs. functional).

In conclusion, these examples highlight once more the importance of understanding the usage and distribution of libraries within a codebase. Knowing exactly where the libraries are being used and the extent of their impact across different components of the system, leads to a better understanding of the system and its possible flaws. Therefore, it plays a crucial role in decision-making processes related to updating, removing, or changing libraries.

C. LIMITATIONS AND THREATS TO VALIDITY

While our approach provides valuable insights into the libraries used in open-source projects, it is important to acknowledge several limitations and potential threats to the validity of our analysis.

1) EXTERNAL VALIDITY

In this category we group the threats to validity that deal with the generality of our conclusions. First of all, the study may be affected by a certain selection bias, as we focused on projects that come from mature organizations and are very much appreciated by the open-source community, which means that we can expect these projects to exhibit a level of library hygiene above average, as open-source projects that are plagued with issues and lack quality are unlikely to gain traction and widespread adoption. Thus, these projects might not be representative of the entire open-source ecosystem and may have different characteristics compared to projects from other organizations or domains. Moreover, in our experience, in commercial (''closed-source'') projects the focus is less on the code itself and more on the functionality of the system. Consequently, both in commercial project and in less popular open-source projects we expect to see a lower level of library hygiene. This means that, while results are expected to differ from those presented in this paper, the relevance of the approach presented here is in fact even higher in these other projects.

[4]https://www.npmjs.com/package/ejs; https://www.npmjs.com/package/typescript; https://www.npmjs.com/package/react; https://www.npmjs.com/package/electron; https://mvnrepository.com/artifact/junit/junit; https://mvnrepository.com/artifact/javax.servlet/servlet-api; https://mvnrepository.com/artifact/com.google.protobuf/protobuf-javalite; https://mvnrepository.com/artifact/org.eclipse.jetty/jetty-server; https://mvnrepository.com/artifact/io.netty/netty

2) TEMPORAL VALIDITY

The analysis relies on the data available up until the knowledge cutoff date, which in our case was January 26, 2023. Given that the software landscape is constantly evolving, new libraries, versions, vulnerabilities, and best practices might have emerged since then. This time gap could impact the relevance and accuracy of the findings when applied to the current state of software development.

3) CONSTRUCT VALIDITY

Our approach involves certain assumptions and simplifications to make the analysis feasible. For example, we assume that the highest-rated projects on GitHub are representative of well-maintained and widely-used projects. However, this assumption might not always hold true, and there could be other factors influencing a project's popularity and quality. These assumptions and simplifications should be taken into account when interpreting the results.

4) INTERNAL VALIDITY

For the extraction of dependencies and gathering information about libraries (licenses, vulnerabilities, release dates), a manual verification process was conducted by examining the configuration files and responses from requests. Although efforts were made to ensure accuracy, manual verification introduces the possibility of human error.

Considering these limitations and threats to validity is crucial for understanding the scope and potential implications of our analysis. They highlight areas that may require further research and caution in applying the findings to other scenarios.

V. CONCLUSION AND FUTURE WORK

The entire system developed offers a better perspective when it comes to analyze a project's dependencies. It offers a view on the age of the dependencies, on the vulnerabilities a dependency has, and also can help to understand the structure of a system by simply seeing its dependencies and where they are used.

Staying up-to-date is the most secure strategy and making incremental changes beats having big-bang updates. That is why analyzing projects is important, to always check the age and see since when a dependency wasn't updated.

Vulnerabilities can be very dangerous and using third-party libraries increases the chances of having them in a system. Having a look at the dependencies at any time, to verify the existence of vulnerabilities, is very important for developing it into a good and stable software system and maintaining it in the future.

If a library has vulnerabilities, it doesn't mean that the entire functionality is vulnerable. It is possible that only a method or a small part of it is vulnerable and the rest of it works without putting the system in danger. Further research in this direction is needed to tell exactly, if, despite having a vulnerable library, the system really has flaws by using the vulnerable part of it or not.

One priority, having the extensible model, would be to analyze more dependency types, offering the software community the possibility of analyzing multiple systems which contain different library dependencies.

When analyzing systems, its history may offer even more information about the libraries and their development throughout the system. Therefore, to offer a more complete Software Composition Analysis, we will try to enhance our approach by looking at the history and the development of the libraries.

In our future work, we plan to improve the dispersion metric used in our approach. Instead of relying on absolute thresholds, we will explore the possibility of implementing relative thresholds. This adjustment will allow for a more adaptable and dynamic assessment of library dispersion within the code, taking into account the specific characteristics of each project.

Currently, our approach identifies whether a library is used, but further improvement can be made to specify exactly which functionalities or components of the library are utilized in the code. This would provide a more detailed and fine-grained understanding of how the library contributes to the overall system.

We will explore the aggregation of risks associated with each library. By combining information on lag, operational risk, and vulnerabilities, we can derive an overall risk score for each library. Libraries that exhibit high or medium risks across all three categories will be identified as ''super-risks.'' This prioritization strategy will help focus efforts on components that require immediate attention and remediation.

Future work should also consider the inclusion of embedded libraries in the analysis process. Currently, our approach focuses on analyzing third-party libraries obtained from external repositories. However, many software systems incorporate embedded libraries, which are libraries included within the system's codebase. By incorporating the analysis of embedded libraries, our approach would provide a more comprehensive view of a system's dependencies, covering both externally sourced libraries and those embedded within the codebase. This expanded scope would enable better decision-making and provide a more accurate assessment of the system's overall health.

Also, while using the approach other questions have been stated: Is the vulnerable code of the library actually used?; What is the gap between declared dependencies and the actual usages?; When did the vulnerable library appear in the system, was it used from the beginning?; How have the libraries evolved during the development of the system?, which we would like to answer after further research. Moreover, knowing where each library is used can show information about the architectural perspective, seeing which libraries are used together or if multiple libraries providing the same functionality are used in the system. This is another aspect worth exploring.

In conclusion, our approach, which emphasizes the importance of understanding the usage and distribution of libraries

within a system, provides a better perspective for prioritizing services and decision-making processes. By gaining insights into where libraries are utilized and the extent of their spread throughout the system, we offer valuable information regarding the potential challenges associated with making changes or updates related to these libraries. Furthermore, by considering all available information about various libraries, including their vulnerabilities and impact on the system, we can effectively prioritize actions to be taken. For instance, when faced with multiple libraries possessing different vulnerabilities, we can determine which ones to address first based on their prevalence across the system, the number of vulnerabilities they possess, and the criticality of these vulnerabilities. This approach enables to allocate resources and efforts more efficiently, ensuring that the most pressing library-related issues are tackled promptly. Therefore, our approach can make an important contribution in the process of evaluating the quality of software systems.

REFERENCES

[1] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm Javascript packages," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Madrid, Spain, Sep. 2018, pp. 559–563, doi: 10.1109/ICSME.2018.00067.

[2] H. Plate, S. Elisa Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Bremen, Germany, Oct. 2015, pp. 411–420, doi: 10.1109/ICSM.2015.7332492.

[3] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proc. 12th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2018, pp. 1–10.

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories (MSR)*, Gothenburg, Sweden, May 2018, pp. 181–191.

[5] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proc. 44th Int. Conf. Softw. Eng., Softw. Eng. Pract.*, Pittsburgh, PA, USA, May 2022, pp. 331–340, doi: 10.1145/3510457.3513044.

[6] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of attacks on open-source software supply chains," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2023, pp. 1509–1526, doi: 10.1109/SP46215.2023.10179304.

[7] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Pittsburgh, PA, USA, May 2022, pp. 672–684, doi: 10.1145/3510003.3510142.

[8] M. Zimmermann, C. A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. 28th USENIX Secur. Symp.*, pp. 995–1010. 2019.

[9] Y. Yano, R. Gaikovina Kula, T. Ishio, and K. Inoue, "VerXCombo: An interactive data visualization of popular library version combinations," in *Proc. IEEE 23rd Int. Conf. Program Comprehension*, Florence, Italy, May 2015, pp. 291–294, doi: 10.1109/ICPC.2015.43.

[10] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Softw. Eng.*, vol. 24, pp. 381–416, Feb. 2019.

[11] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," in *Proc. 2nd IEEE Work. Conf. Softw. Visualizat.*, Victoria, BC, Canada, Sep. 2014, pp. 127–136, doi: 10.1109/VISSOFT.2014.29.

[12] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Campobasso, Italy, Mar. 2018, pp. 288–299, doi: 10.1109/SANER.2018.8330217.

[13] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration," *Empirical Softw. Eng.*, vol. 23, pp. 384–417, Feb. 2018.

[14] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reengineering (SANER)*, Montreal, QC, Canada, Mar. 2015, pp. 520–524, doi: 10.1109/SANER.2015.7081869.

[15] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, May 2015, pp. 109–118, doi: 10.1109/ICSE.2015.140.

[16] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in Java projects," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Adelaide, SA, Australia, Oct. 2020, pp. 35–45, doi: 10.1109/ICSME46990.2020.00014.

[17] *Black Duck, Synopsys*. Accessed: Jan. 8, 2023. [Online]. Available: https://www.blackducksoftware.com

[18] *Snyk Limited*. Accessed: Jan. 8, 2023. [Online]. Available: https://snyk.io

[19] *Dependabot, GitHub*. Accessed: Jan. 10, 2023. [Online]. Available: https://dependabot.com

[20] *WhiteSource Software*. Accessed: Jan. 11, 2023. [Online]. Available: https://www.whitesourcesoftware.com

[21] A. Molin, A. M. Rivis, and R. Marinescu, "Analyzed open-source projects," Fac. Automat. Comput., Politehnica Univ. Timişoara, Timişoara, Romania, May 2023, doi: 10.21227/6bwq-mk13.

ANDY MOLIN was born in Timişoara, Romania, in 1996. He received the bachelor's degree in computer science and the master's degree in software engineering from the Politehnica University of Timişoara, Romania, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree in software engineering (computer and information technology).

Since 2019, he has been a Teaching Assistant with the Politehnica University of Timişoara, teaching laboratories about software engineering, object-oriented programming, artificial intelligence fundamentals, modeling and simulation, design patterns, and testing.

ANDREI MARIO RIVIŞ was born in Timişoara, Romania, in 1994. He received the bachelor's degree in computer science and the master's degree in software engineering from the Politehnica University of Timişoara, Romania, in 2017 and 2019, respectively, where he is currently pursuing the Ph.D. degree in software engineering.

Since 2017, he has been a Teaching Assistant with the Politehnica University of Timişoara, teaching laboratories about software engineering, design patterns, and testing. During the time working with students, he also developed several open-source software analysis tools and analyzing both code and meta-data about the software development lifecycle. Since 2020, he has done several software assessments on large scale software systems.

RADU MARINESCU received the Ph.D. and Habilitation degrees from the Politehnica University of Timişoara, Romania, in 2002 and 2012, respectively. He is currently a Professor of software with the Politehnica University of Timişoara. He has more than 20 years of experience in software analysis as a Consultant, a Researcher, a Professor, and an Entrepreneur. He conducted highly influential research, built award-winning tools, and provided extensive consultancy on quality assessment and large-scale code transformation to major service and product companies in several sectors, from finance to telecom. He is the coauthor of a seminal book on software metrics, titled *Object-Oriented Metrics in Practice* (Springer, 2006).