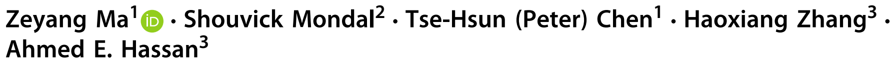
Empirical Software Engineering (2024) 29:83
https://doi.org/10.1007/s10664-024-10448-6

VulNet: Towards improving vulnerability management in the Maven ecosystem

Zeyang Ma[1] · Shouvick Mondal[2] · Tse-Hsun (Peter) Chen[1] · Haoxiang Zhang[3] · Ahmed E. Hassan[3]



Accepted: 16 January 2024 / Published online: 5 May 2024
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract
Developers rely on software ecosystems such as Maven to manage and reuse external libraries (i.e., dependencies). Due to the complexity of the used dependencies, developers may face challenges in choosing which library to use and whether they should upgrade or downgrade a library. One important factor that affects this decision is the number of potential vulnerabilities in a library and its dependencies. Therefore, state-of-the-art platforms such as Maven Repository (MVN) and Open Source Insights (OSI) help developers in making such a decision by presenting vulnerability information associated with every dependency. In this paper, we first conduct an empirical study to understand how the two platforms, MVN and OSI, present and categorize vulnerability information. We found that these two platforms may either overestimate or underestimate the number of associated vulnerabilities in a dependency, and they lack prioritization mechanisms on which dependencies are more likely to cause an issue. Hence, we propose a tool named VulNet to address the limitations we found in MVN and OSI. Through an evaluation of 19,886 versions of the top 200 popular libraries, we find VulNet includes 90.5% and 65.8% of the dependencies that were omitted by MVN and OSI, respectively. VulNet also helps reduce 27% of potentially unreachable or less impactful vulnerabilities listed by OSI in test dependencies. Finally, our user study with 24 participants gave VulNet an average rating of 4.5/5 in presenting and prioritizing vulnerable dependencies, compared to 2.83 (MVN) and 3.14 (OSI).

Keywords Software vulnerability management · Software ecosystems · Empirical software engineering

1 Introduction

Modern software often reuses functionality from other software libraries (i.e., dependencies) to provide a full working system. Such reuses significantly increase developers' productivity

---

Communicated by: Xin Peng

✉ Zeyang Ma
m_zeyang@encs.concordia.ca

Extended author information available on the last page of the article

and improve software quality (Frakes and Kang 2005; Ruiz et al. 2012; Mojica et al. 2014; Epperson et al. 2022). To facilitate software reuse, different communities have proposed various software ecosystems. For Java projects, the Maven ecosystem plays an important role of a software supply chain. For projects built with `mvn` (the Maven build system), developers declare the required external libraries (i.e., dependencies) in the configuration file, namely `pom.xml`. Maven then automatically downloads all the dependencies from Maven Central.

Although Maven provides great convenience to developers, there are also potential risks related to the quality of the used dependencies. In particular, if a library contains vulnerabilities (e.g., security weaknesses), all the dependent software may also become vulnerable (Kula et al. 2018). According to Synopsis (Synopsys 2022; Lemos 2022), a software system depends on an average of more than 500 open source libraries and components. Hence, it is difficult to know if a used dependency has vulnerabilities, especially when these dependencies in turn also depend on other dependencies. For example, in 2021, Apache Foundation revealed a vulnerability in remote code execution (CVE-2021-44228) (Mitre 2022) in `Log4j` (Apache 2022a).

Since `Log4j` is a commonly-used logging library, the vulnerability has a potential impact on hundreds or even thousands of libraries that use `Log4j` directly or indirectly. According to the Google security blog (Google 2022a), they found that 35,863 of the available Java libraries from Maven Central depend on the vulnerable `Log4j`. This means that more than 8% of all packages on Maven Central have at least one version that is impacted by this vulnerability. Among the 35,863 libraries, about 7,000 libraries have `Log4j` as a direct dependency. A larger portion (80%) of the remaining libraries have `Log4j` as a transitive (also called indirect) dependency, which means that developers need to carefully examine the complex dependency graph to know whether their software is impacted by the vulnerability. Due to the notorious vulnerability in `Log4j` and other popular open source libraries, developers are now more aware of and, at the same time, concerned about vulnerabilities in dependencies. To help developers understand and manage vulnerabilities in dependencies, there are existing platforms that provide library information for developers, such as the Maven Repository (MVN) (Repository 2022a) and Google's Open Source Insights (OSI) (Google 2022b). MVN indexes the central repository for the Maven ecosystem, which contains a comprehensive collection of software libraries. MVN also shows the dependency vulnerability data to provide warnings to users who depend on Maven Central. In addition, OSI is a service developed and hosted by Google to help developers better understand the structure, construction, and security of open source software libraries. *So, when developers are in need of deciding whether to use a specific library or whether to upgrade to a specific version, platforms like MVN and OSI can provide some insight and suggestions* (Google 2022c). Given a library, MVN and OSI show the potential vulnerabilities (collected from CVE) that come from the library itself or its dependencies to help developers make their choices and minimize potential vulnerabilities in the used dependencies.

Prior studies (Shen et al. 2011; Johnson et al. 2013; Chen et al. 2016; Barik 2016; Lipp et al. 2022; Nachtigall et al. 2022) found that providing vulnerability information in an effective and prioritized fashion can significantly improve the usefulness of vulnerability reporting tools. Hence, in this paper, we first investigate how the two platforms, MVN and OSI, present and categorize vulnerability information in dependencies. We study how the two platforms present and categorize the vulnerabilities by comparing the top 200 most popular libraries on both platforms. We found that MVN may underestimate the number of dependencies and their vulnerabilities because they only list direct dependencies and their vulnerabilities while transitive dependencies are missing. On the other hand, OSI may have overestimated the number of dependencies and their vulnerabilities. Although OSI does not only calculate

direct dependencies, but also transitive dependencies and inheritance dependencies, the mix of dependencies and putting them in one list may include some unreachable dependencies and their vulnerabilities. We also found that both platforms lack proper mechanisms that would help developers prioritize the vulnerability investigation and resolution effort in the software supply chain.

To address the limitations, we designed a vulnerability management approach VulNet, to help developers address the above-mentioned limitations. VulNet categorizes the dependencies based on their types, provides a risk level for different dependencies, and ranks the dependencies based on the vulnerability severity. Finally, we compare VulNet with MVN and OSI on how the three platforms organize and present the dependencies and the associated vulnerabilities. Through a user study with 24 practitioners, we find that VulNet receives a much higher rating compared to MVN and OSI on dependency vulnerability management.

The main contributions in this paper are as follows:

- We conduct an empirical study on two state-of-the-art vulnerability management platforms (i.e., MVN and OSI) to investigate how they furnish information on vulnerable dependencies. We identify that MVN underestimates the number of dependencies and vulnerabilities, resulting in missing information on some dependencies and vulnerabilities. OSI overestimates the number of dependencies and vulnerabilities, and lists some vulnerabilities with a lower impact level. Additionally, there is room for improvement in their classification and prioritization of results.
- We propose a tool, VulNet, which assists developers in navigating and prioritizing vulnerable dependencies in the Maven ecosystem.
- Through an evaluation of 19,886 versions of the top 200 popular libraries, we find that VulNet includes 90.5% and 65.8% of the dependencies and associated vulnerabilities ignored by MVN and OSI, respectively. VulNet also help reduces 27% of potentially unreachable vulnerabilities listed by OSI.
- In our user study with 24 practitioners, we find that VulNet received an overall rating of 4.5 out of 5 compared to MVN (2.83) and OSI (3.14). Practitioners also give VulNet's vulnerability prioritization mechanisms an average of 4.58 and gave positive feedback on the usefulness of VulNet.
- We made our experimental data publicly available (Ma et al. 2022).

In short, our study uncovers the limitations of the current Maven vulnerability management platforms and discusses the solutions that we designed in VulNet. Our study takes the first step to assisting practitioners with vulnerable dependency management. Our study also sheds light on future research directions in improving software ecosystems.

Paper Organization Section 2 discusses the background of the Maven ecosystem and vulnerability management, and related work. Section 3 presents the limitations and differences in state-of-the-art vulnerability management platforms. Section 4 presents the design of VulNet. Section 5 evaluates VulNet. Section 6 discusses the implications of our study. Section 7 discusses threats to validity. Finally, Section 8 concludes the paper.

2 Background and Related Work

In this section, we present the relevant background for vulnerability management in the Maven ecosystem and related work.

2.1 The Maven Ecosystem and Vulnerability Management

In Java, the Maven build system is primarily used to help developers compile, test, and build software. Developers can easily reuse third-party libraries by declaring the dependencies (i.e., external libraries) in Maven's configuration file (i.e., `pom.xml`). Maven would automatically download the dependencies and all of their upstream dependencies to complete compiling the project.

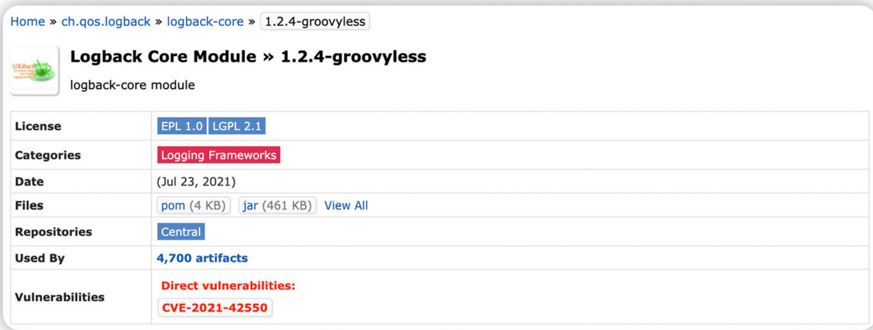
This convenience also comes with some potential risks. Because of the complexity of dependency graphs, the network of upstream dependencies is sometimes complex and difficult to manage (Imtiaz et al. 2021). When some of these upstream dependencies have vulnerabilities, the vulnerabilities may affect other downstream dependencies. To assist developers with understanding and managing vulnerabilities in library dependencies, the library information listing platform for Maven (i.e., MVN), which contains libraries from multiple repositories, now provides lists of possible vulnerabilities that are associated with a library. Similarly, Google created OSI, which provides library vulnerability information for Maven, NPM, PyPi, and some other ecosystems.

Figure 1 shows the vulnerability information for the Maven library `logback-core:1.2.4-groovyless` on the two vulnerability management platforms. The two platforms both list the vulnerabilities associated with the library, where MVN lists the associated CVE ID (e.g., CVE-2021-42550), and OSI lists the alias ID for the CVE (e.g., `GHSA-668q-qrv7-99fm`) and the title of the advisory report. Both platforms divide vulnerabilities into two parts: from the library itself and from dependencies. Showing the vulnerability information helps developers manage the vulnerabilities in the used dependencies. Developers can then decide whether to use a certain dependency or whether to upgrade/downgrade to a certain version (e.g., with less vulnerabilities).

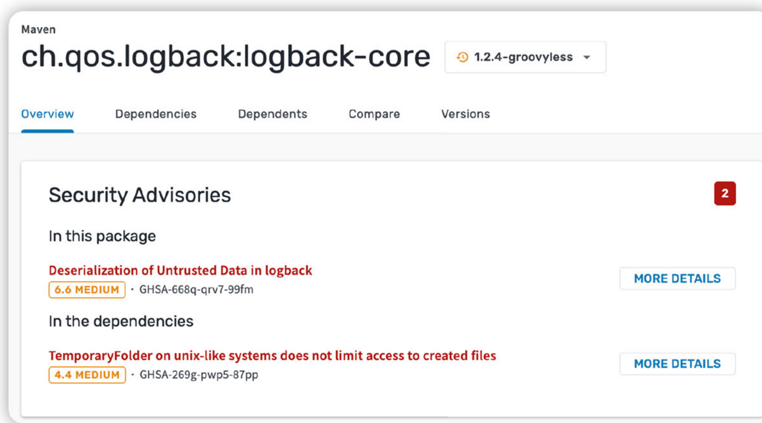
Even though MVN and OSI attempt to provide vulnerability information for the libraries and their dependencies, the complex nature of software dependency makes vulnerability management even more complicated. The Maven build system (`mvn`) supports a wide range of dependency configurations, which would affect how a vulnerability may propagate through the dependency graph. Below, we summarize the types and scope of dependencies in the Maven build system.

Direct and Transitive Dependencies Software upstream dependencies (i.e., external libraries) can be classified into two categories: *direct* and *transitive*. For a library, a dependency (of depth one) that is defined directly in its configuration file (`pom.xml`) is a direct dependency. The other non-immediate upstream dependencies (of depth more than one) introduced by direct dependencies are transitive dependencies. For example, consider the dependency: `LibA` → `LibB` → `LibC`.`LibC` is a transitive dependency of `LibA`, and `LibB` is a direct dependency of `LibA`. Vulnerabilities in both direct and transitive may have an impact on a library (Düsing and Hermann 2022), and due to the scale and complexity of such a dependency graph, a library may have tens or even hundreds of dependencies (Louridas et al. 2008; Harrand et al. 2022). Therefore, it is important to provide developers with information on how a vulnerability propagates through the dependency graph.

Dependency Scope Depending on how a dependency is used, its scope can be further divided into four types (Apache 2022b): *compile*, *runtime*, *provided*, and *test*. A compile dependency means that the dependent library is required during compilation and is available in the classpath of the library. A *runtime* dependency indicates that the library would be used during the execution. For instance, a JDBC driver (Oracle 2022) could be a *runtime*



(a) Maven repository.



(b) Google Open Source Insights.

Fig. 1 An example of the listed CVEs for the library `ch.qos.logback: logback-core: 1.2.4-groovyless` on MVN and OSI

dependency; while it is necessary at runtime for making database connections, the code can be compiled without the JDBC driver (it is an external library that is needed at runtime). A *provided* dependency is similar to a *compile* dependency but is not explicitly included in the classpath. Rather, developers need to include and enable *provided* dependencies in the run-time environment (e.g., so developers have the flexibility in choosing the versions of the library that they want to use). A *test* dependency is used only for compiling and executing the test cases. Besides the aforementioned types, a developer may mark a dependency as optional (excluded by default) by specifying the `<optional>` tag in the `pom.xml` file associated with the code. Such optional dependencies may only be needed if developers are required to use certain features provided by the library. For example, given the dependency `LibA` → `LibB` → `LibC` `<optional>`, `LibA` may still configure its `pom.xml` to include LibC if `LibA` requires the features related to the dependency. Additionally, the dependency scope might be changed during the transit in the dependency chain. For example, consider the dependency: `LibA` → `LibB` <*test*> → `LibC` <*compile*>. In this case, `LibC` should be considered a test dependency for `LibA`, because `LibC` is only used by `LibA`'s test

dependency: LibB. Since test dependencies are excluded from the exported binaries by default, vulnerabilities in test dependencies are less likely to cause issues.

Dependency Management Maven introduces dependency management to help refactor complex dependencies. Specifically, developers can define parent and child libraries in a Maven project. The child libraries can inherit the dependencies from the parent library for more organized dependency management.

2.2 Challenges in Presenting Vulnerability Information

Different dependency types and scopes determine how a vulnerability may propagate through the dependency graph and affect the overall project. For instance, the testScratch project in Fig. 2 is affected, when a vulnerability is discovered in the transitive dependency xmlParserAPIs. This vulnerability propagates to the upper-level junit-addons, then propagates to dbunit, and finally to testScratch. Hence, this propagation path potentially makes the project vulnerable to security flaws. However, if we consider the dependency scope, this effect may sometimes be spurious. For example, if we find a vulnerability in dbunit's downstream dependency junit, and junit is declared as a test dependency of dbunit, the vulnerability does not affect the entire project during the compilation scope as test dependencies (by default) are not included in the final binary file.

To the best of our knowledge, there has been no systematic study on how these vulnerability management platforms categorize and present library vulnerability information. Due to the complex nature of vulnerability management in a library's dependencies, how to present and prioritize vulnerabilities have a direct impact on helping developers make decisions on choosing libraries and selecting appropriate versions. As found in prior studies (Shen et al. 2011; Johnson et al. 2013; Chen et al. 2016; Barik 2016; Lipp et al. 2022; Nachtigall et al. 2022), the adoption of static security testing tools is highly related to how the results are presented to developers. Similarly, we believe that a systematic and better organization of the vulnerability information helps developers with dependency vulnerability management. Hence, our goal is to study the current vulnerability management systems, understand their limitations, and propose potentially better vulnerability management for the Maven ecosystem.

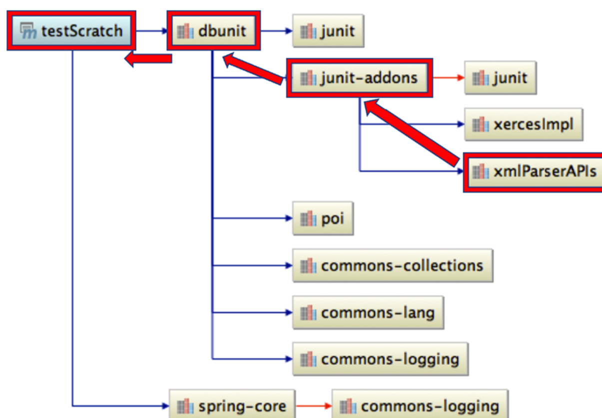


Fig. 2 How a vulnerability propagates in a dependency chain (of depth three) for the testScratch project

2.3 Related Work

We present related work in two directions: vulnerability management in open source ecosystems and reporting vulnerability information to users.

Vulnerability Management in Open Source Ecosystems Vulnerable dependencies are a major problem in today's open source software ecosystems (GitHub 2020; Latendresse et al. 2022; Decan et al. 2018; Croft et al. 2021; Pashchenko et al. 2018; Zerouali et al. 2022; Alfadel et al. 2021, 2020). GitHub reveals that active repositories with a supported package ecosystem have a 59% chance of getting a security alert in the next 12 months (GitHub 2020). A prior study by Prana et al. (2021) on 150 (out of 462,182) Java projects reports that the mean percentage of dependencies with vulnerability is 12.3%. Gkortzis et al. (2021) empirically investigated 1,244 open source Java projects to explore and discuss the distribution of security vulnerabilities. The results show that larger projects are associated with an increase in the number of potential vulnerabilities in both native and reused code. Massacci and Pashchenko (2021) show that libraries with high leverage (with more reused code) have a 1.6× higher probability of being vulnerable in comparison to the libraries with lower leverage. Latendresse et al. (2022) studied 100 JavaScript projects using NPM and indicate that less than 1% of the installed dependencies are released to production. However, our work focuses exclusively on Java projects in the Maven ecosystem. We compare two state-of-the-art vulnerability management platforms in this ecosystem, identify the limitations in organizing vulnerability information, and attempt to guide developers' mitigation efforts by enriching vulnerability information using a wider categorization of dependencies and prioritization.

Reporting Vulnerability Information to Users Vulnerability reports help developers navigate potential vulnerabilities and take needed corrective actions. However, unorganized reports may cause additional investigation overhead or even distrust in the result. Some previous studies attempted to improve the reporting of vulnerability detection tools and provide more effective information (Johnson et al. 2013; Imtiaz et al. 2021; Farris et al. 2018; Smith et al. 2015). Imtiaz et al. (2021) performed a comparative study of nine static security testing tools and reported that there are significant differences in the analysis report they generate. They recommend the usage of multiple tools to minimize false positives and vulnerability misses. Farris et al. (2018) provided a vulnerability prioritization tool VULCON, which uses a mixed-integer multi-objective optimization algorithm to prioritize vulnerabilities. Johnson et al. (2013) conducted interviews with 20 developers and investigated why developers are not widely using vulnerability detection tools and possible improvements. One of the reasons they found was that the displayed warnings were not informative enough. Smith et al. (2015) conducted a user study on how users were satisfied with the report of vulnerability detection tools and found that such tools sometimes provide an incomplete picture of the problem that fails to help developers locate the root causes. In addition to MVN and OSI, there are other dependency management platforms such as OSSIndex (Sonatype 2022), Snyk (Snyk 2022) and Libraries.io (Libraries.io 2022). We observed that OSSIndex excludes vulnerabilities from dependencies and only focuses on vulnerabilities from the library itself. Libraries.io (although covering a total of 32 package managers) does not list any vulnerability information. In the presence of multiple vulnerabilities, Snyk lists vulnerabilities in no particular order (no prioritization), while VulNet provides prioritization of the vulnerabilities in a library's dependencies. There are more platforms that provide dependency management in practice. In this paper, we focus only on two of these platforms (MVN and OSI) due to their rich features. In this paper, we first empirically uncover the issues in current vulnerability management platforms. Based on our findings, we propose VulNet to provide a better reporting of vulnerabilities from upstream dependencies, and provide a more informative vulnerability listing.

We conducted a user study to collect users' perceptions of the information we provided in the report. In RQ3, our approach was validated and rated 4.50 out of 5 by the participants.

3 An Empirical Study on Vulnerability Management in the Java Ecosystem

In this section, we manually analyze how MVN and OSI present vulnerabilities from dependencies in five dimensions. Our goals are to understand how the two vulnerability management platforms present the information, and whether there is room for improvement.

3.1 Studying Vulnerability Management in MVN and OSI

Manual Study Setup We conducted a study on the top 200 most popular libraries in the Maven Repository (Repository 2022b). Our study was conducted in November 2021, so the popularity is based on the usage by other libraries on the Maven Repository at that time. For each library, we categorize how the information is presented on MVN and OSI, and whether there is any difference. Our qualitative analysis involved the following three phases:

- *Phase I*: Select the top 200 most popular libraries in the Maven repository (Repository 2022b). For each library, we studied the version released closest to the end of 2019 to ensure the libraries would have enough time to receive vulnerability reports.
- *Phase II*: For each library, we compare the differences in its vulnerability information between the two platforms. We analyze the `pom.xml` files to study how the dependencies are configured. We also recursively analyze `pom.xml` for all the transitive dependencies if the platforms show that they may be affected by vulnerabilities.
- *Phase III*: We study the differences in the listed vulnerability between MVN and OSI. In particular, we study how the two platforms list the vulnerability and the relationship between the library and the vulnerable dependency to understand the reasons for the difference.

Below, we summarize how MVN and OSI present the vulnerability information along five dimensions: (i) transitive dependency, (ii) inherited dependency, (iii) dependency scope, (iv) prioritizing vulnerable dependencies, and (v) number of libraries included. For a detailed comparison of these dimensions, refer to Table 1. Among these, two dimensions are particularly pivotal:

Dependency Scope The term refers to the capability of a library information listing platform to enumerate the dependency scope information when delineating the dependency data of a library. The dependencies of a library could contain vulnerabilities that might be propagated to the library itself. When a library dependency is identified, not only the fact that there is a dependency relationship, but also the dependency scope (how the dependency is concretely used by the library) could affect how the vulnerability could impact the library. Therefore, we wish to study the dependency scope of a vulnerability to assess its impact, for example, whether the vulnerability would be introduced if a dependency is concretely used in the compilation or testing phase.

Number of Libraries Included This dimension reflects the count of libraries incorporated within the platform. Library dependencies are counted by one of the two platforms we studied. We cross-check such numbers to understand any discrepancy of the number, that is, missing

Table 1 A comparison between MVN and OSI along the five studied dimensions and the practical implications

Dimension	Maven repository	Open source insight	Practical implications
Transitive dependencies	Considers only direct dependencies.	Considers direct and transitive dependencies, excluding transitive optional ones.	Both platforms might not display all vulnerabilities from transitive dependencies.
Inherited dependency	Not considered.	Treats as direct dependencies.	MVN might overlook vulnerabilities from inherited dependencies.
Dependency scope	Classifies dependencies as compile v.s. provided or runtime v.s. test.	Not supported.	OSI's lack of scope distinction might cause false positives or inaccurate prioritization.
Prioritizing vulnerable dependencies	No prioritization.	Lists vulnerabilities with severity scores, but does not show which dependencies contain the vulnerabilities.	Absence of prioritization could hinder developers' tool adoption due to inefficiency in vulnerability management.
Num. of libraries included	Serves as the primary Maven ecosystem repository, hosting an extensive library list.	Omits some libraries. 16 out of 200 studied libraries were not included.	Incomplete library data on OSI might constrain developers in analyzing dependency vulnerabilities.

or redundant library dependencies. The incorrect number of library dependencies could lead to missing or false alarm reports of vulnerabilities.

Results. **We Find that 115/200 (57.5%) of the Studied Libraries Contain One or More Vulnerabilities when Considering Both the Library itself and its Dependencies** However, when only considering vulnerabilities from the library itself, and disregarding vulnerabilities from dependencies, the ratio significantly drops to 36/200 (18%). This underscores the pivotal role dependencies play in the overall vulnerability landscape of a library. We find that there are several differences in how the two platforms present vulnerability information, especially on the vulnerabilities introduced by the dependencies. Below, we discuss the differences and implications of each dimension.

Both MVN and OSI Consider Vulnerabilities in Direct Dependencies, But MVN Does not Consider Vulnerabilities from Transitive Dependencies Vulnerabilities in transitive dependencies may still cause security risks (Alqahtani et al. 2016; Pashchenko et al. 2020). For example, Google's report (Google 2022a) states that more than 80% of the software affected by the notorious vulnerability CVE-2021-44228 from `Log4j` did not use `Log4j` directly, but depended on it transitively. While MVN provides valuable information regarding vulnerabilities in direct dependencies, our findings indicate a limitation in its capacity to effectively track and display vulnerabilities stemming from transitive dependencies. Specifically, MVN is not able to identify vulnerabilities introduced through transitive dependencies in 31% of the studied vulnerable libraries, equating to 36 out of the 115 libraries that contain vulnerabilities. Developers should be aware that when using MVN to review the vulnerability information associated with a library, they may miss important vulnerabilities, especially coming from the transitive dependencies of a library.

MVN Does not Consider Inheritance in Dependency Management, Thus Developers May not be Aware of Those Vulnerable Dependencies Inherited from Parent Libraries The Maven build system offers an inheritance mechanism in dependency management, where a library can inherit the configurations defined in the `pom.xml` of a parent library. As an example, the excerpted Maven configuration below is from the library `logback-classic`, which uses the `<parent>` tag to inherit dependencies from the `1.2.4-groovyless` version of the `logback-parent` library. All the configuration information (e.g., version name, license, dependencies) from the parent library is inherited by the `logback-classic` library.

```
<parent>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-parent</artifactId>
  <version>1.2.4-groovyless</version>
</parent>
<artifactId>logback-classic</artifactId>
<packaging>jar</packaging>
<name>Logback Classic Module</name>
<description>logback-classic module</description>
```

Therefore, a library may become vulnerable if there is a vulnerability in any of its inherited dependencies. In total, we find that 29/115 (25%) of the vulnerable libraries may have vulnerabilities from inherited dependencies. However, while OSI does consider such vulnerabilities, MVN does not.

OSI Does not Categorize the Dependency Scope nor Distinguish Vulnerabilities in Test Dependencies. OSI Also Does not Consider Transitive Optional Dependencies When listing library dependencies and vulnerabilities, MVN takes into account various dependency scopes (i.e., *compile*, *runtime*, *provided*, and *test*). Providing the dependency scope of vulnerable dependencies allows developers to more accurately understand the types of dependencies and prioritize the vulnerability accordingly. In some cases, developers may use `<optional>` dependencies when these dependencies are only needed for certain features to save space and memory (Maven 2022). However, such optional dependencies can be transitive. Therefore, it is important to inform developers of possible vulnerabilities from such optional dependencies in case developers need to enable the feature. However, OSI only provides a list of dependencies and does not differentiate the dependency scope, which may cause false positives and challenges in prioritizing vulnerabilities.

Both OSI and MVN Lack Mechanisms to Prioritize Vulnerable Dependencies Prior studies (Farris et al. 2018; Liu et al. 2012; Huang et al. 2013; Le et al. 2022; Jung et al. 2022) found that although it is important to show developers all the potential vulnerabilities, a lack of prioritization mechanisms may cause distrust and prevent developers from adopting the tool. As shown in Fig. 1, OSI tries to show the severity score of each vulnerability to assist developers with prioritization. However, OSI is listing every individual vulnerability equally regardless of the dependency relationship, while in reality, multiple vulnerabilities may be associated with *one dependency*. Hence, having a prioritization mechanism based on dependency may help developers decide which dependency (e.g., dependency with the most vulnerabilities) they should update/replace/fix when trying to resolve vulnerabilities.

Although OSI shows potential vulnerabilities from transitive dependencies, we find that it does not provide any other information to help developers prioritize their investigation or fixing efforts. Since the number of transitive dependencies grows exponentially, a library may have hundreds or even thousands of transitive dependencies, and the depth of dependencies

is often large (e.g., more than three) (Liu et al. 2022; Valiev et al. 2018; Decan and Mens 2019). Such complex dependency relationships make it difficult for developers to understand the impact of a vulnerability (LaToza and Myers 2010; Aloraini 2020).

The List of Libraries is Incomplete on OSI MVN uses the central repository for the Maven build system, which contains a comprehensive list of libraries. However, we find that this may not be the case for OSI. Among the 200 studied libraries, 16 of them (8%) are not available on OSI. For example, Renjin (2022), which is an implementation of the R language on Java Virtual Machine, is not available on OSI. Renjin is one of the top 200 most popular libraries on MVN, but itself and all its subordinate libraries are not available on OSI. Since Google Open Source Insight is an experimental project with active development, the analyzed libraries are likely to be non-exhaustive. For these missing libraries, developers can only rely on MVN for vulnerability information.

3.2 Limitations of Current Vulnerability Management Platforms

Below, we summarize the limitations of the current vulnerability management platforms that we manually uncovered:

Limitation 1: Both Platforms do not Consider all Dependency Types and Scopes Library dependency graphs can be large and complex (Louridas et al. 2008; Decan et al. 2016). For the Maven build system, there are various dependency scopes, transitive dependencies, optional dependencies, etc. However, we find that both MVN and OSI miss some types of dependencies (e.g., MVN does not consider inherited and transitive dependencies, and OSI does not distinguish between test and other types of dependencies). Hence, MVN and OSI provide an incomplete view of the vulnerabilities due to missing dependencies.

Limitation 2: There is a Lack of Coherent and Consistent Presentation of Vulnerabilities in Dependencies The two platforms present the dependency and vulnerability information differently. For example, MVN shows the dependency scope (e.g., compile or runtime), while OSI only shows the dependencies as direct or transitive. Additionally, some libraries are not included by OSI. In short, it would be important to provide developers with a unified presentation of vulnerabilities for better consistency. The consistency would help developers to get accurate library information more clearly and easily.

Limitation 3: Both MVN and OSI Lack Proper Mechanisms that Help Developers Prioritize the Effort to Investigate and Resolve Vulnerable Dependencies Although both MVN and OSI provide some prioritization mechanisms, we find that such mechanisms are often insufficient. For example, MVN does not show the details of a vulnerability (e.g., severity score), and OSI does not show which dependencies are more vulnerable. In addition, while OSI shows transitive dependencies, some dependencies may have a long chain length in the dependency graph, which makes the associated vulnerabilities less likely to be reachable at run time (LaToza and Myers 2010; Aloraini 2020). In the next section, we discuss the design of our vulnerability management approach to address the above-mentioned limitations.

4 VulNet: A More Informed Vulnerability Management for the Java Ecosystem

Prior studies (Shen et al. 2011; Johnson et al. 2013; Chen et al. 2016; Barik 2016; Lipp et al. 2022; Nachtigall et al. 2022) found that the adoption of static security testing tools is highly

related to how the results are presented to developers. Similarly, we believe that a systematic and better organization of the vulnerability information helps developers with dependency vulnerability management. In this section, we present the design of VulNet, which provides more organized and finer-grained vulnerability management in a library's dependencies.

Based on our empirical findings and the uncovered limitations of MVN and OSI in Section 3, we follow the requirements described below when designing VulNet:

- **REQ1: VulNet should categorize the dependencies based on their properties and scopes (to address Limitations 1, and 2).** To assist developers in understanding the used dependencies and the associated vulnerabilities, VulNet should categorize the dependencies and the associated vulnerabilities based on their scope, and provide a unified dependency information presentation.
- **REQ2: VulNet should provide a mechanism to help developers prioritize vulnerable dependencies (to address Limitation 3).** To assist developers in prioritizing the dependencies, VulNet should provide various information, such as the overall vulnerability severity, depth of the dependency, and the prioritization of dependency scopes.

Below, we discuss the design of VulNet that fulfills the above-mentioned requirements.

4.1 REQ1: Categorizing the Dependencies Based on their Properties and Scopes

Including all Dependency Types and Scopes Since dependency types and scopes have a direct effect on how a vulnerability propagates through the dependency graph, it is important to include all dependency types and scopes. Based on our findings in Section 3, we combine the dependency types and scopes provided by both MVN and OSI. VulNet includes all the dependency scopes: compile, runtime, provided, test, optional, and inherited dependencies from parent libraries. VulNet also lists if a dependency is of depth one (i.e., direct dependency) or more than one (i.e., transitive dependency).
Categorizing the Dependencies based on Dependency Scope Since different dependency scopes pose different risks from vulnerabilities, we categorize the dependencies based on the scopes (e.g., compile vs test). For example, developers can use VulNet to quickly identify all the test dependencies, where vulnerabilities in such dependencies may have a lower risk.

4.2 REQ2: Providing Mechanisms to Help Developers Prioritize Vulnerable Dependencies

Provide a Priority Ranking for the Dependencies Based on Scopes and Associated Vulnerabilities To assist developers with prioritizing the fixing and investigation effort for vulnerable dependencies, we propose a ranking mechanism for the dependency scopes. The ranking is meant to provide a recommendation and we provide a rationale behind the ranking decision. Prior studies found that providing the decision-making process help improve developers' trust in tool adoption (Falessi et al. 2011; Saaty 1994). We rank direct dependencies as critical (they are directly used by a library), transitive dependencies as high, optional transitive optional dependencies as medium (they need to be enabled by developers explicitly), and test dependencies as low. Moreover, we show how many vulnerabilities are associated with one dependency, as some dependencies may have one and some may have tens of associated vulnerabilities. Hence, developers can decide which dependencies they should first upgrade or fix based on the number of associated vulnerabilities and dependency scope.

Ranking the Vulnerabilities Based on Severity In addition to showing the number of vulnerabilities associated with one dependency, we also list the severity scores for each vulnerability (First 2022; Liu et al. 2012). NVD website provides the Common Vulnerability Scoring System (CVSS) to rank the severity of a vulnerability. CVSS ranges from 1 to 10 (the highest severity). For every vulnerability, we retrieve the severity score from the CVE website and rank the dependencies based on the highest score of the associated vulnerabilities.
Ranking Transitive Dependencies Based on the Dependency Depth Prior studies found that transitive dependencies with a larger depth are more likely to be unused (i.e., bloated) (Soto-Valero et al. 2021; Latendresse et al. 2022). In addition, in a large system where there are many dependencies, the depth of the dependency graph can be very large. Investigating the impact of the vulnerabilities associated with transitive dependencies in a large depth can be challenging and time-consuming. Therefore, VulNet provides the dependency depth information for every transitive dependency to help developers prioritize investigation efforts.

We implement VulNet as a web platform based on Python. We implement a web crawler to collect information about the dependencies and vulnerabilities for every version of the top 200 most popular libraries and all their upstream dependencies. The crawler analyzes both the Maven repository and CVE website to collect both the library and vulnerability information. We analyze both MVN and `pom.xml` files to get the dependencies. This is the result of traversing the dependency graph. We consider this dependency graph as the Ground Truth. VulNet builds the same dependency graph based on the ground truth to capture all possible vulnerabilities at the manifest level (based on the `pom.xml` file). In total, we collected 19,886 versions of the top 200 libraries and a total of 623,620 dependencies for all the versions of these 200 libraries. Note that our crawler can be easily extended to retrieve the information for other libraries. We store the library, its dependencies, and vulnerability information in a database. We use a triplet $(GroupID, ArtifactID, Version)$ to uniquely identify a Maven library and represent the dependencies as a graph. Therefore, by doing a search, we are able to identify all the dependencies of a given library. Our processed data is available online (Ma et al. 2022).

A crucial feature of our tool is its ability to classify vulnerabilities introduced by dependencies. We achieve this by analyzing the specific usage of these dependencies. This analysis allows us to prioritize the vulnerability processing of the library based on the classification. By understanding the different usages of dependencies, VulNet can provide a comprehensive display of all vulnerabilities in the library and sort them effectively.

Figure 3 shows an excerpt screenshot of VulNet for the library `spark-core_2.11:1.2.2` . VulNet groups the dependencies based on the scope (e.g., direct compile) and the scope's priority (e.g., critical). VulNet further shows the vulnerabilities and their severity scores associated with each dependency. For example, `Log4j 1.2.17` has four vulnerabilities, where two of which have a severity score of 9.8. Hence, developers can quickly identify the dependencies that have more severity and propose a solution (e.g., replace or upgrade `Log4j`). VulNet also provides the depth of the dependencies. For example, `jackson-mapper-asl` has a depth of three. Developers can consider the transitivity depth information and prioritize their effort when investigating and resolving vulnerability dependencies. The dependencies are sorted based on the number of vulnerabilities, the highest severity score, and the depth. Developers can also unfold the information layout (HTML table) to examine the remaining dependencies easily.

Direct Compile Dependency (39) (Priority:CRITICAL) Directly used

Depth	Group / Artifact	Version	Vulnerabilities
1	log4j » log4j	1.2.17	CVE-2019-17571 (9.8) CVE-2022-23305 (9.8) CVE-2022-23302 (8.8) CVE-2021-4104 (7.5)
1	io.netty » netty-all	4.0.23.Final	CVE-2019-16869 (7.5)
1	com.clearspring.analytics » stream	2.7.0	

Unfold the table

Transitive Compile and Runtime Dependency (109) (Priority:HIGH) Indirectly used

Depth	Group / Artifact	Version	Vulnerabilities
2	org.eclipse.jetty » jetty-webapp	8.1.14.v20131031	CVE-2020-27216 (7.0)
2	com.google.protobuf » protobuf-java	2.5.0	CVE-2021-22569 (5.5)
3	org.codehaus.jackson » jackson-mapper-asl	1.8.8	CVE-2019-10172 (7.5)

Unfold the table

Fig. 3 A screenshot of VulNet for library spark-core_2.11:1.2.2

5 Evaluation of VulNet

RQ1: How Does the Dependency Categorization Provided by VulNet Compare Against State-of-the-Art Vulnerability Management Platforms?

Motivation Based on the issues that we uncovered in Section 3, we proposed VulNet, which aims to provide more organized vulnerability management for library dependencies. In this RQ, we compare the distribution and hierarchical categorization of the library dependencies and the associated vulnerabilities across MVN, OSI, and VulNet. The results provide an initial insight into how each platform presents the dependencies and the associated vulnerabilities.
Approach We conduct our study on all the versions of the top 200 libraries on the Maven repository. We collected the data in November 2021. In total, there are 19,886 versions for these 200 libraries. We analyze the direct and transitive dependencies for all the libraries. As we discussed in Sections 2 and 3, different dependency types and scopes may have different implications on how a vulnerability propagates through the dependencies (e.g., vulnerabilities from the transitive dependencies of a test dependency should not have a real impact). Therefore, in this RQ, we first illustrate how the three vulnerability management platforms present the hierarchical categorization of library dependencies. We categorize the dependencies based on the scope and calculate the average number of dependencies for each library. We divide the dependencies into two broad categories: *direct*, and *transitive*, and list the number of dependencies that belong to each scope. We also manually verify the number of dependencies reported by VulNet on 100 randomly selected libraries. Finally, we study how the vulnerabilities are distributed among the different dependency scopes to understand the prevalence of vulnerabilities in the software supply chain context.
Results. ***OSI and MVN Exclude an Average of 65.8% and 90.5% of the Dependencies per Library, Respectively*** Table 2 shows a hierarchical view of the analyzed dependency scopes. We show the average statistics across all versions of these 200 libraries. OSI would list the required dependencies and test dependencies of the transitive dependencies, but OSI does not categorize transitive dependencies by the dependency scope. In our hierarchical view, we link the transitive dependencies with dotted lines. Because MVN does not consider transitive dependencies and OSI does not consider transitive optional dependencies, both

Table 2 The average (mean) number of dependencies and their distribution for all the versions of the 200 analyzed libraries across different platforms

Type of Dependency	*VulNet*	*MVN*	*OSI*
Dependency	77.85	7.40	26.61
Direct	7.63 (9.80%)	7.40	7.63
Direct / **Non-test**	**5.87 (7.54%)**	**5.85**	
Direct / Non-test / **Compile**	**5.45** (7.00%)	**5.43**	
Direct / Non-test / **Runtime**	**0.14** (0.18%)	**0.14**	
Direct / Non-test / **Provided**	**0.28** (0.36%)	**0.28**	
Direct / Test	1.76 (2.26%)	1.55	
Transitive	70.22 (90.20%)		18.98
Transitive / **Non-test**	**63.11 (81.07%)**		**11.87**
Transitive / Non-test / **Compile&Runtime**	**58.34 (74.94%)**		**10.85**
Transitive / Non-test / Compile&Runtime / **Required**	**10.85** (13.94%)		**10.85**
Transitive / Non-test / Compile&Runtime / *Optional*	*47.49* (61.00%)		
Transitive / Non-test / **Provided**	**4.77 (6.13%)**		**1.02**
Transitive / Non-test / Provided / **Required**	**1.02** (1.31%)		**1.02**
Transitive / Non-test / Provided / *Optional*	*3.75* (4.82%)		
Transitive / Test	7.11 (9.13%)		7.11

Dependency scopes with an immediate negative impact are shown in **Bold** font, and those that may or may not have an impact (i.e., optional) are shown in *Italics*. Even though OSI does not distinguish dependency types, we use dashed lines to show the distribution of transitive dependencies for comparison purposes

MVN and OSI would only include 7.40 and 26.61 dependencies on average, respectively. In contrast, VulNet introduces a more comprehensive dependency graph, with an average of 77.85 dependencies per library. Due to the exclusion of inherited dependencies, MVN excludes an average of 0.23 direct dependencies per library, while OSI and VulNet show the same average number of direct dependencies (i.e., 7.63). Additionally, our manual study on the 100 randomly selected libraries shows that the number of dependencies reported by VulNet matches the actual number of dependencies, which further demonstrates the high accuracy of VulNet.

Even though VulNet includes more dependencies, the categorization provided by VulNet helps break down the dependency based on the scope. For example, developers can be informed that a library has an average of 47.49 optional transitive dependencies. Vulnerabilities in optional dependencies may still have an effect if developers choose to enable the optional features. Therefore, VulNet lists optional transitive dependencies separately and allows developers to decide whether they need these optional features or not. Moreover, VulNet helps developers distinguish test and source dependencies, as test dependencies are excluded in the final executable and binary by default (Apache 2022c).

As shown in Table 2, a library has an average of 1.76 test dependencies and 7.11 transitive test dependencies. However, when using OSI, developers would only see there are an average of 7.63 direct and 18.98 transitive dependencies, and would not be able to distinguish between test and other types of dependencies, which could play a role in analyzing the impact of vulnerability especially those from library dependencies.

On Average, 27% of the Vulnerabilities Listed by OSI in a Library May be Invalid and MVN Excludes an Average of 21.18 Potential Vulnerabilities Table 3 shows the average number of potential vulnerabilities associated with each dependency scope.

For each library studied through our tool, we collect the vulnerabilities that each library itself has been exposed to from the "Vulnerabilities from the library itself" section of Maven Repository. Since MVN does not consider dependencies inherited from parent libraries,

Table 3 The average (mean) number of vulnerabilities that are associated with each dependency scope/type in all the versions of the 200 analyzed libraries

Type of Dependency	*VulNet*	*MVN*	*OSI*
Dependency	28.75	5.19	11.32
├─Direct	5.39	5.19	5.39
│ ├─Non-test	**4.53**	**4.41**	
│ │ ├─**Compile**	**4.31**	**4.19**	
│ │ ├─**Runtime**	**0.04**	**0.04**	
│ │ └─**Provided**	**0.18**	**0.18**	
│ └─Test	0.86	0.78	
└─Transitive	23.36		5.93
├─**Non-test**	**21.18**		**-3.75**
│ ├─Compile and Runtime	19.03		-3.40
│ │ ├─**Required**	**3.40**		**-3.40**
│ │ └─*Optional*	*15.63*		
│ └─**Provided**	**2.15**		**-0.35**
│ ├─**Required**	**0.35**		**-0.35**
│ └─*Optional*	*1.8*		
└─Test	2.18		-2.18

Dependency scopes with an immediate negative impact are shown in **Bold** font, and those that may or may not have an impact (i.e., optional) are shown in *Italics*. Even though OSI does not distinguish dependency types, we use dashed lines to show the distribution of transitive dependencies for comparison purposes

MVN only shows an average of 5.19 vulnerabilities for direct dependencies as opposed to 5.39 as shown by OSI and VulNet. Moreover, MVN excludes an average of 21.18 potential vulnerabilities from non-test transitive dependencies as it does not consider any transitive dependency. In short, MVN does not show most of the potential vulnerabilities. OSI, on the other hand, lists an average of 5.39 and 5.93 vulnerabilities from direct and transitive dependencies, respectively. However, as we found and shown in VulNet, an average of 0.86 and 2.18 vulnerabilities are related to direct and transitive test dependencies. In other words, 27% (average number of vulnerabilities from test dependencies divided by the total number of vulnerabilities) of the vulnerabilities in a library are from test dependencies, which often do not affect the final produced binary. Such a high percentage of invalid vulnerabilities may affect developers' trust and adoption of vulnerability management tools (Imtiaz et al. 2021; Barik 2016; Lipp et al. 2022).

Currently, both MVN and OSI lack details when categorizing vulnerabilities in library dependencies. Thus, when developers resort to leveraging these platforms, they may not be informed of the most probable vulnerabilities. In contrast, VulNet categorizes the dependency based on the scope (e.g., test vs non-test and optional vs required), which may help developers get a more informed list of vulnerabilities based on the nature of the associated dependencies. When developers examine the result provided by VulNet, they can decide which vulnerabilities in a dependency are more likely to cause risks and should be resolved (e.g., by upgrading or by replacing the dependency).

> 90.5% and 65.8% of the dependencies are not covered by MVN and OSI, respectively, when listing dependencies. At the same time, vulnerabilities from the ignored dependencies cannot be enumerated. Categorizing dependencies based on the types may help reduce 27% of the invalid (or less impactful) vulnerabilities listed by OSI (such as those from test dependencies).

RQ2: How are the Vulnerable Dependencies and Vulnerabilities Distributed Across Different Dependency Depths?

Motivation VulNet ranks the dependencies based on the dependency depth and the severity score of the vulnerabilities. In this RQ, we conduct a quantitative study to provide an overview of the prioritization approach. We study the average number of dependencies, the average number of vulnerable dependencies and the average number of vulnerabilities introduced by dependencies at different depths. We also calculated the distribution of the mean vulnerability severity score at different depths. We wish to provide insights into how the management of vulnerabilities associated with a library's dependencies can be prioritized. Especially, when the number of vulnerable dependencies is large, the prioritization of vulnerability mitigation that matters the most to developers can improve the productivity for the stakeholders.
Approach Similar to RQ1, we conduct our study on all the versions of the top 200 libraries on the Maven repository (collected the data in November 2021). We analyze the depth of required non-test dependencies and the depth of transitive optional non-test dependencies for all the libraries. As we discussed in Section 3, lacking the prioritization of vulnerable dependency may prevent developers from effectively using a library due to the effort to filter out those vulnerabilities without a concrete impact. Therefore, in this RQ, we first analyze the distribution of the depth of dependencies, the depth of vulnerable dependencies, and the number of associated vulnerabilities. Then, we analyze the mean severity score for vulnerabilities at different depths.

Results. ***In General, the Mean Number of Vulnerable Dependencies and the Mean Number of Vulnerabilities Decrease as the Dependency Depth Increases*** Table 4 shows the mean number of dependencies, vulnerable dependencies, vulnerabilities, and severity score for our analyzed libraries. We grouped the results based on different dependency depths and types (i.e., required and optional). We exclude test dependencies in the results since they are excluded (by default) in the final binary executables. Note that when the depth value is one, it represents a direct dependency.

We find that, as the dependency depth increases, the average number of vulnerable dependencies and vulnerabilities decreases. The trend is the same for both required and optional dependencies. When the depth is one, there are 2.76 vulnerable dependencies with 5.53 vul-

Table 4 The average (mean) number of dependencies, vulnerable dependencies, vulnerabilities, and severity score per depth level

Dependency type	Depth	Num. dependency	Num. vulnerable dependency	Num. vulnerability	Vulnerability severity score
Required	1	6.20	2.76	5.53	7.70
	2	9.68	1.95	4.01	7.02
	3	7.14	1.71	2.73	7.40
	4	4.51	1.66	2.11	7.25
	5+	7.70	2.55	3.90	7.32
Optional	2	24.59	4.99	6.10	7.90
	3	32.94	4.08	2.72	7.45
	4	33.56	3.28	3.73	8.18
	5+	78.05	6.13	5.31	7.73

Note that we exclude test dependencies in the result

nerabilities (for required dependencies), while there are only 1.95 vulnerable dependencies and 4.01 vulnerabilities when the depth increases to two.

Overall, Vulnerabilities Associated with Direct Dependencies Have a Higher Average Severity Score Compared to Transitive Dependencies Vulnerabilities may cause different consequences due to different severity levels. Therefore, vulnerabilities with a higher severity score should often be investigated and resolved first (Le et al. 2022). Based on our findings, we find that vulnerabilities in direct required dependencies have a higher average severity score (7.70) compared to transitive required dependencies (severity scores range from 7.02 to 7.40). In other words, in the analyzed libraries, vulnerabilities in direct dependencies are more likely to cause severe consequences. For optional dependencies, the severity score for the vulnerabilities with depth two is 7.90, which is the second highest among all the depth levels. However, since optional dependencies are only enabled if developers require the optional features, their impact may be more manageable and may only be concerned by developers who actually use the specific features in such dependencies.

> Our findings show that vulnerable direct dependencies often have more vulnerabilities with higher severity scores. Such vulnerabilities may be easier to investigate due to direct usage. Hence, resolving vulnerable direct dependencies might be more efficient, while developers should not neglect potentially vulnerable transitive dependencies based on their concrete use scenarios for such dependencies.

RQ3: What are Practitioners' Feedbacks on VulNet?

Motivation In the previous RQs, we conducted empirical studies on how the dependencies and vulnerabilities are organized based on the features we proposed in VulNet. Nevertheless, to assess the usefulness of VulNet, it is important to understand practitioners' perceptions on VulNet, the limitations of the existing vulnerability management platforms, and the extent to which VulNet addresses them.

Approach To receive feedback from practitioners, we conducted a survey involving 24 participants (8 professional developers and 16 related field researchers such as graduate students, postdoctoral fellows and professors). These participants have two to twenty years (average of 6.65) of experience in Maven for Java development or testing. The participants were asked to provide a numerical rating on a 5-point scale (i.e., "Strongly agree", "Agree", "Neutral", "Disagree", and "Strongly disagree", where 5 refers to "Strongly agree") on how each one of MVN, OSI, and VulNet present vulnerability information.

The survey is composed of three parts: 1) background information (i.e., years of experience in software development); 2) practitioners' perspectives on how the three platforms present dependencies and vulnerabilities; and 3) practitioners' perspectives on how the three platforms prioritize vulnerabilities. For part 2 and part 3, we first show how each platform presents the information using both screenshots and live web pages. Then, we ask the participants to rate each platform. Finally, we ask the participants to rate the usefulness of the features that we proposed in VulNet: grouping the dependencies based on scope, grouping the vulnerabilities based on the dependency that they belong to, and prioritization based on dependency depth and severity scores. Note that, for each question, we also ask if participants have further comments or ideas regarding dependency vulnerability management.

Results. ***Developers Opine that VulNet Offers more Informed Vulnerability Management and Helps Better Prioritize Vulnerability Mitigation Efforts, with an Average Rating of 4.50, Compared to MVN (2.83) and OSI (3.14)*** Table 5 shows the ratings on the three platforms with respect to vulnerability listing and prioritization from the participants who are experienced in software development, with an average experience of 6.65 years.

Overall, VulNet received a much higher average rating (4.50) compared to MVN (2.83) and OSI (3.14). In both vulnerability listing and prioritization, VulNet received an average rating of 4.38 and 4.58, and all the participants either agree or strongly agree that VulNet provides useful information. Our findings show that the participants acknowledge and appreciate the information that VulNet provides over MVN and OSI.

Practitioners Gave VulNet a Rating of 4.43 and 4.42 on Grouping the Vulnerability by Dependency Scope and Listing the Vulnerabilities that are Associated with each Dependency

Grouping dependencies by scope helps developers better prioritize their efforts toward *in-scope* vulnerabilities, which is rated (4.43) by participants. One participant mentioned: "*If the vulnerability is from a test dependency, I may ignore it.*" Another participant mentioned: "*The reason for grouping is helping make correlations and building better dependencies.*" Additionally, participants gave a rating of 4.42 on VulNet's feature of listing the vulnerability of every single dependency. One participant mentioned: "*Listing every single vulnerability correlated with dependencies creates a better picture and may be used in the future when new vulnerabilities are discovered. In my opinion, this is as important as doing regression testing whenever a new feature is added to a software component.*" In short, the participants agree that providing the number of vulnerabilities from each dependency helps them prioritize and understand the overall quality of the software better.

Practitioners Gave VulNet a Rating of 4.35 on Prioritizing Transitive Dependencies Based on Dependency Depth, and a Rating of 4.87 on Ranking the Vulnerable Dependencies Based on Severity Scores VulNet provides the dependency depth information to help developers understand where the vulnerable dependencies are located in the dependency graph. Such information allows developers to know which vulnerable dependencies are more likely to have an impact on the library. One participant mentioned: " *I will be interested in the complete vulnerabilities that may affect my application. In the second step, I will go through the vulnerabilities of each dependency.*" Additionally, VulNet shows the vulnerability impact for a dependency (i.e., the number of associated vulnerabilities and their severity scores). This may help developers prioritize the mitigation effort when resolving the issue by replacing or upgrading the dependency. One participant mentioned: "*I think developers care more about the severity and the rank of severity.*"

VulNet received a higher rating (4.50) than either MVN (2.83) or OSI (3.14), both overall and from every aspect. The participants also gave positive feedback on VulNet's vulnerability prioritization approach based on dependency depth and vulnerability impact (with an average rating of 4.35 and 4.87, respectively).

Table 5 The average rating (from a scale of 1 to 5, where 5 is strongly agree) on how each of the three platforms present vulnerability listing and prioritization

	MVN	OSI	VulNet
Vulnerability listing	3.17	3.04	4.38
Vulnerability prioritization	2.50	3.23	4.58
Overall average	2.83	3.14	4.50

6 Discussion and Implications

We discuss the implications of our study for practitioners and researchers, respectively.

Implication for Practitioners In this paper, we show the limitations of the state-of-the-art platforms: MVN, and OSI. We also find that practitioners using these platforms may not be able to get complete and precise vulnerability information.

Hence, practitioners who use these platforms in the future may refer to our study to know their limitations. We also present VulNet, which provides a better approach to organizing and presenting the vulnerability information. Practitioners may leverage VulNet in the future to manage and understand the vulnerabilities in the used dependencies.

Implication for Researchers Various limitations of the state-of-the-art vulnerability management platforms fuel the need for better vulnerability management, not just for Java but for other programming languages (C/C++) as well. Due to the complexity of the software dependency graph, some dependencies may be bloated or the vulnerable code may not be used by parent libraries (Gkortzis et al. 2019). Future research may study the effectiveness of a finer-grained analysis, i.e., at the class/method/statement level to localize and understand the actual impact (reachability) of vulnerabilities.

Our research is based on the Maven ecosystem for the Java programming language and the associated ecosystem. Future studies may further investigate how vulnerability management is done in different ecosystems (e.g., NPM) and whether there are any similar or different challenges. Finally, one participant in the survey mentioned that "*'vulnerabilities from deep dependencies may have a lower impact' is not always true. Some deep dependencies may also be used frequently and affect libraries.*" Future studies may leverage the findings in our study and conduct a vulnerability impact analysis at a finer-grained level (e.g., class or method) and evaluate the true impact of dependencies with different dependency depth levels.

7 Threats to Validity

This section discusses the limitations and potential threats to the validity of our experimental observations.

External Validity Our observations and study are based on the top 200 Java projects (conducted in November 2021) built by Maven Repository and our findings are restricted to this scope.

We selected the top 200 popular libraries and, all their versions (19,886 versions in total) as a starting point. Then, we recursively collect and analyze all of the transitive dependencies (and the transitive dependencies of all the associated dependencies) that are available (a total of 274,236 libraries and 623,620 versions). For libraries that are less utilized by the open-source community, they might not be included in our dataset. However, VulNet is constructed to be easily extendable to all varieties of libraries, regardless of their usage frequency. We released our code and dataset which could help future research to extend the study and verify on more libraries. Our findings may not generalize to other dependency management systems (e.g., Snyk (Snyk 2022) and Libraries.io (Libraries.io 2022)). To mitigate this threat to some extent, we compared the vulnerability information for some known vulnerable libraries across MVN, OSI, OSSIndex, Snyk, and Libraries.io. We observed that OSSIndex excludes vulnerabilities from dependencies and only focuses on vulnerabilities from the library itself. Libraries.io (although covering a total of 32 package managers) fails to list any vulnerability information. In the presence of multiple vulnerabilities, Snyk lists vulnerabilities in no particular order (no prioritization), while VulNet provides prioritization of the vulnerabilities

in a library's dependencies. There are more systems that provide dependency management in practice. In this paper, we focus only on two of these platforms (MVN and OSI). More insights to improve software dependency and its associated vulnerability management can be found by studying other systems in the future. We believe that further studies along this line of comparison could support the enrichment offered by VulNet. Another external validity is related to the evolving nature of vulnerability data and the analyzed platforms. The features and data on MVN and OSI may change over time due to new development, feature addition, and newly discovered vulnerabilities and reported CVEs. However, the implications of our findings and developed solutions should remain unchanged.
Construct Validity We conducted empirical studies on how state-of-the-art platforms list vulnerable dependencies. We discuss the limitations and differences between the two platforms. Our analysis may be subjective and others may have different opinions on the issues we found. While providing user feedback on VulNet, the participants may have biases from prior experiences of using MVN or OSI. However, our survey shows that the participants highly agree with our findings and the approach we proposed in VulNet.

8 Conclusions

In this paper, we performed an empirical study on two state-of-the-art vulnerability management platforms (MVN and OSI), and identified their lack of prioritization of vulnerability information provided to developers, categorization of different kinds of dependencies, and overestimation or underestimation of the number of dependencies and their associated vulnerabilities. To address these limitations, we introduced VulNet. The motivation was to furnish more effective information so that it helps prioritize relevant vulnerability information and guide developers' mitigation efforts towards several downstream software quality management tasks, such as library migration, vulnerability localization and removal. We validated the usefulness of VulNet using a user study that involved a comparison with MVN, and OSI. Overall, VulNet was highly appreciated and rated by the practitioners and they opine that it offers more effective vulnerability management capability.

Data Availibility The datasets generated during and analyzed during the current study are available in the `VulNet` repository, https://github.com/SPEAR-SE/Vulnet.

Declarations

Conflict of Interest The authors declared that they have no conflict of interest.

References

Alfadel M, Costa DE, Mokhallalati M, Shihab E, Adams B (2020) On the threat of npm vulnerable dependencies in Node.js applications

Alfadel M, Costa DE, Shihab E (2021) Empirical analysis of security vulnerabilities in Python packages. In: Proceedings of the 28th IEEE international conference on software analysis, evolution and reengineering (SANER'21)

Aloraini B (2020) Towards better static analysis security testing methodologies. PhD thesis: https://uwspace.uwaterloo.ca/handle/10012/16359. Accessed 8 Aug 2022

Alqahtani SS, Eghan EE, Rilling J (2016) SV-AF - a security vulnerability analysis framework. In: 2016 IEEE 27th international symposium on software reliability engineering (ISSRE). pp 219–229

Apache (2022a) Log4j - apache log4j 2. https://logging.apache.org/log4j/2.x/. Accessed 24 Nov 2022

Apache (2022b) Maven - introduction to the dependency mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#dependency-scope. Accessed 17 Aug 2022

Apache (2022c) Maven-Maven documentation. https://maven.apache.org/guides/. Accessed 23 Aug 2022

Barik T (2016) How should static analysis tools explain anomalies to developers? In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016. pp 1118–1120

Chen TH, Shang W, Hassan AE, Nasser M, Flora P (2016) Detecting problems in the database access code of large scale systems: an industrial experience report. In: Proceedings of the 38th international conference on software engineering companion, ICSE'16. pp 71–80

Croft R, Xie Y, Zahedi M, Babar MA, Treude C (2021) An empirical study of developers' discussions about security challenges of different programming languages. arXiv:2107.13723

Decan A, Mens T (2019) What do package dependencies tell us about semantic versioning? IEEE Trans Software Eng 47(6):1226–1240

Decan A, Mens T, Claes M (2016) On the topology of package dependency networks: a comparison of three programming language ecosystems. In: Proccedings of the 10th European conference on software architecture workshops, ECSAW'16

Decan A, Mens T, Constantinou E (2018) On the impact of security vulnerabilities in the Npm package dependency network. In: Proceedings of the 15th international conference on mining software repositories, MSR'18. pp 181–191

Düsing, J. and Hermann B (2022) Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. Digital Threats 3(4)

Epperson W, Wang A, DeLIne R, Drucker S (2022) Strategies for reuse and sharing among data scientists in software teams. In: ICSE 2022

Falessi D, Cantone G, Kazman R, Kruchten P (2011) Decision-making techniques for software architecture design: a comparative survey. ACM Comput Surv 43:33

Farris KA, Shah A, Cybenko G, Ganesan R, Jajodia S (2018) VULCON: a system for vulnerability prioritization, mitigation, and management. ACM Trans Priv Secur 21(4)

First (2022) Common vulnerability scoring system SIG. https://www.first.org/cvss/. Accessed 26 Aug 2022

Frakes W, Kang K (2005) Software reuse research: status and future. IEEE Trans Software Eng 31(7):529–536

GitHub (2020) github-octoverse-2020-security-report. https://octoverse.github.com/2020/

Gkortzis A, Feitosa D, Spinellis D (2019) A double-edged sword? software reuse and potential security vulnerabilities. In: Peng X, Ampatzoglou A, Bhowmik T (eds) Reuse in the big data era. pp 187–203

Gkortzis A, Feitosa D, Spinellis D (2021) Software reuse cuts both ways: an empirical analysis of its relationship with security vulnerabilities. J Syst Softw 172:110653

Google (2022a) Google online security blog: Understanding the impact of apache log4j vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html. Accessed 24 Nov 2022

Google (2022b) Open source insights. https://deps.dev/. Accessed 05 Aug 2022

Google (2022c) Open source insights. https://deps.dev/faq. Accessed 12 Oct 2022

Harrand N, Benelallam A, Soto-Valero C, Bettega D, Barais O, Baudry B (2020) API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client-API usages. J Syst Softw 184:111134

Huang CC, Lin FY, Lin FYS, Sun YS (2013) A novel approach to evaluate software vulnerability prioritization. J Syst Softw 86(11):2822–2840

Imtiaz N, Thorn S, Williams L (2021) A comparative study of vulnerability reporting by software composition analysis tools. In: Proceedings of the 15th ACM / IEEE international symposium on empirical software engineering and measurement (ESEM), ESEM'21

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 international conference on software engineering, ICSE'13. pp 672–681

Jung B, Li Y, Bechor T (2022) CAVP: a context-aware vulnerability prioritization model. Comput Secur 116:102639

Kula R, German D, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23:1–34

Latendresse J, Mujahid S, Costa DE, Shihab E (2022) Not all dependencies are equal: an empirical study on production dependencies in NPM

LaToza TD, Myers BA (2010) Developers ask reachability questions. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 1, ICSE'10. pp 185–194

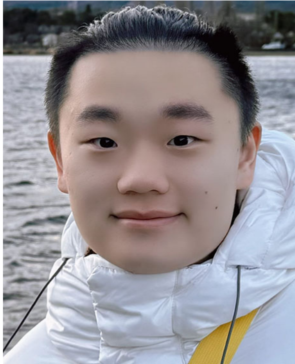
Le THM, Chen H, Babar MA (2022) A survey on data-driven software vulnerability assessment and prioritization. ACM Comput Surv

Lemos R (2022) Dependency problems increase for open source components. https://www.darkreading.com/application-security/dependency-problems-increase-for-open-source-components. Accessed 05 Aug 2022

Libraries.io (2022) Libraries.io-The open source discovery service. https://libraries.io/. Accessed 14 Nov 2022
Lipp S, Banescu S, Pretschner A (2022) An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2022. pp 544–555
Liu C, Chen S, Fan L, Chen B, Liu Y, Peng X (2022) Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In: 2022 IEEE/ACM 44th international conference on software engineering (ICSE). pp 672–684
Liu Q, Zhang Y, Kong Y, Wu Q (2012) Improving VRSS-based vulnerability prioritization using analytic hierarchy process. J Syst Softw 85(8):1699–1708
Louridas P, Spinellis D, Vlachos V (2008) Power laws in software. ACM Trans Softw Eng Methodol 18(1)
Ma Z, Mondal S, Chen THP, Zhang H (2022) Vulnet. https://github.com/SPEAR-SE/Vulnet
Massacci F, Pashchenko I (2021) Technical leverage in a software ecosystem: development opportunities and security risks. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). pp 1386–1397
Maven (2022) Maven-optional dependencies and dependency exclusions. Accessed 17 Aug 2022
Mitre (2022) Cve-cve-2021-44228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228. Accessed 17 Nov 2022
Mojica IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2014) A large-scale empirical study on software reuse in mobile apps. IEEE Softw 31(2):78–86
Nachtigall M, Schlichtig M, Bodden E (2022) A large-scale study of usability criteria addressed by static analysis tools. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2022. pp 532–543
Oracle (2022) JDBC drivers | oracle. https://www.oracle.com/ca-en/database/technologies/appdev/jdbc.html. Accessed 12 Oct 2022
Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2018) Vulnerable open source dependencies: counting those that matter. Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement
Pashchenko I, Plate H, Ponta S, Sabetta A, Massacci F (2020) Vuln4Real: a methodology for counting actually vulnerable dependencies. IEEE Trans Softw Eng 48(01):1–1
Prana G, Sharma A, Shar LK, Foo D, Santosa A, Sharma A, Lo D (2021) Out of sight, out of mind? How vulnerable dependencies affect open-source projects. Empirical Software Engineering 26
Renjin (2022) Renjin | Integrating R and Java | The JVM-based interpreter for the R language for statistical computing. https://www.renjin.org/. Accessed 08 Sep 2022
Repository M (2022a) Maven Repository: Search/Browse/Explore. https://mvnrepository.com/. Accessed 05 Aug 2022
Repository M (2022b) Maven repository: top projects at Maven repository. https://mvnrepository.com/popular. Accessed 06 Aug 2022
Ruiz IJM, Nagappan M, Adams B, Hassan AE (2012) Understanding reuse in the Android Market. In: 2012 20th IEEE international conference on program comprehension (ICPC). pp 113–122
Saaty TL (1994) Fundamentals of decision making and priority theory with the analytic hierarchy process. RWS publications
Shen H, Fang J, Zhao J (2011) EFindBugs: effective error ranking for FindBugs. In: 2011 Fourth IEEE international conference on software testing, verification and validation. pp 299–308
Smith J, Johnson B, Murphy-Hill E, Chu B, Lipford HR (2015) Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015. New York USA, pp 248–259
Snyk (2022) Snyk vulnerability database | Snyk. https://security.snyk.io/. Accessed 14 Nov 2022
Sonatype (2022) Sonatype oss index. https://ossindex.sonatype.org/. Accessed 08 Mar 2023
Soto-Valero C, Harrand N, Monperrus M, Baudry B (2021) A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Softw Engg 26(3)
Synopsys (2022). Synopsys | EDA tools, semiconductor IP and application security solutions. https://www.synopsys.com/. Accessed 05 Aug 2022
Valiev M, Vasilescu B, Herbsleb J (2018) Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. pp 644–655
Zerouali A, Mens T, Decan A, De Roover C (2022) On the impact of security vulnerabilities in the NPM and RubyGems dependency networks. Empir Softw Eng 27(5):1–45

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.



Zeyang Ma received a bachelor's degree in Computer Science (Sino-French cooperation program) from Shanghai Normal University in 2021. He then enrolled at Concordia University to begin his MSc in Computer Science. He fast-tracked to the PhD program in 2023. Currently, he is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. His research focuses on Software Engineering, Mining Software Repositories, and Software Maintenance. More information at: https://zeyang919.github.io/.

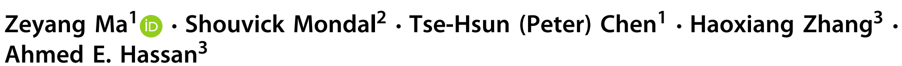
Shouvick Mondal leads the Software Engineering and Testing Lab. at the Indian Institute of Technology Gandhinagar (IITGN). His research activities focus on the improvement of existing and the development of new scalable and performant software analysis methodologies to ensure construction of high-quality and trustworthy software systems. His work has been published and presented in conferences and journals such as ASE, ICSME, ICST, JSS, and TSE. He regularly serves as a Program Committee member in the Artifact Evaluation track of conferences such as ISSTA, and PPoPP. He also serves as a journal reviewer for TSE, JSS, and IST. More information at: https://sites.google.com/view/shouvick.

Tse-Hsun (Peter) Chen is an Associate Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years and served as a research consultant at Ericsson for almost two years. His work has received several prestigious awards and has been published in flagship conferences and journals such as ICSE, FSE, ASE, TSE, and TOSEM. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ICSE, FSE, ASE, ICSME, SANER, and MSR, and he is a regular reviewer for software engineering journals such as EMSE and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.

Haoxiang Zhang is a research fellow at the Software Analysis and Intelligence Lab (SAIL), Queen's University, Canada. His research interests include empirical software engineering, mining software repositories, and intelligent software analytics. He received a PhD in Computer Science from Queen's University, Canada. He received a PhD in Physics and MSc in Electrical Engineering from Lehigh University, and obtained his BSc in Physics from the University of Science and Technology of China. Contact haoxiang.zhang@acm.org. More information at: https://haoxianghz.gitlab.io/homepage/.

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. More information at: http://sail.cs.queensu.ca.

Authors and Affiliations



Zeyang Ma[1] · Shouvick Mondal[2] · Tse-Hsun (Peter) Chen[1] · Haoxiang Zhang[3] · Ahmed E. Hassan[3]

Shouvick Mondal
shouvick.mondal@iitgn.ac.in

Tse-Hsun (Peter) Chen
peterc@encs.concordia.ca

Haoxiang Zhang
haoxiang.zhang@acm.org

Ahmed E. Hassan
ahmed@cs.queensu.ca

[1] The Software Performance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada

[2] Computer Science and Engineering, Indian Institute of Technology Gandhinagar, Gujarat, India

[3] Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, ON, Canada