A Literature Survey on Pitfalls of Open-Source Dependency Management in Enterprise

Andrey Kharitonov, Amro Abdalla, Abdulrahman Nahhas, Daniel Gunnar Staegemann, Christian Haertel, Christian Daase and Klaus Turowski
Otto von Guericke University Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany

Keywords: Open-Source Dependencies, Enterprise Software Development, Dependency Management, Software Dependencies.

Abstract: Open-Source dependencies are an integral part of the modern enterprise software development process for numerous technology stacks. Often, these dependencies are distributed through public repositories located outside of the secure corporate environment, which introduces numerous challenges in ensuring the security, compliance, and maintainability of the developed software. In this work, we conduct a systematic literature review focused on the pitfalls of relying on open-source dependencies. We discovered 23 relevant publications between 2016 and the beginning of 2024 pointing out that supply chain attacks, outdated or abandoned dependencies, licensing issues, security vulnerabilities, as well as reliance on trivial packages and complex dependency trees are mentioned in the analyzed literature as significant challenges. Among the ways to tackle these, it is commonly suggested in the literature to use scanning tools to ensure security, consciously select the used dependencies, document, and keep track of the open-source dependencies used in software projects. Maintaining up-to-date dependencies and actively contributing to the development of the open-source project is encouraged.

1 INTRODUCTION

In recent years, open-source libraries have become an integral part of many software development projects. Reliance on open-source packages for the implementation of many common and complex software features allows to alleviate a lot of workload on software engineers. These packages are commonly referred to as dependencies, as the software products where these are applied depend on these for their implementation of various critical functions. The use of such open source packages also offers numerous benefits such as maintenance cost savings, community support, and flexibility (Lakhan and Jhunjhunwala, 2008; Simon, 2005; Fitzgerald, 2006). Companies benefit from open-source software as it leads to better security and higher software quality. According to Red Hat's industry report done in 2022 (Haff, 2022), 52% of the surveyed companies rely on open-source software for application development. It also notes that the main advantages of open-source software usage for companies are flexibility and innovation access. Furthermore, the publicly available code used as part of the developed software product is often referred to as a dependency of the specific software project.

However, the use of open-source software dependencies introduces risks and challenges that must be carefully managed, particularly when it comes to managing software dependencies (Decan et al., 2018). The success of software development projects often hinges on the effective management of open-source dependencies, particularly in enterprise environments where the security risk is high (Gustavsson, 2020; Kikas et al., 2017).

Managing open-source dependencies is complex and challenging for software developers and project managers (Garrett et al., 2019). It is important to know which potential pitfalls exist in relying on open-source dependencies in software development. In this context, the research question of this paper is: *What are the common pitfalls and solutions for open-source dependency management in enterprise environments?*

To answer this question, we conduct a systematic literature survey. We search for the discussed risks and challenges associated with open-source dependency management, with a special interest in issues

Kharitonov, A., Abdalla, A., Nahhas, A., Staegemann, D., Haertel, C., Daase, C. and Turowski, K.
A Literature Survey on Pitfalls of Open-Source Dependency Management in Enterprise.
DOI: 10.5220/0012710800003753

In *Proceedings of the 19th International Conference on Software Technologies (ICSOFT 2024)*, pages 15-22
ISBN: 978-989-758-706-1; ISSN: 2184-2833

relevant to enterprise software development environments.

Furthermore, we concentrate on the currently popular programming languages so that we avoid extensive discussion of obscure or highly specialized technologies that are typically not used in enterprise software development. The survey covers recent literature from 2016 to the beginning of 2024 and discusses the finding's impact on enterprise environments. With this review, we aim to provide software developers, researchers, and project managers with valuable insights into best practices for managing open-source dependencies in enterprise environments.

Following, in section 2, we present the description of the literature search and selection process. After this, in section 3, we present the pitfalls and mitigation measures discovered in the process of analyzing the selected literature. Then we present with a brief discussion of possible future work directions in section 4. Finally the paper conclusion is summarized in section 6.

2 LITERATURE SURVEY

We begin this work with a brief description of the literature search process. We also discuss specific criteria for selecting or excluding the discovered literature from consideration.

2.1 Literature Search

To identify relevant publications for this literature survey, a systematic search was conducted on the two search terms presented below.

Search Term 1. ("managing" OR "management" OR "manage") AND ("open-source" OR "open source") AND ("dependency" OR "package" OR "packages")

Search Term 2. ("detecting" OR "finding" OR "detect" OR "find" OR "discover") AND ("suspicious" OR "malicious" OR "pitfalls" OR "vulnerabilities" OR "vulnerability" OR "issues") AND ("package" OR "packages" OR "dependency") AND ("npm" OR "maven" OR "pypi" OR "NuGet" OR "Packagist" OR "pkg.go")

Search term 1 aimed at the broad dependency management overview as discussed in the literature. Search term 2 is a second step in the literature review and is aimed at a set of challenges specific to the most popular package management systems that are discussed in the literature.

The first search term was used in three major academic literature indexes: Scopus, ScienceDirect, IEEE. The second search term was used only in Scopus and IEEE, as the search engine in ScienceDirect does not allow such complex search strings at the time of writing. The search was limited to the title, abstract, and keywords of articles and conference publications published between 2016 and 2023. This time frame is chosen to ensure that our review includes the most recent studies on open-source dependency management. Year 2016 was specifically selected, because that is the year when the notorious left-pad incident took place (Decan et al., 2018). This incident is said to have "almost broken the internet." The search is limited to conference publications and journal articles to ensure the studies are peer-reviewed. Only English-language publications are considered. After applying the above-mentioned filter criteria, a total number of 715 was found, as shown in Table 1.

Table 1: Search term results.

Source	Search Term 1	Search Term 2
Scopus	648	34
ScienceDirect	97	Not applicable
IEEE	91	31
Duplicates	186	
Sum	715	

2.2 Inclusion Criteria

For our literature survey process, we defined the following explicit inclusion criteria. First, the analyzed literature must either be general or focused on open-source dependencies, distributed through public package management platforms or repositories, for software written in one of the top 15 programming languages according to the TIOBE index January 2024. These languages include Python, C++, Java, C#, JavaScript, PHP, and Go.

Secondly, we are specifically interested in the potential technical or legal pitfalls of integrating open-source dependencies in software projects. These aspects include issues that compromise security, maintainability, or licensing aspects of software use and distribution.

Lastly, we are focusing on suggestions and solutions to common pitfalls that are relevant to enterprise software developers in managing their open-source dependencies.

2.3 Exclusion Criteria

Additionally, we define a number of clear exclusion criteria to maintain the specific focus of this survey.

Firstly, we do not consider any open-source dependencies outside of these that are directly integrated with the code base of a developed software project. Specifically, any open-source dependencies on the operating or environment level are not considered in this survey. Similarly, container images are excluded from this review. Although there are important parts of the open-source ecosystem that can be used for managing dependencies, this literature review will not include them to narrow the scope of the open-source dependencies to those that integrate with the codebase itself.

Secondly, we do not consider the code quality or functionality assessment of the open-source dependencies. That means that code smells, code quality metrics, test coverage, and similar criteria are outside of this work's scope.

Furthermore, we do not consider any suggestions to open-source repositories maintainers. Management of mirrors for popular package management platforms, as well as private package management distribution best practices, are beyond the scope of this work.

2.4 Selection Process

As shown in Figure 1, according to the mentioned inclusion and exclusion mentioned in subsection 2.2 and subsection 2.3, a total of 715 papers' titles were checked, and based on how the title matched those criteria, 76 papers were selected. Then, the abstract sections for those selected papers were checked, and according to that, 30 papers were chosen. Finally, after reading the 30 selected papers, 23 were selected for the thorough literature review.

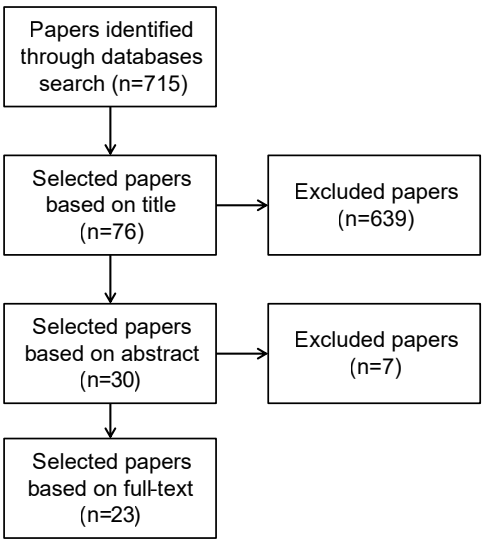

Figure 1: Literature review selection process.

3 ANALYSIS

In this section, we list all the possible pitfalls in open-source dependency management mentioned in the papers selected for the literature review and explain what each one means. Additionally, we list and explain the recommended solutions to avoid these possible pitfalls that were discovered in the literature.

3.1 Possible Pitfalls

Within our literature survey, we discovered a number of possible challenges and pitfalls of relying on open-source dependencies in software development. These are summarized in Table 2 and briefly discussed in this section.

3.1.1 Breaking Functionality - Unmaintained Packages

Open-source dependencies often comprise the building blocks of fundamental software (Gustavsson, 2020) layers (e.g., network communication, encryption, data management). These layers might change in the future, causing breaking functionalities in the packages that are not up to date with that change. Furthermore, old unmaintained packages might contain known but unfixed vulnerabilities (Zimmermann et al., 2019).

It is specifically an issue when the project on which an open-source dependency is based is abandoned while being a fundamental component of the software product. Such unmaintained or abandoned open-source dependencies result in an increased workload on the software developers (Miller et al., 2023) to resolve and replace the outdated packages.

3.1.2 Delayed Dependencies Update - Technical Lag

When software dependencies are not being updated for an extended period of time, the process of changing between two major versions of these might become time and resource-intensive (Gustavsson, 2020).

This also leads to a so-called "technical debt". It is a situation when developers ignore patch updates that provide new functionalities that the project misses (Kaplan and Qian, 2021). Also, according to (Kabir et al., 2022), dependencies are being updated in their study weeks or months after the new release, and according to (Prana et al., 2021), one of the factors that affect the persistence of dependency vulnerabilities is

Table 2: Possible pitfalls in open-source dependency management.

Possible Pitfalls	
Breaking or Unmaintained packages	(Zimmermann et al., 2019; Gustavsson, 2020; Miller et al., 2023)
Delayed dependencies update	(Decan et al., 2018; Gustavsson, 2020); (Prana et al., 2021; Kaplan and Qian, 2021; Kabir et al., 2022)
Interoperability problems	(Gustavsson, 2020; Wang et al., 2023)
License noncompliance	(Bauer et al., 2020; Xu et al., 2023)
Security vulnerabilities	(Pashchenko et al., 2018; Gustavsson, 2020; Kluban et al., 2022)
Phantom artifacts	(Imtiaz and Williams, 2023)
Supply chain attack (SCA)	(Zimmermann et al., 2019; Kaplan and Qian, 2021); (Scalco et al., 2022; Alfadel et al., 2023); (Guo et al., 2023)
Strongly connected components (SCC)	(Kaplan and Qian, 2021; Setó-Rey et al., 2023)
Trivial packages	(Garrett et al., 2019; Kaplan and Qian, 2021)

how fast the vulnerable package is updated to a non-vulnerable version.

3.1.3 Interoperability Problems

Modern software systems might consist of a number of components that are developed semi-independently. These components might rely on the same dependency packages but different versions of them. Such problems might arise indirectly through sub-dependencies (Wang et al., 2023).

This situation might cause issues during the deployment stage of the project (Gustavsson, 2020) when all of the components are finally brought together into the same environment.

3.1.4 License Noncompliance

License noncompliance (Bauer et al., 2020), where developers in an enterprise environment use a package they are not entitled to use, can cause legal and other risks of using open-source dependencies in commercial software.

Furthermore, it's important to remember that open-source dependencies might depend on other open-source projects themselves. These other projects have their own licensing limitations. There is no guarantee that the maintainers of an open-source project meticulously monitor the composition and licensing of the packages they use in their own work. This might lead to unintended licensing violation (Xu et al., 2023) in the final software product.

3.1.5 Security Vulnerabilities

According to (Kluban et al., 2022), the most common vulnerabilities in Github projects npm packages are, Cross-Site Scripting, code injection, prototype pollution, regular expression denial of service(ReDoS), fuzzy hashing and cryptography. The scale of this issue can not be underestimated, as it was shown (Pashchenko et al., 2018) in many software open source projects, a large number of dependencies may contain vulnerabilities.

As mentioned further in subsubsection 3.1.7, security vulnerabilities might be direct and indirect (Gustavsson, 2020). This means every dependency is only as secure as its own sub-dependencies.

3.1.6 Supply Chain Attack

There are a few types of supply chain attacks (SCA), all of which aim at the software developers inadvertently injecting malicious code into their own codebase through the use of package management tools. Alternatively, they expose their own environment to a malicious routine (Scalco et al., 2022) during dependency installation.

In literature, squatting attacks with two distinct types are mentioned (Kaplan and Qian, 2021; Alfadel et al., 2023; Scalco et al., 2022; Scalco et al., 2022): typosquatting and combosquatting. The first is when the attacker creates a malicious package with a typo mistake in its name, which is similar to a popular package name, assuming the developer might make a typo mistake when installing the package. Combosquatting is similar to typosquatting, but instead of making a typo, it changes the order of words in the package name.

However, SCAs aren't limited to simply relying on the chance that a developer makes a mistake while installing the name of a package. Normally, secure packages can be exposed to this type of attack through malicious code injected into them (Zimmermann et al., 2019).

It is shown in the literature that mirrors of the popular package management platforms might be more susceptible to (Guo et al., 2023) presence of malicious packages in the distribution.

3.1.7 Strongly Connected Components

According to the analysis made by (Setó-Rey et al., 2023), Strongly connected components (SCC) are when packages are transitively dependent on others, propagating code defects and vulnerabilities. The larger the scale of the SCC, the effect of the vulnerability is higher(Kaplan and Qian, 2021).

3.1.8 Trivial Packages

It is noted in the literature that it is not uncommon for developers to favor small packages for even simple tasks (Garrett et al., 2019). The so-called ”left-pad incident” (Kaplan and Qian, 2021) that ”almost broke the internet” in 2016 was caused by an 11-line package.

An earlier survey (Abdalkareem et al., 2017) points, however, that most developers share the opinion that trivial packages introduce unnecessary dependency tree complexity. This complexity makes the update of the packages unnecessarily time-consuming and might, in fact, open opportunities for the introduction of vulnerabilities or malicious code (Garrett et al., 2019).

3.1.9 Phantom Artifacts

Authors of a recent study analyzed open-source dependencies (Imtiaz and Williams, 2023) and, in more than 20% of cases, discovered the distributed packages of these to contain files that are not present in the repositories of these dependencies. These phantom artifacts might be simply erroneous but might, in fact, point to a compromised dependency in the package management system.

3.2 Mitigation Measures

While within our survey, we did not discover a specific all-encompassing methodology that would always ensure a safe and secure way to manage open-source dependency, a number of measures that might be employed to mitigate the challenges mentioned in subsection 3.1 were discovered. These are summarized in Table 3 and briefly elaborated upon in this section.

3.2.1 Conscious Decision

As mentioned in subsubsection 3.1.8, trivial packages are one of the common pitfalls in open-source dependency management. In the literature, it is recommended (Gustavsson, 2020) to limit the open-source dependencies to deliberate decisions, as the cost of maintaining this dependency might be higher than letting the team write the required function themselves.

One of the important factors when choosing the package is that the project is active, and there are often new releases when bugs are discovered. Three processes were recommended when choosing a new package, firstly is to create a group that decides on using common packages among the different teams and projects, second is to develop selection criteria to help in choosing the right package, automate the process of detecting open-source dependencies.

3.2.2 Contribute Upstream - Support Open-Source Projects

When the use of an open-source dependency is unavoidable, it’s recommended (Gustavsson, 2020; Miller et al., 2023) to engage in the maintenance of the dependency explicitly. Upstream contributions include bug reporting or fixes, feature requests or code, support other users and participation in their community forum, and finally, documentation and translation. Sustainability is an important factor when choosing a dependency, to improve the sustainability of the open-source package, the company may support the maintainers of the package financially, sign a maintenance contract, or sponsor the infrastructure.

3.2.3 Separate Dev and Prod Configurations

According to (Latendresse et al., 2022), dependencies should be treated differently based on the deployment configuration. Only the dependencies that are used in the production code should be deployed with the code, and the development dependencies should not be installed in production, as the security vulnerabilities in the production code dependencies should be prioritized more than the development dependencies.

3.2.4 Following the Package Manager Best Practices

According to (Kikas et al., 2017), experts suggest best practices when using npm packages. Those best practices are scanning and removing the vulnerabilities and unused/duplicated packages using commands provided by the package manager and enforcing the lock file to pin the dependencies versions.

However, according to (Zimmermann et al., 2019), this defense is not enough as those commands do not take into consideration the transitive packages, are limited to known vulnerabilities, and are insufficient against malware attacks.

Table 3: Measures to mitigate the possible pitfalls in open-source dependency management.

Proposed measures	
Conscious decisions	(Gustavsson, 2020)
Contribute upstream - Support open-source projects	(Gustavsson, 2020; Miller et al., 2023)
Separate dev and prod configurations	(Latendresse et al., 2022)
Following the package manager's best practices	(Kikas et al., 2017; Zimmermann et al., 2019)
Localize dependency use	(Miller et al., 2023)
Document and maintain a dependency list	(Gustavsson, 2020; Miller et al., 2023)
Use of scanning tools	(Carlson et al., 2019; Gustavsson, 2020); (Kaplan and Qian, 2021; Zerouali et al., 2022); (Liu et al., 2022)
Update dependencies continuously	(Gustavsson, 2020; Prana et al., 2021)
Verify integrity	(Gustavsson, 2020; Imtiaz and Williams, 2023)

3.2.5 Document and Maintain a Dependency List

According to (Gustavsson, 2020), maintaining a dependency list has many advantages, like ensuring license compliance, helping in maintaining the technical dependency, and understanding the functionality of each dependency.

The dependency list should have pointers to the original sources, and organizations should also consider maintaining the dependency list part of the bill of materials over open-source components (Gustavsson, 2020). Furthermore, this practice can also facilitate the monitoring (Miller et al., 2023) of the development or lack thereof for all of the used dependencies. This, in turn, might allow for a faster reaction to breaking changes or abandonment of the dependencies.

Having a properly documented list of dependencies also contributes to the retention of knowledge on the project code base. Extensive documentation removes the risk of potential loss of knowledge on which dependency is used and why.

3.2.6 Use of Scanning Tools

There are tools (Gustavsson, 2020) that scan the dependencies for known vulnerabilities, which helps in managing the risks when it is included in the development process. In the work done by (Carlson et al., 2019), they used the scanning tool Snyk [1], in order to analyze 600 GitHub projects searching for vulnerable dependencies in these projects. Furthermore, (Garrett et al., 2019) developed a machine learning-based tool that has an anomaly detection approach to detect malicious dependency updates that contain vulnerabilities. It is noted in the literature, however, that tools used for scanning code for the presence of malicious logic might be susceptible to an excessive number of false-positive detections (Guo et al., 2023). Therefore, these tools should be used with care.

3.2.7 Update Dependencies Continuously

According to (Gustavsson, 2020; Prana et al., 2021), dependencies should be updated regularly to avoid the most common pitfalls. The automated test should also be performed after each update as part of a continuous integration pipeline. In addition to that, a process to verify that system functionalities are still working. However, the scale of the benefit from maintaining updated dependencies may vary depending on the programming language (Prana et al., 2021).

3.2.8 Verify Integrity

As obvious from Table 2, supply chain attacks are one of the most common pitfalls. Avoiding these can be done by verifying the integrity of the package, there are two recommended ways, first is to verify a hash of the downloaded software with the has in the source and the second is verify the digital signature of the downloaded package (Gustavsson, 2020).

The importance of making sure that a dependency distributed from a package management platform does not contain any files or additions that are not intended or present in the code of the dependency. In the literature (Imtiaz and Williams, 2023), similarly to what was already mentioned in subsubsection 3.2.6, it is proposed to use automated solutions for validating dependencies to mitigate such potential issues.

3.2.9 Localize Dependency Use

It is noted in the literature (Miller et al., 2023) that one of the preventative strategies for issues like unmain-

[1] Snyk's website: https://snyk.io [Last Visited: 27.04.2024]

tained or abandoned dependencies or other problems is an attempt to localize the use of open-source dependencies.

This means that an abstraction layer can be added in the program using such a dependency so that in case of issues, the dependency might be replaced with as little effort as possible. A situation where a single open-source package is linked throughout the entire application must be avoided.

4 FUTURE WORK

The use of scanning tools to ensure the integrity and security of open-source dependencies is a promising area of investigation. Specifically, it catalogs the existing approaches, commercial and algorithmic, while comparing capabilities. As noted in subsubsection 3.2.6, it is a suggested strategy for mitigating various issues associated with reliance on open-source dependencies but not without challenges itself.

Furthermore, we believe it is a promising area of research to investigate the effects on software engineering processes and its costs arising from the varying quality of overall program code, not just security vulnerabilities, in open-source dependencies (Imtiaz and Williams, 2023; Go et al., 2023).

Additionally, extending this survey to include the possible pitfalls of relying on publicly available container images is another topic that we plan to tackle in the future.

5 LIMITATIONS

It is worth noting that the basis of this work is the systematic literature review of the published scientific literature on the common challenges of relying on open-source dependencies in enterprise software projects. Therefore, this work is not focused on specific technical solutions proposed by the industry.

It is, nevertheless, important to mention that various projects and tools exist to support developers in the mitigation of risks mentioned in section 3. For example, various projects exist (Williams, 2022) to mitigate supply chain attacks mentioned in subsubsection 3.1.6. Another example is a project (Nocera et al., 2023) that proposes a standardized software bill of materials to assist developers in transparently documenting the list of dependencies, as mentioned in subsubsection 3.2.5. However, a recent study (Nocera et al., 2023) suggests a currently low degree of adoption of this standard but with a positive increasing trend.

6 CONCLUSION

In this literature survey, we investigate the common pitfalls in open-source dependency management and solutions to avoid them. The most frequently mentioned potential issues in the literature are outdated dependencies and supply chain attacks on dependency management platforms.

The most mentioned mitigation strategies involve the use of a scanning tool to help in finding vulnerabilities. In addition to that, other solutions like updating dependencies continuously and following the best practices provided by the package manager were recommended by more than one publication.

It is noted that every dependency should be added to the software project consciously, with a specific reason, and it must be documented.

REFERENCES

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA. Association for Computing Machinery.

Alfadel, M., Costa, D. E., and Shihab, E. (2023). Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 28(3):59.

Bauer, A., Harutyunyan, N., Riehle, D., and Schwarz, G.-D. (2020). Challenges of tracking and documenting open source dependencies in products: A case study. In *Open Source Systems: 16th IFIP WG 2.13 International Conference, OSS 2020, Innopolis, Russia, May 12–14, 2020, Proceedings 16*, pages 25–35. Springer.

Carlson, B., Leach, K., Marinov, D., Nagappan, M., and Prakash, A. (2019). Open source vulnerability notification. In *Open Source Systems: 15th IFIP WG 2.13 International Conference, OSS 2019, Montreal, QC, Canada, May 26–27, 2019, Proceedings 15*, pages 12–23. Springer.

Decan, A., Mens, T., and 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*, pages 181–191.

Fitzgerald, B. (2006). The transformation of open source software. *MIS quarterly*, pages 587–598.

Garrett, K., Ferreira, G., Jia, L., Sunshine, J., and Kästner, C. (2019). Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE.

Go, K. R., Soundarapandian, S., Mitra, A., Vidoni, M., and Ferreyra, N. E. D. (2023). Simple stupid insecure

practices and github's code search: A looming threat? *Journal of Systems and Software*, 202:111698.

Guo, W., Xu, Z., Liu, C., Huang, C., Fang, Y., and Liu, Y. (2023). An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177.

Gustavsson, T. (2020). Managing the open source dependency. *Computer*, 53(2):83–87.

Haff, G. (2022). The State of Enterprise Open Source: A Red Hat report. White paper, Red Hat.

Imtiaz, N. and Williams, L. (2023). Are your dependencies code reviewed?: Measuring code review coverage in dependency updates. *IEEE Transactions on Software Engineering*, 49(11):4932–4945.

Kabir, M. M. A., Wang, Y., Yao, D., and Meng, N. (2022). How do developers follow security-relevant best practices when using npm packages? In *2022 IEEE Secure Development Conference (SecDev)*, pages 77–83. IEEE.

Kaplan, B. and Qian, J. (2021). A survey on common threats in npm and pypi registries. In *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*, pages 132–156. Springer.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE.

Kluban, M., Mannan, M., and Youssef, A. (2022). On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 917–930.

Lakhan, S. E. and Jhunjhunwala, K. (2008). Open source software in education. *Educause Quarterly*, 31(2):32.

Latendresse, J., Mujahid, S., Costa, D. E., and Shihab, E. (2022). Not all dependencies are equal: An empirical study on production dependencies in npm. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., and Peng, X. (2022). Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 672–684.

Miller, C., Kästner, C., and Vasilescu, B. (2023). "we feel like we're winging it:" a study on navigating open-source dependency abandonment. page 1281 – 1293. Cited by: 2; All Open Access, Hybrid Gold Open Access.

Nocera, S., Romano, S., Penta, M. D., Francese, R., and Scanniello, G. (2023). Software bill of materials adoption: A mining study from github. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 39–49.

Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., and Massacci, F. (2018). Vulnerable open source dependencies: counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA. Association for Computing Machinery.

Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., and Lo, D. (2021). Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26:1–34.

Scalco, S., Paramitha, R., Vu, D.-L., and Massacci, F. (2022). On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–8.

Setó-Rey, D., Santos-Martín, J. I., and López-Nozal, C. (2023). Vulnerability of package dependency networks. *IEEE Transactions on Network Science and Engineering*.

Simon, K. D. (2005). The value of open standards and open-source software in government environments. *IBM Systems Journal*, 44(2):227–238.

Wang, C., Wu, R., Song, H., Shu, J., and Li, G. (2023). smartpip: A smart approach to resolving python dependency conflict issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

Williams, L. (2022). Trusting trust: Humans in the software supply chain loop. *IEEE Security & Privacy*, 20(5):7–10.

Xu, W., He, H., Gao, K., and Zhou, M. (2023). Understanding and remediating open-source license incompatibilities in the pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–190.

Zerouali, A., Mens, T., Decan, A., and De Roover, C. (2022). On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 27(5):107.

Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. (2019). Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX security symposium*, volume 17.