12-2023

Lessons from the long tail: Analysing unsafe dependency updates across software ecosystems

Supatsara WATTANAKRIENGKRAI

Raula KULA

Christoph TREUDE
Singapore Management University, ctreude@smu.edu.sg

Kenichi MATSUMOTO

Citation

WATTANAKRIENGKRAI, Supatsara; KULA, Raula; TREUDE, Christoph; and MATSUMOTO, Kenichi. Lessons from the long tail: Analysing unsafe dependency updates across software ecosystems. (2023). *ESEC/FSE '23: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, December 3-9*. 2077-2081.
Available at: https://ink.library.smu.edu.sg/sis_research/8903

Lessons from the Long Tail: Analysing Unsafe Dependency Updates across Software Ecosystems

Supatsara Wattanakriengkrai
Nara Institue of Science and Technology
Nara, Japan
wattanakri.supatsara.ws3@is.naist.jp

Raula Gaikovina Kula
Nara Institue of Science and Technology
Nara, Japan
raula-k@is.naist.jp

Christoph Treude
University of Melbourne
Melbourne, Australia
christoph.treude@unimelb.edu.au

Kenichi Matsumoto
Nara Institue of Science and Technology
Nara, Japan
matumoto@is.naist.jp

ABSTRACT

A risk in adopting third-party dependencies into an application is their potential to serve as a doorway for malicious code to be injected (most often unknowingly). While many initiatives from both industry and research communities focus on the most critical dependencies (i.e., those most depended upon within the ecosystem), little is known about whether the rest of the ecosystem suffers the same fate. Our vision is to promote and establish safer practises throughout the ecosystem. To motivate our vision, in this paper, we present preliminary data based on three representative samples from a population of 88,416 pull requests (PRs) and identify unsafe dependency updates (i.e., any pull request that risks being unsafe during runtime), which clearly shows that unsafe dependency updates are not limited to highly impactful libraries. To draw attention to the long tail, we propose a research agenda comprising six key research questions that further explore how to safeguard against these unsafe activities. This includes developing best practises to address unsafe dependency updates not only in top-tier libraries but throughout the entire ecosystem.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Supply Chain, Libraries, Software Ecosystems

ACM Reference Format:
Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Christoph Treude, and Kenichi Matsumoto. 2023. Lessons from the Long Tail: Analysing Unsafe Dependency Updates across Software Ecosystems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613086

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613086

1 INTRODUCTION

Widespread adoption and use of third-party library dependencies in applications is evident by the massive scale of software ecosystems (e.g., NPM for JavaScript, Maven for Java, PyPI for Python). For example, the NPM ecosystem supports more than 2.42 million JavaScript libraries[1] as of 2022 [8]. Libraries in these ecosystems form complex dependency relationships in which they depend on each other for functionality. A side effect of this heavy reliance on such ecosystems is the potential for vulnerabilities in libraries to spread. One such example is the Log4Shell vulnerability, which potentially left a large number of systems, including Fortune 500 companies [15], open to malicious attacks. Furthermore, in 2021 there was a 650% year-on-year growth in security attacks by exploiting Open Source Software supply chains [14].

In response to recent hijacking of widely used libraries, such as `ua-parser-js` [6], `coa` [4], and `rc` [5], GitHub has intensified its efforts to enhance the security of these highly depended-upon libraries. Other industry efforts include mandating two-factor authentication (2FA) for high-impact libraries, such as those with more than one million weekly downloads or 500 dependents [16]. Efforts such as the Alpha and Omega Project specifically aim to help the most critical open source projects improve their security postures [10]. Combined with the OpenSSF (Open Source Security Foundation), they target and select projects based on the work of the OpenSSF Securing Critical Projects Working Group using various techniques, such as expert opinions, data analysis, and the OpenSSF Criticality Score to identify the most critical open source software.

In this paper, we argue that identification and remedy of supply chain attacks should not be limited to high-impact libraries, as insecure practises (e.g., unsafe dependency updates) might span across the entire ecosystem. We define unsafe dependency updates as any pull request (PR) that risks being unsafe during runtime. The term *"unsafe"* originates from programming languages [20], where the compiler is explicitly instructed *"trust the code being executed without question"*. Our vision entails the implementation of protective measures across all tiers of the ecosystem. We collected 88,416 PRs from 1,500 curated libraries from three tiers of NPM libraries based on their dependence. Using features proposed in previous work, we flag and analyse unsafe dependency updates.

[1] We use the term libraries instead of packages for consistency.

Table 1: Collected dataset of update related PRs (with changes to `package.json` and/or `.js files`)

Tiers	# dependents		# PRs	# update related PRs	update related PRs per lib		
	Max	Min			Mean	Median	SD
Top-500	850,362	46,221	40,941	29,283	58.56	10.0	345.90
Middle-500	1,000	572	39,341	28,088	56.17	14.0	116.86
Bottom-500	1	1	8,134	6,458	12.91	2.0	71.7
Total			**88,416**	**63,829**			

Our preliminary results suggest that developer practises that occur along the long tail of libraries are equally important as those in high-impact libraries. This finding suggests that developer practises should be re-evaluated to reduce the prevalence of unsafe dependency updates and to understand the reasons behind developers’ choices to employ these unsafe practises.

2 PRELIMINARY ANALYSIS

Our preliminary study covers 1,500 JavaScript npm libraries.

2.1 Detecting unsafe dependency updates

Existing defences against supply chain attacks are often deemed impractical in realistic settings due to their high computational costs and scaling issues. These include carefully reviewing all updates to verify any activity (e.g., the execution of untrusted external sources) related to dependencies, thus hardening the package infrastructure (e.g., transport security, two-factor authentication), program analysis, and anomaly detection [9, 24, 27, 29]. Garrett et al. [24] suggested a lightweight method to detect potential supply chain attacks. The authors argue that their features are lightweight and do not require code compilation, and refer to them as unsafe:

(1) *adding new scripts* - As highlighted in the documentation [7], these scripts allow new scripts to be added to the configuration files, thus running the risk of introducing code or scripts from unknown sources that are different from the source code.
(2) *accessing http, http2, https* - Send and receive Hypertext Transfer Protocol (HTTP) requests. As mentioned by Garrett et al. [24], there is a risk that unknown information could be introduced into the application.
(3) *executing fs()* - Create, read, and write to the file system [2]. This poses the potential to erase or modify the local file system.
(4) *executing net module* - Based on the documentation [3], this module allows functions to open, listen, or write to stream-based TCP or IPC sockets. Exploits may enable the introduction of information into the application.
(5) *executing eval()* - This is a module that can execute JavaScript code represented as a string [1]. This practise is unsafe because it allows introduced sources to execute at runtime.
(6) *executing require()* - The require command [19] allows modules that exist in separate files to be run at runtime. This can be risky if malicious modules are imported and executed.

Various solutions have been proposed to protect against unsafe dependency updates based on these metrics. This includes lightweight permission [23], machine learning techniques [30], and has also led to other empirical studies in the area [22]. **In our work**, we use the six features proposed by Garrett et al. [24] to detect prevalence in three tiers of NPM libraries.

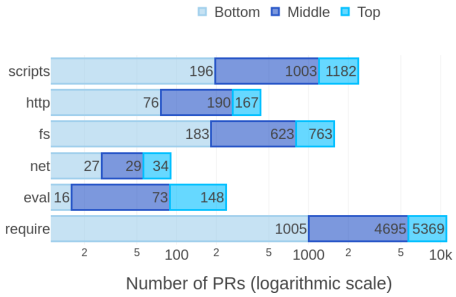

Figure 1: Frequency of unsafe dependency updates

2.2 Library tiers

Following prior work [24], we identify a PR as an *update-related PR* if it involves changes to JavaScript files (`.js files`) and/or modifications to the `package.json` configuration file. Note that the package.json file is used to record changes in dependencies. We decided on a purposive sampling approach, taking an equal sample of libraries to represent the different populations within the ecosystem. Our classifications are as follows:

- **Top-500** - The first tier of libraries are the most depended-upon libraries (outliers) within the NPM ecosystem. They are critical to the security supply chain. We ranked all the dependencies and selected the top 500 most depended-upon libraries.
- **Middle-500** - The second tier of libraries is still highly depended-upon by the NPM ecosystem. According to GitHub, a high-impact library is defined as a library with more than one million weekly downloads or 500 dependents. To create this sample, we selected libraries that had a number of dependents between 500 and 1,000.
- **Bottom-500** - The final tier of libraries includes those with the least number of dependents. Libraries in this tier are the least depended upon, and thus we select libraries with at least one dependent.

Using previous work [32] as a starting point, we analysed 107,242 NPM libraries, which are then cross-referenced with the library dataset from libraries.io [8] used in previous studies [21, 25, 35]. We collected 88,416 PRs from 1,500 sampled libraries based on their dependence as of March 2023.

Table 1 shows the details of the libraries selected for each tier. Filtering for update-related PRs (including package.json and/or .js files), we obtained 63,829 related PRs. To flag unsafe dependency updates, we employ the six features identified earlier, using a simple regular expression to detect these features in lines added to PR code commits. As shown in Figure 1, when applying the six features to detect unsafe dependency updates, we find that these updates are more likely to include the commands `require` and `new scripts`. In contrast, the `net` and `eval` functions are less prevalent compared to the others. All scripts and data can be found at https://doi.org/10.5281/zenodo.6719258.

Table 2: Unsafe dependency updates for each tier

Tiers	# unsafe lib	# unsafe PRs	Mean	Median	SD
Top-500	405 (81%)	6,167	12.33	2.0	56.49
Middle-500	402 (80%)	5,212	10.42	3.0	27.87
Bottom-500	220 (44%)	1,086	2.17	0.0	12.58
Total	**1,027**	**12,465**			

Table 3: Unsafe dependency update outcomes per tier

		# unsafe PRs	
Top-500	merged PRs	4,333	70%
	closed PRs	1,508	24%
	opened PRs	326	6%
	Total	**6,167** **(21% of update related PRs)**	
Middle-500	merged PRs	3,704	71%
	closed PRs	1,242	24%
	opened PRs	266	5%
	Total	**5,212** **(19% of update related PRs)**	
Bottom-500	merged PRs	863	79%
	closed PRs	173	16%
	opened PRs	50	5%
	Total	**1,086** **(17% of update related PRs)**	

2.3 Prevalence

Table 2 shows that we flagged a total of 12,465 PRs belonging to 1,027 libraries, which were distributed among the different tiers. The evidence clearly shows that unsafe dependency updates are prevalent not only in the most impactful and dependent libraries, but across the other tiers within the ecosystem, including the Bottom-500 set of libraries. For example, at least 81% of the libraries in the Top-500 category have at least one unsafe dependency update. This is slightly higher than the Middle-500 (80%). Even Bottom-500 tier has almost 44%, with at least one unsafe dependency update.

2.4 Acceptance

We calculate the acceptance rate of unsafe dependency updates by collecting the outcomes of the PRs after the review team has received them. Typically, a PR is either accepted (i.e., merged) into the code base, closed without being merged into the code (i.e., closed) or still under review by the maintainers (i.e., opened). We compare the outcomes of the reviews of these unsafe dependency updates across the three tiers.

Table 3 shows the acceptance of these unsafe dependency updates by the three tiers. The results indicate that libraries in the least depended-upon tiers tend to accept and merge unsafe dependency updates. As shown in the table, all tiers (i.e., Top-500 at 70%, Middle-500 at 71%, and Bottom-500 at 79%) have most of their unsafe dependency updates merged into the codebase.

2.5 Code Change Differences

Since we employ the lightweight method from [24], it remains unconfirmed whether these unsafe dependency updates are indeed malicious. Therefore, we investigate the differences in both code and textual content to better categorise these unsafe dependency updates. We analyse both textual and committed file types to distinguish differences between the different tiers of unsafe dependency updates. We extract content from the titles and descriptions while identifying the types of file changes to determine the kind of changes that occurred. Using a taxonomy of code changes from a previous study [31], two of the authors applied a saturation sampling approach to obtain the following coding. The codes consist of a combination of keywords and the types of files modified in the PR. Noted that each keyword was removed non-textual symbols and punctuation, and applied lemmatization.

Table 4: The frequency of six unsafe dependency update types, comparing those labeled "require attention"

		Unsafe PRs with attention keywords		Unsafe PRs no attention keywords	
Tiers	# merged & closed PRs	# PRs	PR types (#)	# PRs	PR types (#)
Top-500	5,841	1,406	Feature (1,120) Test Cases (838) Bug (725) Doc (546) Refactoring (524) Other (53)	4,435	Feature (2,797) Bug (2,254) Doc (1,832) Test Cases (1,310) Refactoring (863) Other (425)
Middle-500	4,946	1,136	Feature (905) Bug (641) Test Cases (609) Doc (542) Refactoring (252) Other (48)	3,810	Feature (2,273) Bug (1,525) Doc (1,310) Test Cases (892) Refactoring (482) Other (438)
Bottom-500	1,036	149	Feature (90) Bug (53) Test Cases (53) Doc (52) Refactoring (42) Other (20)	887	Feature (402) Bug (283) Doc (278) Test Cases (172) Other (160) Refactoring (109)
Total	**11,823**	**2,691**		**9,132**	

- Feature - *keywords are:* integrate, add, feat, update, upgrade, support, dependency, feature, improve, version, automate, compatibility, bundle, improvement, bump
- Bug - *keywords are:* avoid, fix, resolve, close, bug, solve, solution, issue, fixing
- Test Cases - *keywords are:* test case, test, unit test, CI, continuous integration
- Refactoring - *keywords are:* remove, unnecessary, refactor, performance, optimise
- Documentation - *keywords are:* documentation, doc; *files include:* .md files
- Other - *files include:* .json only or anything that does not fall into the above categories

To assess the impact of a PR, we follow related work [28] to identify keywords that attract developer attention (such as 'attention', 'breaking', and 'performance').[2]

Table 4 shows a consistent frequency count and content of unsafe dependency updates that attract attention and those that do not. As shown, the number of unsafe dependency updates that require attention is relatively small across all tiers (i.e., 1,406 PRs for Top-500,

[2] The full list of the keywords is available at https://doi.org/10.5281/zenodo.6719258.

1,136 PRs for Middle-500, and 149 PRs for Bottom-500), as these updates indicate critical problems for a library (e.g., performance issues and breaking changes). Our results seem consistent across the tiers, with PRs classified as being new features and bugs being prevalent. Interestingly, changes that required the maintainers attention included test cases, while those that did not (no attention), did include documentation changes. We assume that test cases and documentation may indicate that the unsafe practises are justifiable, and intentional.

3 RESEARCH AGENDA

Our research agenda is based on six key research questions that includes developing best practises to address unsafe dependency updates throughout the entire ecosystem.

3.1 Placing Safeguards

The first analysis findings indicate that unsafe dependency updates are prevalent across all tiers within the ecosystem. Thus, our aim is to establish a close connection between unsafe practises and their alternatives, based on existing research that has explored the exploitability of code [26] and how practitioners are using OSS libraries in their code [33].

Safer Alternatives: Safer alternatives to these unsafe practises exist; but each has its own drawbacks that might not attract use. For example, an alternative to using the `require()` function is to instead use the `import()` function. However, the key limitation is that the import function must be defined at the top of a class, while the require function can be executed at any point in the code [18]. Similarly, for the `eval()` function, developers have documented the potential risks of using `eval()` [17]; developers should consider using alternatives such as JSON.parse(), Function(), or templating engines.

Adding Layers of Security: Another alternative is to add a layer of security by using appropriate HTTP security headers, which can help prevent some common attack vectors (for example, by using the helmet package). Other solutions include implementing external source security certificate controls. In terms of security access (i.e., unprotected access to files and directories on the Operating System), developers can design the program so that read/write access is restricted to certain files and directories only. Other options include sandboxing and the implementation of a permission system based on file and network access, as first studied by Ferreira et al. [23].

The first research question in our research agenda outlines the promises and perils of using alternative but safer update practises.

(1) What are the promises and pitfalls of using an alternative but safer implementation across the ecosystem?

3.2 Reasoning behind these Updates

As mentioned in the previous section, there are safer alternatives and methods to remedy these unsafe practises; however, it remains unknown why developers persist in implementing these unsafe practises.

(2) What are the reasons why developers (open source and industry) are likely to trust (and accept) these unsafe dependency updates into their code?

As the initial findings show that unsafe dependency updates are not limited to highly impactful libraries, it would be interesting to understand how the coding practises differ between the head and tail of the ecosystem. For example, maybe a change of practices at the tail may propagate best practices at the head or vice versa. As a community, the adoption and enforcement of practices may reap better results. Researchers can further explore how the levels of unsafety may be different between updates in the head and tail of the ecosystem. This is essential because not all of the levels of insecurity/unsafety are equally concerning. For example, some of the issues are just potential vulnerabilities that can hardly be exploitable in a realistic manner. Therefore, our next research question aims to suggest learning from the long tail.

(3) How does the practise differ between the head and tail of dependencies, and how can we characterise those?

Answering these questions will provide a more comprehensive understanding of why these unsafe dependency updates exist. Furthermore, we can complement and improve existing initiatives such as the criticality score [11] and the OSS scorecard [12], which are currently being investigated by Zahan et al. [34].

Automation for Safer Code: The second analysis findings suggest that these unsafe dependency updates are being accepted. Hence, developers are either unaware that these practises are unsafe or find them unavoidable due to time constraints and effort. Actionable implications for researchers include tool support, such as code suggestion tools (e.g., automatic code completion), and refactoring existing unsafe code to make it safer in the appropriate situations. Other tools include automatic detection of these unsafe dependency updates.

(4) How much effort is required to refactor unsafe code?

Utilising the Long Tail: The third analysis findings also show that unsafe dependency updates are feature-related and are accompanied by test cases or documentation, serving as a validation of trust that is consistent throughout the entire ecosystem. Since this practise is prevalent, both researchers and security experts can target a larger collection of data and perform sampling across the long tail.

(5) What practises (open source and industry) of unsafe dependency updates can be trusted?

(6) What kind of validation is needed to gain trust from unknown sources that are using unsafe dependency updates?

Answering these questions will provide us with actionable insights and also align with the OSSF goal of open source projects to adopt best practises when securing their code [13].

4 CONCLUSION

Studying the long tail of the software ecosystem demonstrates that unsafe practises exist in all tiers. Therefore, the key to securing supply chains is not only to focus on highly impactful libraries but also to take into account existing practises and explore how the community as a whole can work together to create a safer and more resilient OSS supply chain.

ACKNOWLEDGMENTS

Supported by JSPS KAKENHI Grant Numbers 20K19774, 23H03375 and 20H05706, and JST SICORP Grant Number JPMJSC2206.

REFERENCES

[1] 1998. eval() - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.

[2] 2009. File system | Node.js v20.0.0 Documentation. https://nodejs.org/api/fs.html#fs_file_system.

[3] 2009. Net | Node.js v20.0.0 Documentation. https://nodejs.org/api/net.html.

[4] 2011. veged/coa: Command-Option-Argument: Get more from defining your command line interface. https://github.com/veged/coa.

[5] 2012. dominictarr/rc: The non-configurable configuration loader for lazy people. https://github.com/dominictarr/rc.

[6] 2012. faisalman/ua-parser-js: UAParser.js - Detect Browser, Engine, OS, CPU, and Device type/model from User-Agent data. Supports browser & node.js environment. https://github.com/faisalman/ua-parser-js.

[7] 2014. About npm | npm Docs. https://docs.npmjs.com/about-npm.

[8] 2015. Libraries.io - The Open Source Discovery Service. https://libraries.io/.

[9] 2019. The complete package: Everything you need to know about npm security | The Daily Swig. https://portswigger.net/daily-swig/the-complete-package-everything-you-need-to-know-about-npm-security.

[10] 2020. Alpha-Omega - Open Source Security Foundation. https://openssf.org/community/alpha-omega/.

[11] 2020. GitHub - ossf/criticality_score: Gives criticality score for an open source project. https://github.com/ossf/criticality_score.

[12] 2020. GitHub - ossf/scorecard: OpenSSF Scorecard - Security health metrics for Open Source. https://github.com/ossf/scorecard.

[13] 2020. GitHub - ossf/wg-best-practices-os-developers: The Best Practices for OSS Developers working group is dedicated to raising awareness and education of secure code best practices for open source developers. https://github.com/ossf/wg-best-practices-os-developers.

[14] 2021. Sonatype's 2021 Software Supply Chain Report. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021.

[15] 2021. YfryTchsGD/Log4jAttackSurface. https://github.com/YfryTchsGD/Log4jAttackSurface.

[16] 2022. Top-100 npm package maintainers now require 2FA. https://github.blog/2022-02-01-top-100-npm-package-maintainers-require-2fa-additional-security/.

[17] 2023. JavaScript eval security best practices. https://www.codiga.io/blog/javascript-eval-best-practices/.

[18] 2023. JavaScript require vs import. https://flexiple.com/javascript/javascript-require-vs-import/#how-it-works.

[19] 2023. JavaScript Require – How to Use the require() Function in JS. https://www.freecodecamp.org/news/how-to-use-the-javascript-require-function/.

[20] 2023. Unsafe Rust - The Rust Programming Language. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[21] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *EMSE* 25 (2020). https://doi.org/10.1007/s10664-019-09792-9

[22] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2022. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. https://doi.org/10.48550/ARXIV.2002.01139

[23] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1334–1346. https://doi.org/10.1109/ICSE43902.2021.00121

[24] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting Suspicious Package Updates. In *ICSE: New Ideas and Emerging Results*. 13–16. https://doi.org/10.1109/ICSE-NIER.2019.00012

[25] Mehdi Golzadeh. 2019. Analysing Socio-technical Congruence in the Package Dependency Network of Cargo. In *ESEC/FSE*. https://doi.org/10.1145/3338906.3342497

[26] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S. Păsăreanu, and David Lo. 2022. Test Mimicry to Assess the Exploitability of Library Vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 276–288. https://doi.org/10.1145/3533767.3534398

[27] Benjamin Livshits and Leo Meyerovich. 2009. CONSCRIPT: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. https://doi.org/10.1109/SP.2010.36

[28] Vittunyuta Maeprasart, Supatsara Wattanakriengkrai, Raula Kula, Christoph Treude, and Kenichi Matsumoto. 2023. Understanding the Role of External Pull Requests in the NPM Ecosystem. *EMSE* (03 2023). https://doi.org/10.1007/s10664-023-10315-w

[29] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium*. 1271–1287.

[30] A. Sejfia and M. Schafer. 2022. Practical Automated Detection of Malicious npm Packages. In *ICSE*. 1681–1692. https://doi.org/10.1145/3510003.3510104

[31] Vikram N. Subramanian, Ifraz Rehman, Meiyappan Nagappan, and Raula Gaikovina Kula. 2022. Analyzing First Contributions on GitHub: What Do Newcomers Do? *IEEE Software* 39, 1 (2022), 93–101. https://doi.org/10.1109/MS.2020.3041241

[32] Supatsara Wattanakriengkrai, Dong Wang, Raula Gaikovina Kula, Christoph Treude, Patanamon Thongtanunam, Takashi Ishio, and Kenichi Matsumoto. 2022. Giving Back: Contributions Congruent to Library Dependency Changes in a Software Ecosystem. *TSE* (2022), 1–13. https://doi.org/10.1109/TSE.2022.3225197

[33] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Yasemin Acar Harshini Sri Ramulu, and Sascha Fahl. 2023. "Always Contribute Back": A Qualitative Study on Security Challenges of the Open Source Supply Chain. In *44th IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP46215.2023.10179378

[34] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. 2023. Do Software Security Practices Yield Fewer Vulnerabilities?. In *ICSE: Software Engineering in Practice*. https://doi.org/10.1109/ICSE-SEIP58684.2023.00032

[35] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the Impact of Security Vulnerabilities in the Npm and RubyGems Dependency Networks. *EMSE* 27, 5 (2022), 45 pages. https://doi.org/10.1007/s10664-022-10154-1

Received 2023-05-03; accepted 2023-07-19