Know vulnerabilies OSS libraries

Ronaldo Canesqui
Southern Adventist University
ronaldocanesqui@southern.edu

***Abstract*—The use of open-source software is a reality. The exponential growth of available packages and their dependencies, The abstract should describe the basic message of the paper, including: the problem, why your solution should be of interest, some notion that your solution is effective, and a teaser about how it has been evaluated. Cover all of this using between 75 and 150 words. The abstract is thus the hardest part to write. Sometimes I try to write it first, but the final version is usually composed of items drawn from the introduction, and then condensed, as the last step of writing the paper.**

I. Introduction

The problem we have solved:

The use of open-source libraries is a reality in the software industry. In 2018, Synopsys[1] found open-source code in more than 96% of their audits[20]. Open-source software (OSS) usage has increased exponentially during the last decade. The most popular source for libraries in the Java ecosystem, Maven Central Repository, growth 542.17% from 2010 to 2016 [10]. Another popular open-source repository for the Javascript ecosystem, npm growth from 0 packages at its creation in 2010 to 1,000,000 packages in 2019. [2] The figure 1 extracted from [6] shows the number of packages evolution on other popular ecosystems. How to leverage the power of open source

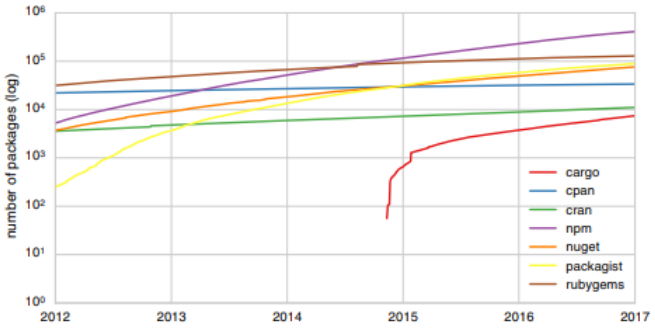

Fig. 1. Evolution of the number of packages.

libraries and still manage the risk of vulnerabilities they carry is a problem that either remain ignored by many software engineers or present considerable challenges to them.

Why the problem is not already solved or other solutions are ineffective in one or more important ways

A 2014 study from Sonatype determined that over 6% of the download requests from the Maven Central Repository were for component versions that included known vulnerabilities. In their review of over 1,500 applications, each of them had an average of 24 severe or critical flaws inherited from their components. [3] A white paper produced by Contrast security, stated that over 25% of all libraries download from Maven Central Repository has vulnerability. Only one vulnerable version of the Java GWT package was downloaded 17,666,703 times [21]. Among the 10 most popular npm packages, 6 present 1 or more vulnerabilities [4].

In a study on security vulnerabilities impact[5] the researchers found that out of 610,097 available packages (2017 data) 133,602 packages directly depend on a vulnerable package and 72,470 packages had at least one release that relies on a vulnerable package. In the same study, they also found a crescent number or vulnerabilities in the npm repository. The figure 2 shows he evolution of the number of vulnerabilities and the impact on dependent packages. Comparing 2014 data, where the distinct packages (doted lines) impact almost the same number of dependent package (straight lines), with 2017 data, it is evident the tendency of increasing impact on dependent packages.

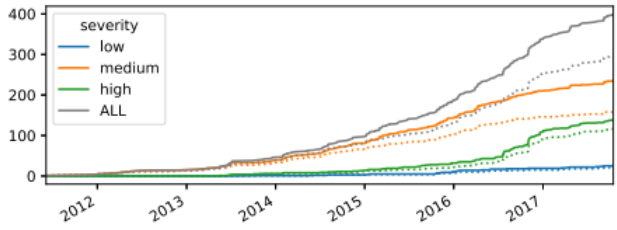

Fig. 2. Evolution of the number of discovered vulnerabilities (straight lines) and corresponding distinct packages (doted lines) per severity.

Why our solution is worth considering and why is it effective in some way that others are not

During this paper studies related to dependencies management will be presented along with the most recent studies about how to identify know vulnerabilities in open source libraries. The rest of this paper will cover studies on how long vulnerable package remains harmful and how long a fix takes to spread over the dependent applications II, what prevents software developers from updating their dependencies (section III) and solutions on how to manage the risk involved on open source libraries adoption. The rest of this paper first discusses related work in **??**, and then describes our implementation in **??**. V describes how we evaluated our system and presents

[1]https://www.blackducksoftware.com/

[2]https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/ accessed 10/10/2019

[3]Report published January 02, 2015 at http://goo.gl/i8J1Zq.

[4](https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn/ accessed 10/10/2019)

the results. VI presents our conclusions and describes future work.

II. Vulnerability Timeframe

To analyze for how long a vulnerability is harmful the study [5] used a 700 security vulnerabilities report made available by Snyk.io [5] and retrieved the list of its releases from the open source discovery service libraries.io [13]. Based on the list of releases, they identified which ones were affected by the vulnerability. Based on the relationship between package, vulnerability, first release data, vulnerability discovery date, it was possible to trace a vulnerability timeframe.

How long does the package remains vulnerable Figure 3 shows Kaplan-Meier estimator curve [9] for the event "vulnerability is fixed". The data presented considers the date of the affected release and the date that the fix was available. After 10 months, there is a probability higher than 80% that a high severity vulnerability is still unfixed.

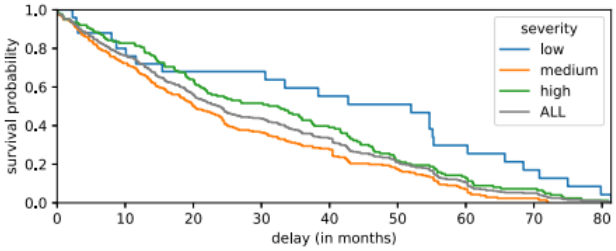

Fig. 3. Survival probability for event "vulnerability is fixed" w.r.t. the date of first affected release.

When a vulnerability is discovered Figure 4 shows that most severities are discovered in old packages. 75% of all vulnerabilities are found in libraries older than 13 months. Even tough it was not highlighted in the original study, the shorter wave in high severity vulnerabilities may suggest the higher priority in which they are handled, especially when compared to the low severity graph that shows smother curves. Most severities are found in packages older than 28 months.

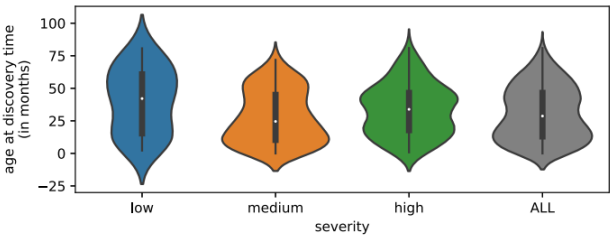

Fig. 4. Violin plots of packages age at discovery time by vulnerability severity.

When a vulnerability is fixed Most of the vulnerabilities are fixed between the discovery date and the public announcement. Figure 5 shows that there is a probability of 50% of a fix becomes available in the first month after discovery. And 80% of all vulnerabilities are fixed between 12 and 13 months. After 20 months of the discovery all high severity vulnerabilities were fixed. Some medium severity vulnerabilities, according to the graph, will take more than 40 months to be fixed.

[5]https://snyk.io

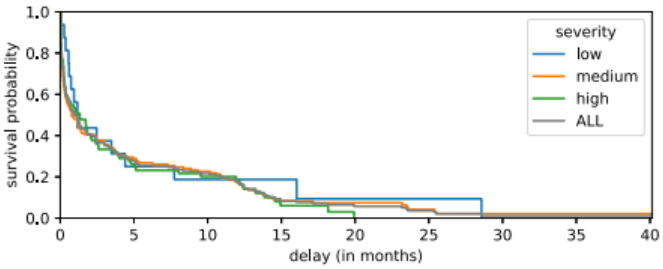

Fig. 5. Survival probability for event "vulnerability is fixes" w.r.t. vulnerability discovery time.

When a vulnerability is fixed in a dependent package The impact on dependent packages is important due to increasing number of dependencies, faster then the number of available packages. The figure 6 shows a more step curve when compared to the number of available packages (figure 1). Figure 7 shows that after 20 months, 100% of high severity vulnerabilities are fixed while there are 40% of the dependencies vulnerable at the same time. The slowness on updating dependencies found in this study is supported by similar findings in different studies covering a wide range of ecosystems such as SmallTalk[18], Pharo[8], Java [19], Apache products[2], Windows ecosystem [12], and Javascript ecosystem [11].

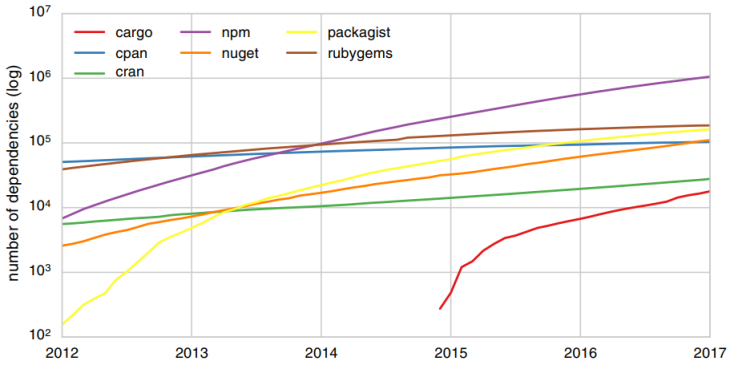

Fig. 6. Evolution of the number of dependencies (considering for each point in time the latest available release of each package).

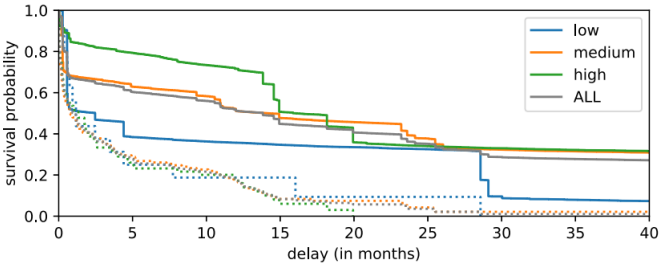

Fig. 7. Survival probability for event "package is fixed" w.r.t. vulnerability discovery time. Dependent packages are shown as straight lines and upstream packages as dotted lines.

III. Developer's Timeframe

To understand the vulnerabilities timeframe it is required to understand the application's maintainer behavior on updating and selecting their dependencies. A study [7] conducted with 203 app developers from Google Play clarified the following research questions: What is the common workflow to search for and to integrate third-party libraries into applications? How frequently do developers update their apps/libs and what is their main motivation for updates? What are possible reasons to not update dependencies and what solutions could app developers think of? Based on the applications maintainer's answers, the study conclude that 78% of them do not have a fixed schedule for app update (Figure 8) and only 33% of them mention a library update as a reason to update their app (Figure 9). When the maintainers were questioned why do you update your app's libraries, 96.47% of them answered bug fixing and 57.65% mentioned security (figure 10). Regarding library selection criteria, only 26.58% answered security, even thought the answer update frequency is related and received 35.16% of the votes (figure 11). When asked reasons why your app would include outdated libraries?, 57.03% answered that the library was still working, 50% answered to prevent incompatibilities and 32.81% were unaware of updates (figure 12).

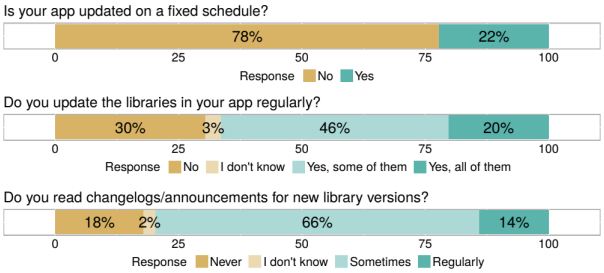

Fig. 8. Anwsers for questions regarding app/library release frequency.

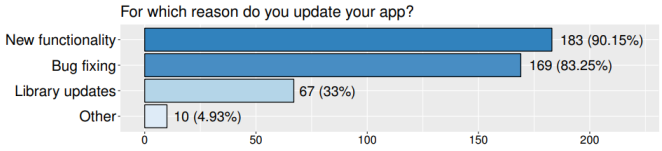

Fig. 9. Answers for questions:For which reason do you update your app?

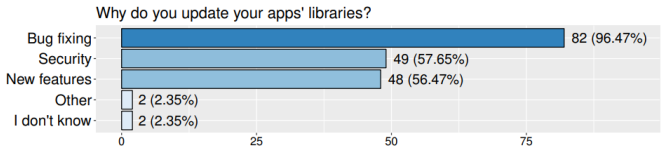

Fig. 10. Answers for questions: Why do you update your apps' libraries?

The study concludes that the main reasons to keep a library outdated are incompatibility prevention, unaware of updates and too much effort. Other study on this field [10] reaches at the same conclusion regarding why maintainers keep outdated dependencies. On following sections, studies that proposed ways to deal with those main problems are presented along with their conclusions.

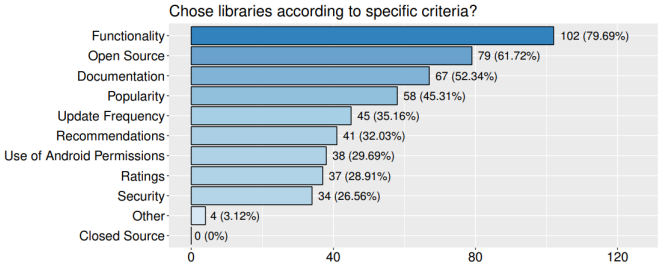

Fig. 11. Answers for questions: Chose libraries according to specific criteria?

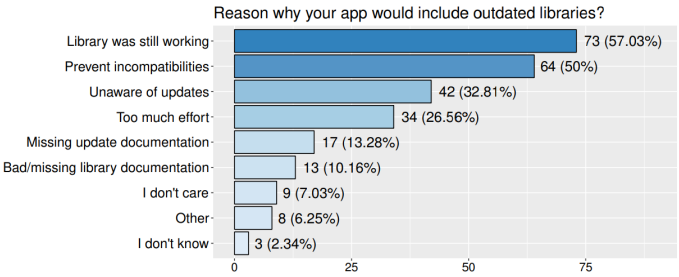

Fig. 12. Answers for questions: Reason why your app would include outdated libraries?

IV. Increasing Awareness and Reducing Effort on Dependency Management

Various researchers have worked to solve the problem of know vulnerabilities in open source packages and their dependencies. In this section, the most recent works will be presented along with their conclusions.

Predicting vulnerabilities: Even thought the focus of this work is on papers that detect vulnerabilities using know vulnerabilities databases, it is worth mentioning that there is a branch of proposed solutions that operates identifying vulnerabilities based on a set of characteristics. Part of this solutions uses machine learning to identify vulnerabilities. One of this works [15] mapped the CVE[6] entries to the commit that generated the vulnerability and trained a SVM-based model using the commit's metadata. According to the authors this method reduced the amount of false positives by 99% when compared to Flawfinder[7]. It was capable of detecting 53 of the 219 known vulnerabilities used in the study and only producing 36 false positives. The proposed solution analyses each commit and tries to predict the presence of vulnerabilities.

Measuring dependency freshness The study [4] proposes a method to measure the dependency freshness. The authors used information from project's pom.xml file to determiner what is the version being used and used the Maven Central Repository to determine the release history for the library. Based on this information, the authors were able to classify each used component in a risk profile with 4 categories: low, moderate, high and very high. To determine the relationship between their risk profile and the security vulnerabilities the researchers matched the dependencies list and tried to match

[6]Common Vulnerabilites and Exposures - https://cve.mitre.org/
[7]https://https://dwheeler.com/flawfinder/

with the CVE vulnerability database. They were able to find a correlation between their freshness index and the number of vulnerabilities. The median variance of the rating, the researchers were able to classify the systems according to dependency freshness:

- **Stable** Systems with a stable dependency freshness rating. The system dependencies see little to no updates.
- **Improving** Systems with an increasing dependency freshness rating. Dependencies are updated faster than they are released.
- **Declining** Systems with an decreasing dependency freshness rating. Dependencies are updated slower than they are released.

The metrics presented have a great potential in quantifying the dependency freshness. The method has at least one serious limitation: only direct dependencies can be measure, which can generate wrong classifications, specially when direct dependency is updated but a transient dependency exists and is outdated.

Library updatability The study [4] was able to correlated library freshness with number of vulnerabilities. The finding that old libraries carry more vulnerabilities is also supported by other works [5], [20]. The study [7] proposes an automated method to evaluate the updatability of dependencies. Their method consist of three steps: determine the API robustness, determine the library usage and determine the library updatability. To determine the API robustness, the researchers extracted the public API from multiple versions of the same library, resulting in a library version/API pair list. According to authors this is a more fine-grained approach compared to [1]. The authors then proceed with the library usage, where they inspect the bytecode looking for call to the API. Finally they matched the library version/APi pair list with the library usage to produce the library updatability, which informs what is the latest version the API can be updated to without causing incompatibilities.

The authors test their approach analyzing 98 libraries and 1,246,118 apps from Google Play. The results shows that in 85.6% of the cases the identified library can be upgraded by at least one version (Upgrade1+) and on 48.2% of the cases the library can be updated to the most current version simply by replace the old library, without any code change. Figure 13 shows the library updatability from their study.

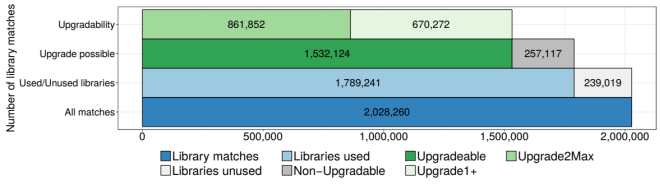

Fig. 13. Library updatability of current apps on Google Play.

Vulnerability Alert Service (VAS) The Vulnerability Alert Service (VAS) is proposed by [3]. Their focus is to increase the awareness about known vulnerabilities by making it part of the software quality process. The overall process is illustrated by figure 14. The target project has its dependencies extracted and recognized. The list of dependencies is then analyzed by the matching task, which tries to match the dependency with a entry from the vulnerability disclosures, in this study a CVE entry. Upon a successful match, an alert is produced which is consumed by a human operator. The authors extended the OWASP Dependency Check Tool to extract the dependencies list from Maven pom.xml files and used this tool as the vulnerability checker described in the diagram. The conclusion of this work shows a false positive rate of over 70%, but the authors state that the rate is smaller when the solution was actually deployed. The researchers affirm that the high false-positive can be caused by situations like the MySQL-connector jar which is flagged with the vulnerabilities of the MySQL database server. The evaluation with the human operators shown that despite the false positive rate, an alert system was considered useful. Even the authors were satisfied with the operators' evaluation, since their solution looks for dependencies in pom.xml files, only direct dependencies can be identified.

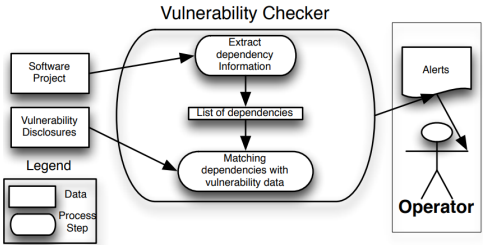

Fig. 14. The Vulnerability Alert Service process

Code-based vulnerability identification The study [17] uses a method to identify vulnerabilities that, according to the authors, is more accurate then the methods proposed so far. Their work aims to answer the questions: does my application is vulnerable if it uses a vulnerable dependency? The researchers claim that this is not always the case and other studies failed to answer this question. The authors state that the software that contains a certain library that has known vulnerability does not need to be subject to costly efforts if proven that the vulnerability does not present harm. This way, the fix can be applied in the subsequent releases under no additional cost. Conceptually their solution can be used on any ecosystem, but their implementation is based on the Java ecosystem and the Maven Central Repository. They used the project's pom.xml file to determine the dependencies. After the dependencies are determined, they remove all dependencies classified as provided and test, since those will not be deployed. Then the halted libraries are identified. One key aspect for this approach to work is a database vulnerability that contains the meta-data about library, such as name and version according to Maven Central Repository, the vulnerable source code and the source code for the fix. Using this database is possible to associate the dependency identified previously and the vulnerability, along with the source-code. This database was manually produced by the authors it covers 90% of Java ecosystem vulnerabilities. The authors used static and dynamic analysis to determine if the vulnerable code is reachable. The

dynamically analysis is done by instrumenting the vulnerable code and running automated and manual tests. Their solution will notify the user if the vulnerability has impact on the application, meaning, the vulnerable code can be reached.

Other studies from the same authors use the same method to identify know vulnerabilities but focus on different aspects of the solution. [16] introduces a simpler but very similar method to the one previously described. It was presented as a proof-of-concept and targets performance comparison between their approach, OWASP Dependency Check and two commercial tools. [14] focus on how the impact of know vulnerabilities is inflated because other studies are counting vulnerable dependencies that are not deployed, such as libraries related to tests.

Then, describe the structure of the rest of this section, and what each subsection describes. [16]

How our solution (will | does) work This is the body of the subordinate paper describing your solution. It may be divided into several subsections as required by the nature of your implementation.

The level of detail about how the solution works is determined by what is appropriate to the type of paper (conference, journal, technical report)

This section can be fairly short for conference papers, fairly long for journal papers, or *quite* long in technical reports. It all depends on the purpose of the paper and the target audience

Proposals are necessarily a good deal more vague in this section since you have to convince someone you know enough to have a good chance of building a solution, but that you have not *already* done so.

V. Evaluation

How we tested our solution

- Performance metrics
- Performance parameters
- Experimental design

How our solution performed, how its performance compared to that of other solutions mentioned in related work, and how these results show that our solution is effective:

- Presentation and Interpretation
- Why, how, and to what degree our solution is better
- Why the reader should be impressed with our solution
- Comments

Context and limitations of our solution as required for summation: State what the results *do* and *do not* say.

VI. Conclusions and Future Work

The problem we have solved: The most succinct statement of the problem in the paper. Ideally one sentence. More realistically two or three. Remember that you simply state it without argument. If you have written a good paper you are simply reminding the reader of what they now believe and of how much they agree with you.

Our solution to the problem: again, a succinct statement of the presented solution Sometimes it works well to leave it at that and not even describe your solution here. If you do, then again state your solution in one or two sentences taking the rhetorical stance that this is all obvious. If you have a good solution and have written an effective paper, then the reader already agrees with you.

Why our solution is worthwhile in some significant way: Again, a succinct restatement in just a few sentences of why your solution is worthwhile assuming the reader already agrees with you

Why the reader should be impressed and/or pleased to have read the paper: A few sentences about why your solution is valuable, and thus why the reader should be glad to have read the paper and why they should be glad you did this work.

What we will (or could) do next

- Improve our solution
- Apply our solution to harder or more realistic versions of this problem
- Apply our solution or a related solution to a related problem

References

[1] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 356–367. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978333

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, Oct 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9325-9

[3] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 516–519.

[4] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 109–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819027

[5] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 181–191. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8595201

[6] A. Decan and T. M. P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems." [Online]. Available: https://link.springer.com/content/pdf/10.1007/s10664-017-9589-y.pdf

[7] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. [Online]. Available: https://dl.acm.org/citation.cfm?id=3134059

[8] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 251–260.

[9] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958. [Online]. Available: http://www.jstor.org/stable/2281868

[10] G. D. O. A. Kula, R.G., "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, 2018. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s10664-017-9521-5.pdf

[11] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web." [Online]. Available: https://arxiv.org/abs/1811.00918

[12] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 692–708.

[13] A. Nesbitt and B. Nickolls, "Libraries.io Open Source Repository and Dependency Metadata," Jun. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.808273

[14] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2018.

[15] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," 10 2015, pp. 426–437.

[16] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 411–420.

[17] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 449–460.

[18] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: The case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 56:1–56:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393662

[19] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular java apis," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 400–410.

[20] Synopsys, "2019 open source security and risk analysis," *Synopsys Open Source Security and Risk Analysis*, 2019.

[21] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1–26, 2012.