Analyzing the Criticality of NPM Packages Through a Time-Dependent Dependency Graph

A.J.M.Brands
Supervisor(s): G. Gousios, D. Spinellis
EEMCS, Delft University of Technology, The Netherlands
20-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

In (open-source) development, developers routinely rely on other libraries to improve their coding efficiency by reusing code. This reliance on other packages could cause issues when critical dependencies have suddenly have a vulnerability introduced to them. This work analyzes the criticality for NPM. To get an accurate picture of what the most-critical and thus possibly most-vulnerable packages are, the entirety of NPM must be analyzed. However, this proved too big to be able to fit in 500GB of memory. This work therefore examines a small subset of 100 thousand packages. To do the analysis, this paper proposes a novel approach of embedding a time dimension into the package network to provide better accuracy. This papers analysis show that both with and without this time dimension, `babel` packages are by far the most important in the package graph (as measured by PageRank). We should, however, keep in mind that this came from only analyzing 100 thousand packages. Thus, further research is required to confirm this conclusion. In particular, other importance measures should be used to find out the packages' criticality.

1 Introduction

In open-source software development, developers routinely use open-source packages or libraries as building blocks for their own projects. According to Mohagheghi and Conradi [1], such re-use of code can help developers considerably speed up their work. Nevertheless, this introduction of other packages inherently makes a project dependent on them [2]. This could make projects vulnerable to problems caused by direct dependencies when said dependencies are not kept up-to-date [3]. Examples of such a problem would be breaking changes to a package, or its outright removal. In addition to that, dependencies of dependencies (transitive dependencies) could themselves introduce problems and cause incidents [4].

Over the years, many such incidents have illustrated that blindly depending on third-party packages is not risk free (e.g. *left-pad*, *NotPetya*, *SolarWinds*, and most recently *Log4Shell*) [5]–[8]. The *left-pad* incident is a perfect example of how one single package can cause disastrous consequences for many other packages through transitive dependency [2], [5].

Because of the disastrous effect a change in one critical package can have on packages that (transitively) depend on it, studies have addressed the need to analyse dependencies in package managers [2], [9], [10]. This is commonly achieved by analyzing package metadata available in package managers such as NPM, Maven, and PyPI. Research often starts with the inference of Package Dependency Networks (PDNs), followed by analysis of said networks to determine the most-critical packages [10]. What these PDN studies commonly fail to consider, however, is how different versions of packages can have entirely different transitive dependencies in different time frames.

Because the time dimension is often missed in the graph generation, the PDNs only show (transitive) dependencies for the latest version of packages. As Kikas *et al.* [2] mention, this approach might not be enough, and a more in-depth analysis of network is needed to understand what impact a critical package can have. This suggests that leaving out the time dimension yields a mere subset of the information about package dependencies. This research proposes a time-based dependency network analysis, by incorporating a time dependency in the dependency network. The incorporation of the time dimension to PDNs will make sure the dependency graph is complete for every version of every package. To generate a time-based PDN, this work will devise an algorithm that resolves dependency graphs for every version of a package with the addition of time metadata. In the final step of the algorithm, all dependency graphs are merged together. This results in a more detailed dependency network that can specify which specific versions of packages are the most critical packages. To demonstrate the feasibility of this approach, the aforementioned algorithm will be implemented for `NPM`.

In short, this work contributes the following:

- An algorithm to create time-based package dependency networks (tPDNs).
- An approach for generating a tPDN of `NPM`.
- A study comparing `NPM` network analysis using PDNs with and without the time dimension.
- The generated `NPM` tPDN for replication.

Answering the first two sub questions will help answer the implicit main research question (RQ3):

1. *RQ1*: "What should a graph data structure modeling package dependencies look like?"
2. *RQ2*: "On average, does the introduction of the time dimension lead to a significant change in the number of dependent packages per package?"
3. *RQ3*: "What are the most-critical packages on NPM?"

Structure In section 2, important terms and background information are discussed. After that, section 3 provides an overview of the work done to answer the research questions. The following subsection provides a more detailed description of the experimental setup in section, after which the results are discussed in detail in section 4. Section 5.1 discusses the ethical concerns involved in this research, followed by a justification of how reproducible this research is in 5.2. The section after that (section 6) discusses these results and their limitations, comparing them to contemporary research. Lastly, section 7.1 answers the research questions and summarizes the main contributions of this research, after which 7.2 summarizes open issues and new questions arising from this work.

2 Background

The following subsections will help a reader inexperienced with package managers and related terminology better understand the main sections of the report. In the first section, the most important terminology is defined. In the section after

that, some related work is provided to illustrate what is still missing in contemporary research.

2.1 Terminology

1. *Transitive dependencies*: The dependencies of a package that are indirectly introduced through the inclusion of other dependencies. In other words: the dependencies of dependencies and their dependencies, and so on.
2. *NPM*: Node Package Manager[1]. The default package manager for JavaScript packages.
3. *Libraries.io*[2]: A service indexing about twenty package managers to provide all kinds of metadata on them. For example, it provides popularity, the current number of dependents, and the current number of dependencies per package.
4. *Package Dependency Network (PDN)*: A graph representation modeling the interdependency between (a subset of) a package manager's packages. It has package (optionally per version) as nodes, with a directed edge between package `A` and `B` meaning that `A` depends on `B`.
5. *Semantic versioning*: A versioning system in the following format: `MAJOR.MINOR.PATCH`. Changing the major version (`MAJOR`) implies a breaking change. Changing the minor version (`MINOR`) implies a minor (backwards-compatible) change. Lastly, changing patch (`PATCH`) version implies an even smaller change [11].
6. *Time-Based Package Dependency Network (tPDN)*: A package dependency network extended with the ability to filter on some time frame. Using this filter, one can see which exact versions of packages other packages depend on during said time frame.
7. *PageRank*: One of the algorithms Google uses to rank web pages based by calculating the number of links between web nodes to estimate their importance [12]. In this research it will be used to estimate package importance in a similar fashion.

2.2 Related Work

As Kikas *et al.* [2], Vidoni [9], and Hejderup *et al.* [10] all demonstrated, analyzing package repositories has become crucial because of incidents such as the deletion of the `left-pad` package from the NPM repository [5]. This incident in particular showed that a package containing very little code could affect a large portion of the other packages not even directly depending on it. According to Hejderup *et al.* [10], a lot of packages depend on popular packages such as `babel`, which themselves depend on smaller packages that do relatively little work. However, as Kikas *et al.* [2, p. 110] demonstrated, these so-called "utility packages" can potentially influence more than a third of all packages stored in the NPM repository. In addition to that, the rapid growth of the amount of packages in the NPM registry makes critical packages grow even more and more important [13]. The main problem with this growth is the ever-increasing number of transitive dependencies. Moreover, Zimmermann *et al.* [14] show that in 2019, 40% of `NPM`'s packages included at least one known vulnerability. In combination with Decan *et al.* [13]'s work, this suggests that a lot of NPM might still be quite vulnerable to malicious actors influencing utility packages.

Researchers often make use of network analysis with a PDN generated from metadata available package managers[10]. What these studies often fail to consider, is how a package's transitive dependencies can potentially drastically change over time. Because this time dimension is often overlooked, analyses often only show the transitive dependencies for the latest versions of all packages at some specific time. Kikas *et al.* [2] suggest that contemporary PDN analysis is potentially too shallow. In fact, Wang *et al.* [15]'s work suggest that in general, time-dependent networks can provide critical insight into a network's evolution. Combining both suggests that including the time dimension could be crucial to having a more complete picture of how packages transitively influence one another. This work proposes the addition of a time filter to a package dependency network. This will provide a more complete picture of package interdependency throughout time. To generate such a graph, this work proposes an algorithm that incorporates Wang *et al.* [15]'s work to create a time-dependent graph from a static PDN generated from NPM metadata.

3 Finding the Most-Critical Packages on NPM

To find out what incorporating a time dimension in the NPM dependency graph should look like, a short literature review of related work was performed. In addition to that, a small sample of Libraries.io data was used to get insight into what Libraries.io deems the most important packages. After that, the answers to the sub-questions can be used to answer the main research question (question 3.3).

3.1 RQ1: "What should a graph data structure modeling package dependencies look like?"

Three main steps were required to answer the first research question. First, the required data for all packages [3] had to be collected from NPM. After that, the requirements for the graph and the graph generation process were formalized. After implementing the graph generation algorithm, some automated tests were run on a small test set to verify that the node generation and edge building process worked. Following that, manual correctness tests were run on a small subset of actual data. This entailed picking 10 random packages [4], and setting the time range such that it spans the entire time

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

[2] https://libraries.io/

[3] names, versions, version release timestamp, and dependencies per version

[4] `@angular/animation@4.0.0-beta.8`, `@angular/cli@14.0.2`, `@angular/pwa@14.0.2`, `@babel/core@7.18.5`, `@babel/cli@7.17.10`, `@babel/generator@7.18.2`, `@bazel/angular@4.6.2`, `@cocreate/cli@1.12.48`, `@codaco/eslint-plugin-spellcheck@0.0.14`, `@commitlint/cli@2.0.0` (See Section 6.2)

range of packages in the input file. After that, the latest version and dependencies that the graph and NPM reported were compared using the formula in Figure 1. This made sure the tPDN could at least show the same data NPM reports by itself.

Let:

- A be the set of transitive dependencies resolved by NPM
- B be the set of transitive dependencies resolved using the implemented algorithm
- E be the number of dependencies that have a correct name but incorrect version

We calculate the accuracy of the algorithm by this formula:

$$Acc = \begin{cases} 1 - \frac{|A| - |(B \cap A)| + 0.5 * E}{|A|}, & \text{if } A \neq \emptyset \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

Figure 1: Accuracy equation. Heavily inspired by [16]

3.2 RQ2: "On average, does the introduction of the time dimension lead to a significant change in the number of dependent packages per package?"

Answering the second research question required two main steps. In the first step, a random sample of 10 packages[4] were ranked on number of transitive dependents using NPM's own data using `npm-remote-ls`. After that, for the same sample of packages, the PageRank was measured using the tPDN for two different time frames [5]. After measuring what both the tPDN and NPM itself reported, the Pearson correlation coefficient was calculated to see if the PageRank for the time frames had any relation with the amount NPM reports.

3.3 RQ3: "What are the most-critical packages on NPM?"

To find out the most-critical packages on NPM, the three main steps were generation of the tPDN for NPM, followed by the calculation of criticality measures, and investigating the correlation between criticality and download count. In the first step, a data dump of the top 100000 packages[6] of the NPM repository was processed and converted to a graph data structure. After that, the criticality of packages was measured using PageRank[12] and node betweenness centrality[17] for two different time frames[5]. Lastly, the Pearson correlation [18] between these measures and the package download count was computed for the first 25 packages and last 5 packages from the top 100 by PageRank. to see whether the most-downloaded packages are actually also the most-critical packages. Provided that this correlation coefficient would be highly positive (between 0.7 and 1.0), package rank could actually be correlated with download count.

[5] the year 2019, and the year 2021

[6] In lexicographic order. (See Section 6.2)

3.4 Experimental Setup

Tools used The following tools were used to analyse the NPM package dependency network:

- The Go programming language [7] and the Gonum graph library [8]
- The NPM database [9]
- Custom code to interactively create and explore graphs [10]
- Libraries.io to get an idea of what the NPM data looked like
- Graphia [11] to visualize graphs to provide an intuitive view of what the tPDNs look like [12]

Gathering the required data To generate the time-based dependency graph, package metadata, had to be collected. This was first tried through the official NPM API, which unfortunately implements rate-limiting. Data collection using NPM's undocumented database endpoint[9] resulted in a big data dump that had to be pre-processed first. This data dump can be found at Zenodo [19].

Preprocessing the data The large amount of data gathered in the data acquisition step was distilled into useful data[3] using custom code that can be found in Appendix B. This led to an intermediate file format that could be used for graph generation (See Figure 6). This intermediate file can also be found at Zenodo [19]

Graph Design After gathering and preprocessing the data, the team debated on what useful time-based dependency graph representation was. This resulted in a graph structure where nodes store the required information [3], and a directed edge from node A to node B signifies that A depends on B. This is quite similar to what Kikas *et al.* [2] proposed. Where the proposed tPDN differs is that it also stores release time stamps for packages. This will enable time-dependent querying of the graph. (See section 4.1 for more details)

Generating the graph To generate the described graph data structure, the algorithm roughly works as follows:

1. Using the file created in the preproccessing step, create nodes per package version
2. Generate indices for quick node lookup
3. Again using the input file, determine which nodes should be connected because they have a dependency relationship

For a more detailed look, Appendix B describes where to find the source code.

Calculating metrics After all the steps above, the interactive graph explorer that can be found in the source code was used to calculate a ranking for packages using the PageRank. The graph was also filtered as described in Section 3.2.

[7] https://go.dev/solutions/#case-studies

[8] https://pkg.go.dev/gonum.org/v1/gonum/graph

[9] https://replicate.npmjs.com/_all_docs?include_docs=true

[10] See Appendix B

[11] https://graphia.app/

[12] See appendix C

4 Results

In the subsections below, the results of the three research questions will be presented.

4.1 RQ1: ”What should a graph data structure modeling package dependencies look like?”

Graph structure After thorough discussion, the graph data structure had all packages connected to all possible versions of their dependencies (see Figure 2 for an example). After filtering by time stamp, the graph only connects packages to the latest possible version allowed by the semantic versioning constraint (see Figure 3 for an abstract example).

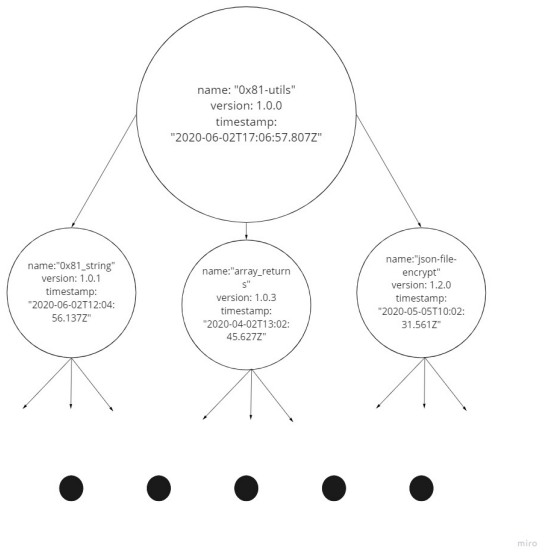

Figure 2: Dependency graph without filtering applied

Automated testing The small set of automated tests all passed, indicating that for small datasets, the node and edge generation was as expected [13] (see Appendix B for details about the code).

Manual accuracy testing Ten packages were picked for accuracy testing [4]. The ground truth about these packages was established using the command `npx npm-remote-ls package` for every package. After that, the graph was queried for the latest dependencies. The accuracy scores calculated by the formula in Figure 1 can be found in Table 1 below. Clearly, the accuracy scores are quite low because of the small amount of data used. Section 6 goes into more detail about why this is the case.

4.2 RQ2: ”On average, does the introduction of the time dimension lead to a significant change in the number of dependent packages per package?”

For the year 2019, we can see that all of the sample packages had a negligible PageRank (Table 2). This led to an undefined Pearson coefficient, which means nothing of note can be said about the correlation between the PageRank and the number of transitive dependencies NPM reports. The same unfortunately holds for the year 2021.

[13] i.e. as described in section 3.4

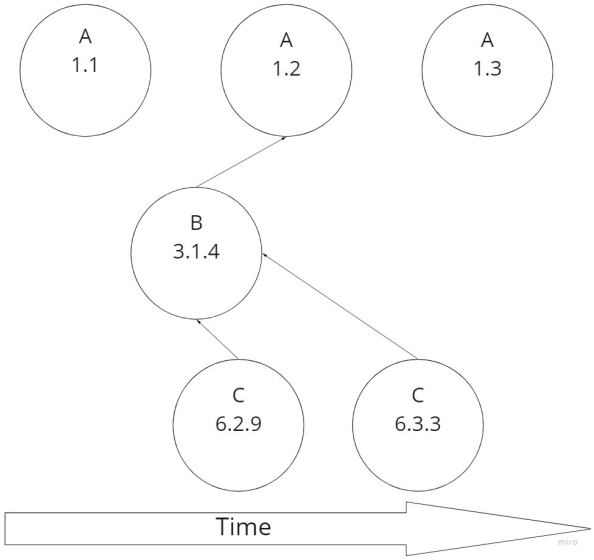

Figure 3: Timed dependency graph. (Abstract example for clarity)

Package name	Accuracy score
@angular/animation	1.000
@angular/cli	0,014
@angular/pwa	0.047
@babel/core	0.039
@babel/cli	0.080
@babel/generator	0.188
@bazel/angular	0.150
@cocreate/cli	0.102
@codaco/eslint-plugin-spellcheck	0.005
@codaco/shared-consts	0

Table 1: Package resolution accuracy scores (rounded to 3 decimals)

Package	2019	2021	#trans. deps
@angular/animation	0	0	1
@angular/cli	0	0	383
@angular/pwa	0	0	60
@babel/core	0	0	129
@babel/generator	0	0	16
@bazel/angular	0	0	20
@cocreate/cli	0	0	891
@codaco/eslint-plugin-spellcheck	0	0	980
@codaco/shared-consts	0	0	870

Table 2: PageRank versus number of transitive dependencies for 10 random packages. The first column specifies the rank for 2019, and the specifies the rank for 2021. The third column is the number of transitive dependencies

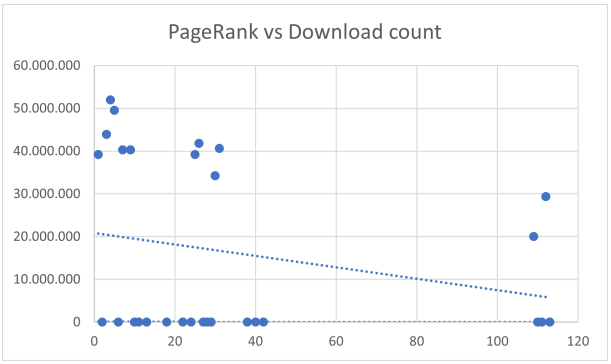

Figure 4: Package normalized PageRank vs Download count. The horizontal axis is the ranking and the vertical axis signifies the download count.

4.3 RQ3: "What are the most-critical packages on NPM?"

Most-critical packages by PageRank The PageRank score indicates that `@babel` packages are really heavily depended upon, since they fill up more than 50 percent of the top-100 most-critical packages (See Figure 8). Purcaru [16] provides a more detailed look into what packages can do over time (keeping in mind that this research is for PyPI).

Correlation between most-critical and most-downloaded packages Due to time constraints, a relatively small sample was used to calculate the correlation. As we can see below, there does not seem to be much correlation between the variables (Figure 4). In fact, the Pearson correlation coefficient for this sample is approximately -0.245.

5 Responsible Research and Reproducibility

To make sure the research in this report was conducted responsibly, the two subsections below outline the ethical aspects and the reproducibility of this work.

5.1 Ethics

The main ethical concern of this research is that it could potentially have a major influence on how the development community sees NPM and how they choose to use it. First of all, it may cause a large-scale deprecation of many so-called "utility packages" [2, p. 110] containing a significantly small amount of code. However, as they report, this may be mostly positive, as this would decrease the number of transitive dependencies per package. In addition to that, Decan *et al.* [13] and Zimmermann *et al.* [14] demonstrated that transitive dependencies were increasing despite incidents and research warning about the side-effects of using utility packages. Assuming that still holds today, one more research paper is not likely to change this growth. Other than the possibly influence this work can have on the JavaScript developing community, there are not many ethical concerns because no human research is involved. In addition to that, all data used is publicly available, thus not causing more privacy violation than NPM does on its own. Moreover, the data processing pipeline discards personal information and only keeps package names, versions, and their release timestamps.

5.2 Reproducibility

Since the first research question requires requirements analysis, it is inherently open to interpretation. Therefore, another researcher answering this question might not get the same result. Future researchers should get the same conclusion about the correctness of the graph (source code linked in Appendix B).

In contrast, the second research question is quite reproducible, provided that the description of the methodology, and the experimental setup and results in sections 3 and 4 is detailed enough. When computing the statistics, this should yield the same results within a margin of error.

The third and main research question requires both the algorithm used to process data and generate a graph from it, and the actual input data. Since the experimental setup mentions where to find the input data and all the source code with the help of appendix A and B, this part is quite reproducible. In addition to that, section 4 should be detailed enough to verify the results.

An additional point not related to any specific research question is that the *unofficial* way to query the NPM database[9], might stop existing at any time. This might threaten the reproducibility of the research, since the data at Zenodo [19] is only valid for the specific point in time that the download was completed.

Lastly, the fact that the code is hardly documented and not cleaned-up should be considered. The only clear documentation it has is the `README` file. Future researchers wishing to extend or modify the code will likely have a hard time understanding what every part does.

6 Discussion

In the following subsections, the results are discussed. The first subsection argues for the significance of the result, while the subsection after that discusses limiting factors in this research.

6.1 Significance of the results

In the results for RQ1, we can see that the graph achieved a very poor accuracy score for the random sample of packages. This is mainly because the memory constraints discussed in the next section. Given a combination of more memory-efficient code and more RAM, the graph would have fit more package dependencies. This would most likely have led to a higher accuracy score as shown by Purcaru [16]. Furthermore, the results for RQ2 for the years 2019 and 2021 are inconclusive because all the sampled packages had a negligible PageRank. This could signify a possible bug in the code or a lack of input data.

Lastly, the limited sample of packages in RQ3 results in a seemingly negative correlation between package PageRank and number of downloads. To confirm this unlikely result, an investigation with a much larger sample size and more input data is possibly required.

6.2 Limitations

It should be noted that the results mentioned in section 4 have some limitations, however. In addition to the fact that the

research questions could have been clearer, the research had some other limitations.

Graph memory footprint On of the most important limitations is the fact that not all NPM packages fit into a reasonable amount of memory using Gonum's default graph implementation, as it would require upwards of 500GB of memory [14]. With some experimentation, this memory footprint was potentially reduced by half. Nevertheless, only the graph representation of about 100000 packages could fit into memory[15] with the approximately 2 million unique versions and 270 million edges it creates. Because of this, only this amount of packages could be used for sampling, which unfortunately led to an incomplete graph and thus incomplete and inaccurate results.

Limitations in input data The next limitation in the results is the fact that the input data has some limitations. First of all, some dependencies do not follow the format specified by the semantic versioning specification [11]. This leads to guesswork about which dependency version a package actually requires, possibly resulting in an inaccurate package dependency graph. In addition to that, some packages do not specify versions at all, which lead to the preprocessing pipeline possibly erroneously excluding it from the intermediate file. Lastly, NPM has a lot of garbage packages polluting it, taking needless memory space and possibly corrupting other package's records in said intermediate format.

Limitations in metrics Another important limitation to consider is the fact that the metric used to calculate package importance might not be entirely accurate. First of all, the fact that PageRank [12] is probabilistic must be considered. Given a low enough tolerance, it might appear deterministic, but could still lead to slightly different results. The second problem with PageRank is that in its gonum implementation[16], it takes an extraordinarily large amount of memory [17]. Luckily, it also provides a memory-friendlier implementation that assumes a sparse graph. The problem with this is that, without further analysis, this assumption cannot be blindly assumed to be correct. This may therefore lead to inaccurate results. Another metric used that has some limitations is node betweenness centrality. The way it is calculated in gonum requires a lot of cpu time [18], which led to a lack of results for this metric. A last point of the metrics gathering is that part of it relied on manual comparison, which is prone to human error.

Code limitations Lastly, the code likely has some limitations. Despite manual and automated testing, the existence of bugs is very likely. Before continuing research with the code in this paper, these errors will probably need to be corrected.

[14] Estimation based on some experimentation

[15] This takes about 40GB of RAM

[16] https://pkg.go.dev/gonum.org/v1/gonum/graph/network#PageRank

[17] For 100 thousand packages, it wanted to reserve approx. 1TB of memory

[18] https://pkg.go.dev/gonum.org/v1/gonum/graph/network#Betweenness

7 Conclusions and Future Work

7.1 Conclusions

This work's most important contribution is the time-based graph approach to exploring package dependency networks. That is, exploring the package dependency networks using the tPDN theorized and implemented in this paper. In addition to that, this paper provides a starting point for insight in the most-critical packages hosted on NPM. To provide an accurate view about what packages are actually the most-critical, there are some future improvements the next subsection mebntions. The most-critical packages, as this work re-iterates, might be the most vulnerable to threats as well.

7.2 Future work

Graph generation efficiency The most important limitation of this work is the memory requirements for graph generation. Therefore, to get more accurate results, it is imperative that research extending this work implement a more memory-efficient graph generation algorithm. One suggestion is to build a graph library from the ground up in a more memory-efficient language, therefore only using the absolute minimum amount of memory. If the memory footprint is minimized, the graph generation can potentially be concurrent, speeding up the process. Another suggestion is to only store the absolutely required part of the graph in memory and then storing the rest on disk. To evaluate the validity of these suggestions, further research is needed.

Potential of tPDNs Furthermore, as mentioned in Section 6.2, the accuracy of the tPDN was not fully verified because of the limited amount of samples. Therefore, future research must more thoroughly assess its validity. After tPDN is more thoroughly validated, though, it could probably be used to compare different criticality measures as well. In addition to that it should probably be used to research other package managers than Maven, NPM, PyPI, and the Debian package manager. They could potentially benefit from time-based analysis as well. Lastly, the correlation between most-critical packages and package vulnerability should be examined to understand what can be done to repair vulnerable packages.

References

[1] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: A review of industrial studies," *Empirical Software Engineering*, vol. 12, pp. 471–516, 5 Sep. 2007, ISSN: 1382-3256. DOI: 10.1007/s10664-007-9040-x. [Online]. Available: http://link.springer.com/10.1007/s10664-007-9040-x.

[2] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," IEEE, May 2017, pp. 102–112, ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.55. [Online]. Available: http://ieeexplore.ieee.org/document/7962360/.

[3] J. Cox, E. Bouwers, M. V. Eekelen, and J. Visser, "Measuring dependency freshness in software systems," *Proceedings - International Conference on Software Engineering*, vol. 2, pp. 109–118, Aug. 2015, ISSN: 02705257. DOI: 10.1109/ICSE.2015.140.

[4] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules? — tu delft repositories," Delft University of Technology, May 2015. [Online]. Available: http://resolver.tudelft.nl/uuid:3a15293b-16f6-4e9d-b6a2-f02cd52f1a9e.

[5] I. Schlueter, *Npm blog archive: Kik, left-pad, and npm*, Mar. 2016. [Online]. Available: https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[6] C. Cimpanu, *Petya ransomware outbreak originated in ukraine via tainted accounting software*, Jun. 2017. [Online]. Available: https://www.bleepingcomputer.com/news/security/petya-ransomware-outbreak-originated-in-ukraine-via-tainted-accounting-software/.

[7] ——, *Microsoft says it identified 40+ victims of the solarwinds hack*, Dec. 2020. [Online]. Available: https://www.zdnet.com/article/microsoft-says-it-identified-40-victims-of-the-solarwinds-hack/.

[8] S. Povolny and D. McKee, *Log4shell vulnerability is the coal in our stocking for 2021*, Dec. 2021. [Online]. Available: https://www.mcafee.com/blogs/enterprise/mcafee-enterprise-atr/log4shell-vulnerability-is-the-coal-in-our-stocking-for-2021/.

[9] M. Vidoni, "A systematic process for mining software repositories: Results from a systematic literature review," *Information and Software Technology*, vol. 144, p. 106 791, Apr. 2022, ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106791. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921002317.

[10] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: From package-based to call-based dependency networks," *Empirical Software Engineering*, Jan. 2021. [Online]. Available: http://arxiv.org/abs/2101.09563.

[11] T. Preston-Werner, *Semantic versioning 2.0.0*, Jul. 2013. [Online]. Available: https://semver.org/spec/v2.0.0.html.

[12] L. Page, "U.s. patent us7058628b1," Jun. 2006. [Online]. Available: https://patents.google.com/patent/US7058628B1/en.

[13] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381–416, 1 Feb. 2019, ISSN: 1382-3256. DOI: 10.1007/s10664-017-9589-y. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9589-y.

[14] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.

[15] Y. Wang, Y. Yuan, Y. Ma, and G. Wang, "Time-dependent graphs: Definitions, applications, and algorithms," *Data Science and Engineering*, vol. 4, pp. 352–366, 4 Dec. 2019, ISSN: 2364-1185. DOI: 10.1007/s41019-019-00105-0. [Online]. Available: http://link.springer.com/10.1007/s41019-019-00105-0.

[16] A. Purcaru, "Analyzing the effect of introducing time as a component in python dependency graphs," 2022.

[17] M. L. LongJason and Zhang, *Edge betweenness centrality*, Olaf, C. Kwang-Hyun, Y. H. D. Werner, and Wolkenhauer, Eds., 2013. DOI: 10.1007/978-1-4419-9863-7_874. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_874.

[18] *Statistical correlation*. [Online]. Available: https://mathworld.wolfram.com/StatisticalCorrelation.html.

[19] A.J.M Brands, *NPM Packages (∼1.9M packages)*, Zenodo, Jun. 2022. DOI: 10.5281/zenodo.6631536. [Online]. Available: https://doi.org/10.5281/zenodo.6631536.

A Sample of input data

The full input data can be found at Zenodo [19]. Figure 5 shows an overview of what this input data looks like.

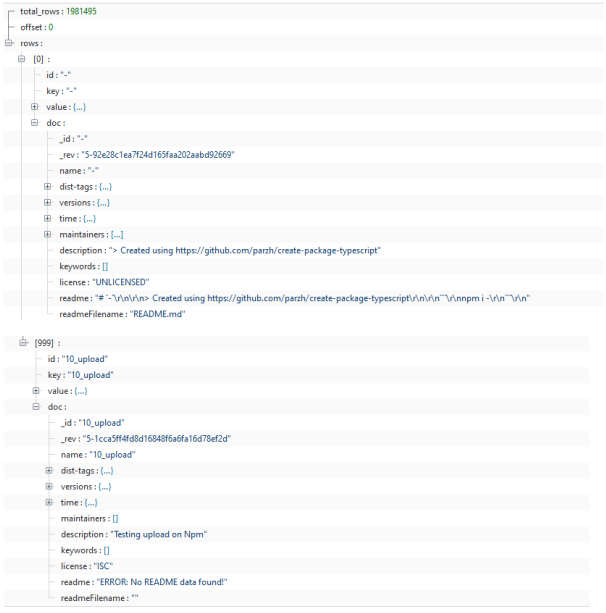

Figure 5: A sample of what the input data [19] looks like

B Data Processing and Graph generation code

The graph generation and analysis code can be found at https://github.com/AJMBrands/SoftwareThatMatters-NPM in the main branch. On the same github repository, the code to preprocess the data can be found and the preprocessed data can be found at Zenodo [19]. This data looks like a list of records that resemble figure 6. Both the data processing code and the graph generation code include a `README` file explaining how to use them.


```
{
  "name": "react",
  "versions": {
    "1.00": {
      "timestamp": "06-05-2022T10:00:01",
      "dependencies": {
        "name": "^1.0.2"
      }
    }
  }
}
```


Figure 6: Example record for processed data

C Resulting Graphs

In figure 7, we can see what kind of sub networks the package dependency network for the first 10000 packages creates.

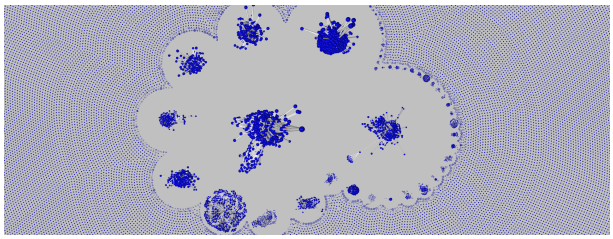

Figure 7: Package graph for 10000 packages (approx. 10 thousand unique package versions and 2 million edges). The blobs represent groups of packages that all transitively depend on the same dependency. The nodes outside the main blobs are disconnected.

Index	Package	Version	PageRa
0	@babel/core	7.18.0	0.031577
1	@babel/preset-env	7.18.0	0.011795
2	@babel/types	7.18.0	0.011650
3	@babel/helper-validator-identifier	7.16.7	0.011405
4	@babel/parser	7.18.0	0.009756
5	@babel/helper-fixtures	7.17.10	0.008027
6	@babel/helper-plugin-utils	7.17.12	0.007966
7	@babel/helper-transform-fixture-test-runner	7.18.0	0.007890
8	@babel/runtime	7.18.0	0.007851
9	@babel/code-frame	7.16.7	0.007636
10	@babel/helper-plugin-test-runner	7.16.7	0.006647
11	@babel/generator	7.18.0	0.006597
12	@babel/highlight	7.17.12	0.006551
13	@babel/cli	7.17.10	0.005239
14	@babel/traverse	7.18.0	0.005146
15	@babel/preset-react	7.17.12	0.003888
16	@babel/helper-check-duplicate-nodes	7.17.9	0.003781
17	@babel/plugin-transform-modules-commonjs	7.18.0	0.003354
18	@babel/template	7.16.7	0.003346
19	@babel/preset-typescript	7.17.12	0.002927
20	@babel/helper-module-transforms	7.18.0	0.002675
21	@babel/plugin-proposal-class-properties	7.17.12	0.002662
23	@babel/helper-compilation-targets	7.17.10	0.002525
25	@babel/helpers	7.18.0	0.002420
27	@babel/plugin-transform-runtime	7.18.0	0.002012
32	@babel/register	7.17.7	0.001680
33	@babel/helper-validator-option	7.16.7	0.001668
46	@babel/eslint-parser	7.17.0	0.001095
48	@babel/plugin-proposal-object-rest-spread	7.18.0	0.001047
49	@babel/polyfill	7.12.1	0.000962
51	@babel/helper-split-export-declaration	7.16.7	0.000913
52	@babel/helper-environment-visitor	7.16.7	0.000892
53	@babel/plugin-syntax-dynamic-import	7.8.3	0.000863
56	@babel/plugin-transform-react-jsx	7.17.12	0.000817
57	@babel/compat-data	7.17.10	0.000817
58	@babel/helper-create-class-features-plugin	7.18.0	0.000799
63	@babel/plugin-transform-arrow-functions	7.17.12	0.000696
68	@babel/plugin-proposal-optional-chaining	7.17.12	0.000657
69	@babel/plugin-syntax-object-rest-spread	7.8.3	0.000652
71	@babel/helper-simple-access	7.17.7	0.000649
72	@babel/helper-function-name	7.17.9	0.000633
73	@babel/helper-module-imports	7.16.7	0.000629
74	@babel/plugin-transform-typescript	7.18.1	0.000629
76	@babel/plugin-syntax-class-static-block	7.14.5	0.000585
78	@babel/plugin-external-helpers	7.17.12	0.000537
80	@babel/node	7.17.10	0.000532
81	@babel/plugin-proposal-decorators	7.17.12	0.000525
84	@babel/helper-hoist-variables	7.16.7	0.000523
91	@babel/runtime-corejs3	7.18.0	0.000463
92	@babel/plugin-syntax-jsx	7.17.12	0.000462
94	@babel/plugin-transform-react-jsx-development	7.16.7	0.000453
95	@babel/preset-flow	7.17.12	0.000444
96	@babel/plugin-transform-react-display-name	7.16.7	0.000438
99	@babel/plugin-transform-react-pure-annotations	7.18.0	0.000421

Figure 8: Babel entries in top-100 PageRank from 01-01-2019 to 01-01-2023