Study of Bugs and Popularity in NPM Ecosystem Using Dependency Graph

Mohammadreza Saeidi
University of British Columbia (Okanagan)
BC, Canada
ariasd@student.ubc.ca

Abstract—The reliance on third-party libraries (packages) within Software Engineering has become a prevailing trend due to their ability to significantly reduce implementation efforts. Software ecosystems like NPM play a pivotal role in storing and facilitating the integration of these packages into various projects. The presence of an extrinsic bug within such an ecosystem holds the potential to destabilize the entire ecosystem. In this project, we collect a dataset containing dependency information from 12,050 NPM packages. Leveraging this dataset, we construct a dependency graph and apply graph metrics such as degree centrality, eigenvector centrality, closeness centrality, and betweenness centrality. These metrics allow us to identify critical packages, assess their susceptibility to bugs, and delineate strategies to impede bug propagation within the NPM ecosystem. Our analysis reveals that specific packages exhibit heightened popularity, vulnerability to bugs, and play crucial roles in maintaining ecosystem stability.
Keywords: Software Ecosystem, NPM, Extrinsic Bug, Dependency Graph

1 Introduction

The reliance on third-party libraries (packages) is a prevalent trend in Software Engineering due to the significant reduction in implementation efforts they offer. These packages are often adopted in open-source projects and provide essential functionalities that would otherwise require extensive development time [1].

Central to the efficient use of third-party packages are Software Ecosystems or package managers, which act as repositories and facilitators for these packages. Among these ecosystems, the Node Package Manager (NPM) stands out as a pivotal element in the JavaScript community, hosting a vast repository of packages essential for developing JavaScript-based applications. The role of such ecosystems in software development is monumental, serving as a centralized hub where developers access, contribute, and integrate these packages seamlessly into their projects [2].

However, the use of third-party packages is not without challenges. Bugs in these packages can cause significant issues in the main application. They can lead to software crashes, data corruption, and even expose security vulnerabilities [1]. Consequently, the stability of an ecosystem, characterized by having fewer bugs, is of utmost importance. It ensures the reliability of the software built within this ecosystem and reduces the risks associated with the adoption of third-party packages.

To comprehend the nature of bugs within an ecosystem like NPM, it is vital to distinguish between intrinsic and extrinsic bugs. Intrinsic bugs refer to issues stemming from the core functionalities of the software itself, while extrinsic bugs, our focal point in this study, are those introduced through external dependencies, such as packages integrated into a project [3]. These extrinsic bugs often pose unique challenges as they are reliant on the maintenance and updates of the external packages, thereby affecting the overall stability and robustness of the software ecosystem.

The use of a dependency graph is an effective way to identify critical packages within an ecosystem. It provides a visual representation of how different software packages depend on each other. While numerous studies have utilized dependency graphs to assess the security of the NPM ecosystem [2], [4], [5], there remains a dearth of research focusing on the ecosystem's stability. In this project, we try to use a dependency graph to identify packages that, if they contain bugs, could potentially affect a large number of other packages, thereby threatening the stability of the entire ecosystem. Thus, dependency graphs can serve as a useful tool for prioritizing bug-fixing efforts in order to maintain the stability and reliability of the software ecosystem. Furthermore, in scenarios where a bug is introduced into the ecosystem, utilizing a dependency graph can facilitate the implementation of an effective strategy to prevent the spread of the bug.

This project aims to create a directed graph representing the relationships between the most popular NPM packages and their dependencies. By using graph metrics, the study will provide insights into the ecosystem's structure and dynamics.

The rest of this report is organized as follows. In Section 2, the research questions of this project will be discussed. Section 3 describes the data collection process. Section 4 introduces our study design. Section 5 reports the findings of our study. Section 6 discusses the threats to our study and future works. Section 7 draws the conclusions.

2 Research Questions

The research questions we aim to address in this project are outlined below:

- **RQ1:** *Which graph metric can be used to measure the popularity of packages within the NPM ecosystem?*

This question focuses on identifying and ranking the packages that enjoy the highest usage and adoption rates within the NPM ecosystem. Understanding the popularity of these packages provides insights into their widespread use among developers.

- **RQ2:** *Which NPM packages are more susceptible to extrinsic bugs?*
  This research question aims to delve into the vulnerability of software packages within the NPM ecosystem to extrinsic bugs. Understanding which packages are more prone to these types of bugs is crucial for assessing their resilience and stability within the ecosystem.
- **RQ3:** *Which packages are the most critical for the stability of the NPM ecosystem?*
  This question delves into pinpointing the packages that play the most pivotal roles in maintaining the stability and robustness of the NPM ecosystem. Identifying these critical packages helps in prioritizing efforts to fortify the overall stability of the system.
- **RQ4:** *What graph measures can be taken to halt bugs propagation?*
  This question aims to explore potential strategies and interventions that can effectively impede the spread of bugs within the ecosystem, thereby mitigating their impact on system stability and reliability.

3 Data Collection

The initial phase of this project involves collecting a suitable dataset to construct a dependency graph within the NPM ecosystem. To accomplish this, data is gathered through the API provided by libraries.io. This API supplies valuable information concerning the relationships and dependencies among various NPM packages.

Access to this API necessitates an API key. One of the challenges of using this API is the rate limit imposed on each API key, allowing only 60 requests per minute. To address this constraint, we acquire five API keys and alternate between them, effectively preventing any single key from reaching its rate limit.

Given the vast number of NPM packages, exceeding three million [6], attempting to create a comprehensive graph using R is impractical. Consequently, our focus is directed solely on a small component of the NPM packages containing the top 500 most frequently utilized NPM packages and their dependencies. The specific list of these top packages is obtained from [7].

The data collection process starts with these 500 packages, which are systematically queued. We iteratively process this queue until it is empty. During each iteration, a package is dequeued, and its dependencies are extracted using the libraries.io API. These newly discovered packages are then added to the queue to retrieve their respective dependencies. This data collection process reveals a distinct component within the NPM ecosystem comprising interconnected members. These members exhibit strong connections among themselves but do not have any connections with other packages outside this specific component within the NPM ecosystem.

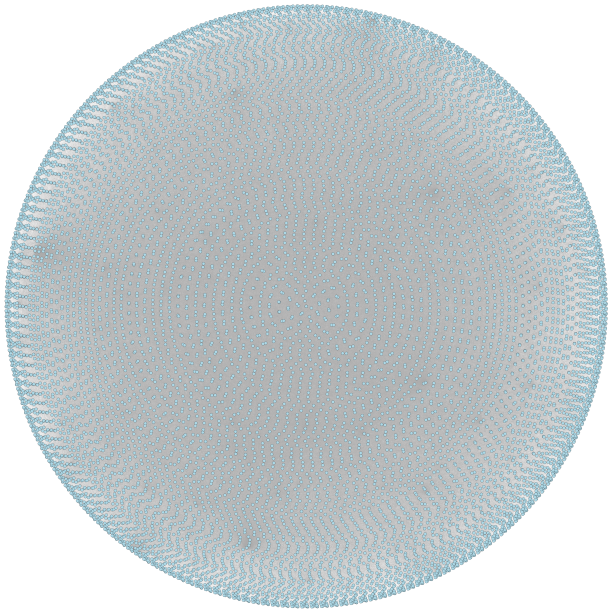

Fig. 1: Plot of the dependency graph for a community in the NPM ecosystem.

It is important to note that not all dependencies associated with a package are available within the NPM ecosystem. In instances where a package can not be found in NPM, the API returns an error indicating the absence of the requested package. As our project is exclusively focused on the NPM ecosystem, we disregard this error and remove the package from the queue.

Overall, a total of $14,611$ packages were fetched, out of which $12,050$ belonged to the NPM ecosystem. This selection of packages constitutes our initial dataset, representing the nodes within our graph. Additionally, a dataset outlining the relationships among these $12,050$ packages (edges) comprises $102,279$ records. The source code for this phase of the project can be accessed here, and the resulting datasets are available here.

4 Methodology

The results from the data collection phase yield two datasets, representing the vertices and edges essential for constructing the dependency graph intended for this project. This graph is a directed graph, where vertices denote individual NPM packages, and edges denote the dependencies existing between these packages. Specifically, a directed edge originating from vertex A and pointing towards vertex B signifies that package A relies on package B as one of its dependencies. Our analysis and construction of this graph is executed using the R programming language in conjunction with the "igraph" library. The source code of this project can be found here.

Upon importing these datasets into R, we generate a graph. A visual representation of this graph is depicted in Figure 1.

The graph comprises $12,050$ vertices and $102,279$ edges. Despite the extremely low density of this graph, standing

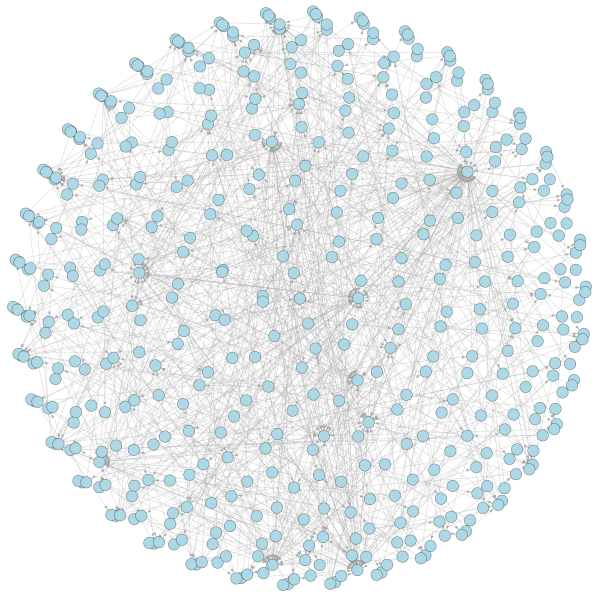

Fig. 2: Plot of the dependency graph for the first 500 popular packages within the NPM ecosystem.

at a mere 0.07%, the visual representation appears convoluted, rendering it difficult to extract meaningful insights. This visual complexity arises from the abundance of edges within the graph, compounded by the substantial number of vertices. Nevertheless, amidst this complexity, discernible clusters or spots are noticeable. These spots signify certain packages exhibiting notably higher degrees of connectivity compared to others.

For enhanced visualization of the graph, a subgraph has been generated, comprising solely the top 500 most popular packages within the NPM ecosystem and their relationships. This specific subgraph is illustrated in Figure 2.

4.1 Popularity (RQ1)

Degree Centrality stands as a fundamental metric addressing the first and second research questions (RQ). For RQ1, which focuses on pinpointing the most popular packages, assessing the in-degree centrality of nodes within the NPM dependency graph is crucial. In this project, the in-degree represents the count of dependents that a package has. In other words, this metric illuminates the extent to which packages are relied upon by a multitude of others as essential dependencies, effectively indicating their popularity within the NPM ecosystem. Higher in-degree centrality for an NPM package implies that numerous other packages rely on it as a dependency, signifying its popularity within the ecosystem.

4.2 Vulnerability to extrinsic bugs (RQ2)

In contrast, for RQ2 aimed at identifying the packages most susceptible to extrinsic bugs, out-degree centrality emerges as a pertinent metric. In this project, the out-degree of a package represents the quantity of dependencies that the package possesses. Analyzing out-degree centrality could reveal packages that are more intricate and have numerous dependencies. A higher out-degree suggests that the package relies on multiple other packages to function properly. This complexity elevates the package's susceptibility to extrinsic bugs originating from other interconnected packages within the ecosystem, highlighting its vulnerability to external factors.

4.3 Stability of the NPM (RQ3)

In the NPM ecosystem, packages rely extensively on each other to function properly. The mean out-degree within the dependency graph illustrates the average number of other packages that each individual package in the NPM ecosystem depends on for its operations. In the created graph, this mean out-degree, calculated at 8.49, indicates that, on average, each package relies on approximately 8.49 other packages to operate effectively. Similarly, the mean in-degree of 8.49 for this graph suggests that, on average, each package in the NPM ecosystem is relied upon by approximately 8.49 other packages.

This reciprocal dependency nature becomes significant when considering the potential propagation of bugs within the ecosystem. If a particular NPM package contains a bug, it may affect at least 8.49 other packages directly dependent on it. Furthermore, given that these 8.49 dependent packages, on average, also have 8.49 dependencies themselves, a bug originating from one package has the potential to impact a cascade of interconnected packages. This scenario underscores the concept of Bug Propagation in a software ecosystem, wherein the presence of a bug in a single package poses a risk to the entire ecosystem's stability.

To identify critical packages that might threaten the stability of the ecosystem, metrics like eigenvector centrality and closeness centrality prove useful. Eigenvector centrality measures a node's influence in a network, considering not just the number of connections but also the quality of those connections [8]. In the context of the NPM ecosystem, packages with higher eigenvector centrality are those that are not only extensively connected but are connected to other influential packages, thereby potentially propagating bugs more significantly through the ecosystem.

Closeness centrality gauges how quickly a node can interact with other nodes in a network [8]. In the NPM ecosystem, packages with higher closeness centrality are those that are more readily interconnected with other packages, meaning they could potentially propagate bugs more efficiently due to their swift communication with a large number of other packages.

Both these metrics offer insights into the criticality of packages concerning bug propagation. By identifying packages with higher eigenvector centrality and/or closeness centrality, we can pinpoint those whose vulnerabilities might have a more profound impact on the overall stability and reliability of the ecosystem due to their potential to propagate bugs efficiently. Identifying these critical packages becomes pivotal in managing and securing the ecosystem against potential cascading bugs.

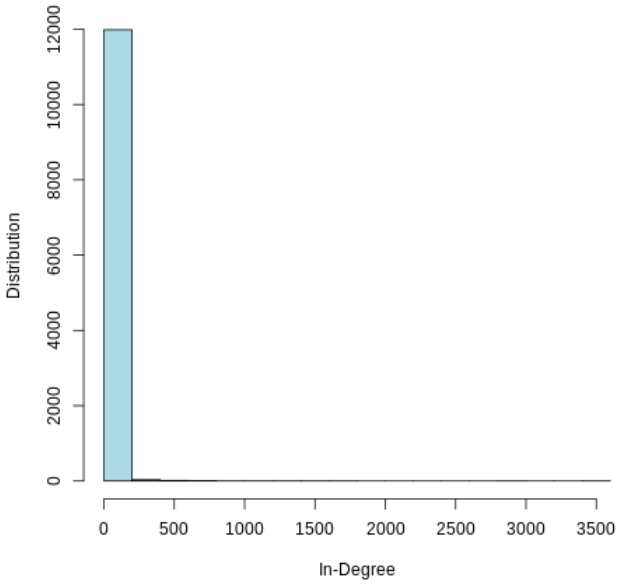

Fig. 3: In-degree distribution of the graph.

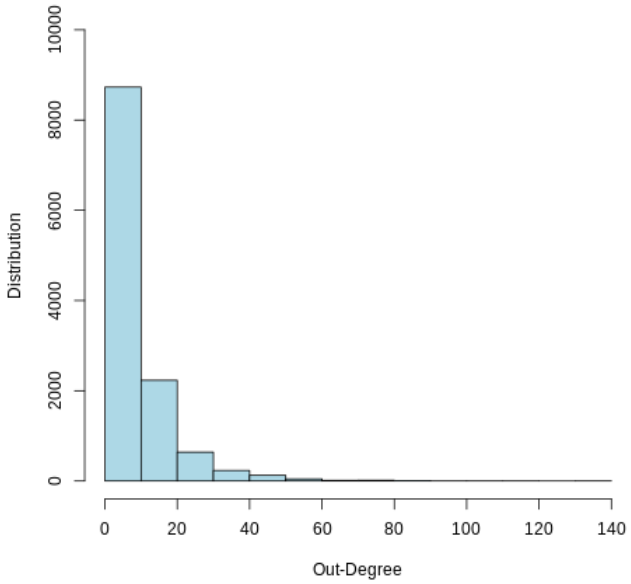

Fig. 4: Out-degree distribution of the graph.

4.4 Halting bugs propagation (RQ4)

The previous section discussed how a bug can potentially spread within an ecosystem, especially if it is introduced into a critical package. To prevent such bugs from circulating, one effective approach is to identify a specific package and temporarily halt the release of new versions until the bug in the critical package is fixed. One way to pinpoint these key packages is by using a metric called betweenness centrality, a measure within network analysis that assesses the significance of a node (in this context, a package) in terms of its influence on the flow of information or activity between other nodes [8]. Packages with high betweenness centrality act as crucial connectors in the ecosystem, influencing the transmission of code dependencies. By strategically using betweenness centrality metrics, software maintainers can identify these influential packages and implement tactics, like temporary release halts, to reduce the risks of bug spread within the software ecosystem.

5 Results

5.1 Degree distribution

Given that the graph is directed, it becomes essential to analyze both the in-degree and out-degree distributions. Figure 3 illustrates the histogram of the in-degree distribution, providing a visual representation of the occurrence frequency of various in-degrees within the graph.

Observing the histogram, it becomes evident that the majority of nodes or packages possess an in-degree ranging between 0 and 250. However, a select few packages stand out, showcasing notably higher in-degrees. This distribution pattern suggests a characteristic of a heavy-tailed distribution, typical in Power Law distributions, where a small number of nodes possess significantly higher degrees compared to the majority [8]. To empirically investigate whether the in-degree distribution conforms to a power law distribution, a fitting analysis is performed on the in-degree data. The results of this analysis reveals the following insights:

- **alpha:** The estimated exponent of the power law distribution is approximately 1.89. This parameter indicates the slope of the power law distribution. A value closer to 2 suggests a scale-free network (typical for power law distributions).
- **KS.stat:** The Kolmogorov-Smirnov (KS) statistic tests the goodness of fit between the empirical distribution and the power law distribution. The smaller the value, the better the fit. Here, the KS statistic is 0.01260073.
- **KS.p:** The p-value associated with the KS test. If this value is greater than a chosen significance level (e.g., 0.05), it suggests that the data is consistent with the fitted distribution. Here, the p-value is 0.5144118, which is greater than 0.05, indicating that the data is not significantly different from a power law distribution.

Therefore, the in-degree distribution of this graph seems to exhibit characteristics of a power law distribution, as suggested by the calculated exponent and the KS test results.

We examine the out-degree distribution using a similar approach as the in-degree analysis. Figure 4 illustrates a histogram showcasing out-degree distribution in the graph.

Upon observation, the out-degree distribution appears more evenly spread across various values compared to the in-degree distribution. This diversity suggests that the out-degree distribution might not align well with a power law distribution. To confirm this, we fit the out-degree distribution to a power law model. The analysis results are summarized as follows:

Package Name	In-degree
mocha	3,588
eslint	2,845
typescript	1,720
prettier	1,468
tape	1,313
chai	1,262
nyc	1,043
rimraf	978
xo	845
jest	784

TABLE 1: 10 packages with the highest in-degree.

Package Name	Out-degree
testcafe	138
antd	137
ember-cli	114
ember-cli-addon-docs	103
mocha	92
next	91
prettierx	89
rc-tools	89
postcss-cssnext	89
parcel-bundler	87

TABLE 2: 10 packages with the highest out-degree.

- **alpha:** The estimated exponent alpha is 3.6 which is higher than 2. This is often indicative of a distribution with a rapidly decaying tail.
- **KS.stat:** Here, the KS statistic is 0.04760682.
- **KS.p:** The KS test's p-value is 0.0222946 which is quite small (less than 0.05), suggesting that the out-degree distribution significantly deviates from a power law distribution.

Based on these results, the out-degree distribution of this graph might not conform well to a strict power law distribution. The higher exponent value and the significant KS test p-value indicate a potential deviation from a pure power law distribution.

5.2 Popularity (RQ1)

As highlighted earlier, a package's popularity is associated with its in-degree. By computing the in-degree values of the given packages and arranging them in descending order, we identify the top 10 popular packages within the NPM ecosystem. These leading packages are shown in Table 1.

These packages hold the highest in-degree values, signifying their substantial usage within the NPM ecosystem.

5.3 Vulnerability to extrinsic bugs (RQ2)

As the out-degree of a package grows, its resilience against extrinsic bugs tends to decrease. By computing the out-degree values of the provided packages and arranging them in descending order, we identify the first 10 packages exhibiting higher vulnerability to external bugs within the ecosystem. These packages are shown in Table 2.

These top 10 packages, listed in descending order based on their out-degree values, are presumed to be more susceptible to extrinsic bugs.

Package Name	Eigenvector Centrality
eslint	1.000
mocha	0.999
nyc	0.653
chai	0.546
prettier	0.545
rimraf	0.537
typescript	0.536
glob	0.413
sinon	0.399
husky	0.356

TABLE 3: 10 packages with the highest eigenvector centrality.

Package Name	Closeness Centrality
mocha	0.549
eslint	0.533
prettier	0.442
chai	0.441
typescript	0.436
nyc	0.433
rimraf	0.430
glob	0.420
sinon	0.406
coveralls	0.402

TABLE 4: 10 packages with the highest closeness centrality.

5.4 Stability of the NPM (RQ3)

As outlined in the previous section, closeness centrality and eigenvector centrality serve as tools to pinpoint packages crucial to the stability of the NPM ecosystem. We identify the top 10 packages with the highest eigenvector centrality (Table 3) and the top 10 with the highest closeness centrality (Table 4).

It is imperative to scrutinize pull requests submitted to these identified packages meticulously because any introduced bugs in these critical packages have the potential to pose a serious threat to the entire ecosystem.

5.5 Halting bugs propagation (RQ4)

It was previously mentioned that betweenness centrality serves as a fitting metric to impede the spread of bugs. In Table 5 We pinpoint 10 packages that could be selected when a bug infiltrates the ecosystem.

If a bug is introduced into the NPM ecosystem through one of the critical packages, temporarily halting the release of new versions for each of these packages becomes a viable strategy to prevent the bug from spreading further.

Package Name	Betweenness Centrality
mocha	43,802,652
eslint	23,163,026
tape	10,333,324
webpack	9,140,590
karma-sauce-launcher	9,063,381
rollup	8,278,072
global-agent	7,560,863
anyproxy	7,366,999
memfs	6,473,824
antd	6,326,511

TABLE 5: 10 packages with the highest betweenness centrality.

6 Threats to Validity and Future Works

In our study, we looked at only the most popular component of the NPM ecosystem. This means the findings might not apply to the entire system. To improve our research in the future, we plan to gather data on all NPM packages and create a complete picture of how they depend on each other.

Sometimes, certain packages in our study did not have all their dependencies listed in the NPM ecosystem. This could have caused some minor errors in our results. To make our findings more accurate next time, we aim to collect information about these missing dependencies. This will help us avoid mistakes and make our research stronger.

7 Conclusion

Our study focused on extrinsic bugs (bugs introduced through external dependencies) in the NPM ecosystem to understand their impact on the ecosystem's stability. Our methodology involved constructing a dependency graph and utilizing graph metrics such as degree centrality, eigenvector centrality, closeness centrality, and betweenness centrality. We found that certain packages exhibited higher popularity, vulnerability to bugs, and criticality in maintaining ecosystem stability. We also outlined strategies to halt bug propagation within the NPM ecosystem, emphasizing the importance of these findings in maintaining the ecosystem's stability.

References

[1] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[2] 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)*, 2019, pp. 995–1010.

[3] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, vol. 25, pp. 1294–1340, 2020.

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

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

[6] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "npm-follower: A complete dataset tracking the npm ecosystem," *arXiv preprint arXiv:2308.12545*, 2023.

[7] S. Wattanakriengkrai, R. G. Kula, C. Treude, and K. Matsumoto, "Lessons from the long tail: Analysing unsafe dependency updates across software ecosystems," *arXiv preprint arXiv:2309.04197*, 2023.

[8] M. Newman, *Networks*. Oxford university press, 2018.