npm-miner: An Infrastructure for Measuring the Quality of the npm Registry

Kyriakos C. Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas L. Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece
{kyrcha, mpapamic, thdiaman, michael.tsapanos}@issel.ee.auth.gr, asymeon@eng.auth.gr

ABSTRACT

As the popularity of the JavaScript language is constantly increasing, one of the most important challenges today is to assess the quality of JavaScript packages. Developers often employ tools for code linting and for the extraction of static analysis metrics in order to assess and/or improve their code. In this context, we have developed npn-miner, a platform that crawls the npm registry and analyzes the packages using static analysis tools in order to extract detailed quality metrics as well as high-level quality attributes, such as maintainability and security. Our infrastructure includes an index that is accessible through a web interface, while we have also constructed a dataset with the results of a detailed analysis for 2000 popular npm packages.

KEYWORDS

static analysis, software quality, javascript, npm

1 INTRODUCTION

A popular meme in the JavaScript (JS) community is that of Atwood, stating that 'Any application that *can* be written in JavaScript, *will* eventually be written in JavaScript'[1]. And, indeed, the popularity of JS is increasing, with the explosion of frameworks for building server (Node.js), web (React, Vue.js, Angular, etc.), desktop (Electron), mobile applications (React Native, NativeScript, etc.), even IoT solutions (Node-RED) predisposing for larger growth. Indicatively, a 2017 RedMonk survey[2] places JS at the top position with respect to the combination of GitHub pull requests and number of tags in Stack Overflow questions, while Module Counts[3] depicts an exponential growth of npm modules (the package manager of JS) against module repositories of other programming languages. The npm repository is often seen as one of the JS revolutions[4] that brought JavaScript from 'a language that was adding programming capabilities to HTML' into a full-blown ecosystem, with such a rapid growth that terms like 'JS (framework) fatigue' have become common among JS developers[13]. Obviously, such a growth of the npm registry will have us conjecture that the quality and usefulness of the packages will follow a Power Law[5] or the Pareto Principle[6], making the identification of 'good' packages an essential task.

From a software development perspective, the use of linting tools and static analysis metrics for software quality monitoring has become a state-of-the-practice procedure, given the fact that they provide valuable information regarding numerous source code issues and vulnerabilities [8, 9, 14]. A major advantage of automated static analysis tools (ASAT) is that they enable discovering crucial refactoring opportunities from the very first stage of the software development process, where less man effort/development cost is required [7]. According to recent studies [3], this fact is even more prominent in dynamically-typed languages like JS, which are considered to be highly error-prone, given that they do not require programmers to follow any strict code-typing discipline and thus are subject to accumulate technical debt in the form of hidden bugs [11].

Against this background, we have built `npm-miner`, a platform that continuously crawls the npm registry, analyzes the quality of its packages against major quality characteristics such as maintainability and security [6], and provides a search engine for developers to query for specific npm packages and receive informative responses. In addition, we have constructed a dataset with raw analysis results for the 2000 most popular npm packages.

2 NPM-MINER ARCHITECTURE AND TOOLS

The complete `npm-miner` architecture is depicted in Figure 1. The basic components of the platform are discussed next.

Data management layer. The database management system employed is a CouchDB[7] and hosts two databases: a) one that mirrors the npm's registry database through continuous replication (this way always having an updated snapshot of npm) and b) one that stores the analysis results for each one of the processed packages. In order to retrieve different representations of the database such as a sorted list of GitHub repositories by the number of stars or statistics about maintainability index results, CouchDB's map-reduce design views are used.

Workers. Two types of worker processes have been designed for the system: the crawler and the analyzer process. The crawler process is responsible for retrieving a batch of 500 packages from the npm registry, at periodic intervals (1 hour) and publishing the equivalent number of analyzing tasks to a message queue (RabbitMQ). The analyzer processes continuously monitor the queue for new messages, retrieve a message that contains information on a specific package, download its source code and analyze it using the JavaScript Static Analyzer (JSSA) module.

JSSA. The JSSA module is responsible for running five open source static analysis tools. These are:

[1] https://blog.codinghorror.com/the-principle-of-least-power/
[2] http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/
[3] http://www.modulecounts.com/
[4] https://youtu.be/L-fx2xXSVso
[5] https://en.wikipedia.org/wiki/Power_law
[6] https://en.wikipedia.org/wiki/Pareto_principle
[7] http://couchdb.npm-miner.com:5984/_utils

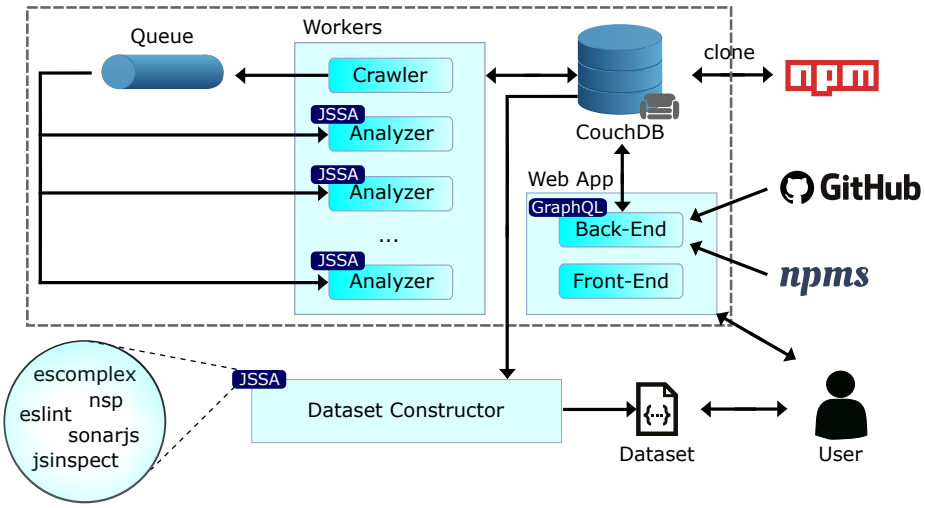

Figure 1: `npm-miner` architecture overview.

- `eslint`[8] (and a security plugin[9]), for finding linting issues.
- `escomplex`[10], for measuring complexity.
- `nsp` (node security package)[11], for identifying vulnerabilities with dependencies.
- `jsinspect`[12], for finding duplicate code snippets.
- `sonarjs`[13], for applying rules the 'Sonar way'.

Web application. The `npm-miner` web application[14] is responsible for allowing the user to search for a package and in return, displays information about the package with respect to its quality and popularity. The web application retrieves information from three different resources, the REST API of the `npm-miner`'s Database, which contains aggregated results of the analyses, the REST API of GitHub, for up-to-date popularity metrics like stars, forks and watchers, and the REST API of npms.io, another search engine of npm that contains alternative quality, popularity and maintenance metrics. The aggregation of the requests and responses is performed through a GraphQL back-end. GraphQL ensures extensibility, allowing easy integration of more external sources.

Dataset Constructor. The Dataset Constructor is responsible for recreating the dataset presented in this paper. Given as input a list of packages (specified in Section 4), the Dataset Constructor retrieves their tarballs and runs the complete suite of static code analyses. The list can obviously be modified to download other npm packages and perform the same analyses.

3 NPM CRAWLING

In an open source and popular ecosystem like that of JavaScript and the npm registry, there exist many problems that make crawling and processing challenging, especially if one would like to follow a systematic and pristine analysis process. Some of the typical problems encountered with the npm registry are: a) The declared GitHub URL of a certain npm package leads to a not found page, b) The declared GitHub URL of a certain npm package redirects to a different (to the one declared) GitHub page (probably this is a maintenance issue of the `package.json` file), c) many npm packages contain copied-pasted popular open source projects with only the package name changed in the `package.json` file, d) many npm packages share the same GitHub repository and finally e) sometime the JSSA fails or does not complete due to large or unexpected input. The decisions we made in order to address these issues are presented below:

- *We store only the analysis of the latest version of each package.* This decision was made in order to avoid the exponential growth of data in our infrastructure, which would have sidetracked our research prototype into an industrial grade big data solution. However, as the infrastructure becomes more mature, we intend to include evolution analytics of the packages.
- *We analyze only packages that have declared a GitHub repository in their package.json file*, since we wanted to be able to connect popularity (stars, forks, pull-requests) and quality metrics for research purposes. In addition, this helps us filter out packages that their declared repository in GitHub returns a 404 HTTP error (probably due to the repository being deleted or made private), driving us to the next decision.
- *Packages whose GitHub repository was not found (404 error) were filtered out from npm-miner.*
- *If there is a redirection after following the GitHub URL, the inconsistency is detected and logged but the analysis continues.* The inconsistency is detected by a mismatch between the GitHub URL detected in the package.json and the final GitHub URL in terms of user and repository names. We continue with the analysis, since the package might have changes and the package.json might have not been updated, however still this constitutes a quality error.

[8] https://eslint.org
[9] https://github.com/nodesecurity/eslint-plugin-security
[10] https://github.com/escomplex/escomplex
[11] https://github.com/nodesecurity/nsp
[12] https://github.com/danielstjules/jsinspect
[13] https://github.com/SonarSource/sonarjs-cli
[14] http://npm-miner.com

- *We exclude from the analysis the* `node_modules` *and* `dist` *directories, which usually contain dependencies and built code.*
- *The linting package eslint is run with the same set of rules for all packages, even if the package had its own linting file.* We used the recommended eslint rules[15]. This decision was made in order to have a basis for comparing the packages with respect to linting issues. In future work we plan to include both analyses.

The `npm-miner` provides the following high-level information and metrics for each npm package analyzed:

(1) The `package.json` of the package.
(2) The complete analysis of npms.io
(3) The star count of the GitHub repository
(4) File and directories statistics: number of files, minimum, maximum and sum of directory depths for directories where `.js` files are found.
(5) Error and warning counts of `eslint`.
(6) Aggregate complexity measures for the whole package using the `escomplex` package. More specifically we preserve package-level measurements for first order density, change cost, core size, average lines of code, cyclomatic complexity, halstead effort, parameters, maintainability index, total lines of physical code and total lines of logical code.
(7) Number of vulnerabilities found in package dependencies by the `nsp` package.
(8) Number of linting errors in terms of security rules based on the `eslint-security-plugin`.
(9) Number of code duplicates based on `jsinspect`.
(10) Number of rule violations based on `sonarjs`.

An example of such a record can be found at: http://couchdb.npm-miner.com:5984/_utils/document.html?npm-packages/mapbox.

At the moment of writing this paper, the npm registry has 598,862 packages (stored in a 21.5GB database). Having applied our filtering rules, the `npm-miner` has analyzed 279,342 packages so far (stored in a 9.2GB database), with the total lines of code analyzed so far being around 356 million. Given the `npm-miner` Workers' node capabilities, there is a processing throughput of around 0.3 packages per second.

Table 1 summarizes certain statistics for some of the quality metrics employed. Extreme values are attributed mainly to the openness and popularity of the registry and are handled appropriately.

Table 1: Statistical analysis of npm-miner findings

Metric	Min	Max	Mean	Std
Cyclomatic complexity	1	288,5	1,67	0.62
Maintainability index	-119.33	171	125	16
eslint issues	0	766,991	181.63	2,780
nsp issues	0	162	0.52	2.70
Number of files	1	999	7.58	27
Number of lines	1	2,506,278	1,577	15,456.5
Number of dependencies	0	652	4.01	7.39

[15]https://eslint.org/docs/rules/

4 DATASET

Our dataset consists of 2000 packages residing in the top-starred GitHub repositories. Upon performing static analysis (as mentioned in Section 3), we discovered that many npm packages are merely copy-paste of entire repositories of popular JS projects. To filter out such packages and thus refrain from having duplicates and false-positives in terms of high popularity, we have filtered out any packages that have not yielded at least 1000 downloads from npm in the last month.

As already mentioned, static analysis is performed on our dataset using five different tools that provide the values of various metrics, such as the widely used maintainability index (MI), along with different coding violations grouped by their severity (e.g. minor, major, critical etc.). We also perform duplicate code checking and security analysis based on the dependencies of every project.

Table 2 provides aggregate information regarding the dataset, while Figure 2 illustrates its structure. The dataset contains the analysis results of the 2000 packages organized in chunks of 100 packages. The results are given in `json` format and are organized in five different files, each corresponding to the output of the employed tool. The dataset also contains information for each package (`packages_info`) extracted from CouchDB (see Section 3). The dataset along with the corresponding code repository can be found online.[16][17]

Table 2: Dataset Statistics

Metric	Value
Number of npm packages	2,000
Number of .js files analyzed	56,416
Lines of code analyzed	3,216,244
Total errors found	476,044
Total warnings found	279,046
Total size	112GB (1.15GB compressed)

5 IMPACT AND RESEARCH DIRECTIONS

Our dataset can be used to confront several interesting challenges in current research. At first, given that it comprises a large set of static analysis metrics, it could be useful for building and calibrating quality assessment models [2, 12]. Further research can also be performed on the reusability of npm packages as perceived by developers. As the dataset offers information about the number of stars and forks as well as the number of npm package downloads, it can be used to investigate the correlation between package popularity and quality [10], or even determine which are the quality characteristics that constitute a package popular and more downloadable by the community. In this context, one could find also answers to interesting research questions, such as e.g. whether small yet complex projects are more (re)usable [4] or whether packages with proper documentation and/or code style are forked more often [1].

[16]Dataset: https://doi.org/10.5281/zenodo.1165550
[17]Repository: https://github.com/AuthEceSoftEng/msr-2018-npm-miner

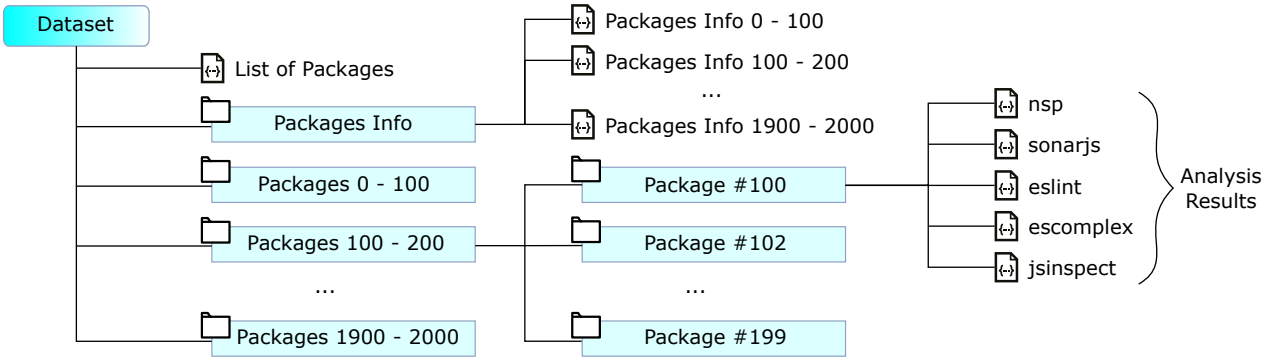

Figure 2: Dataset internal structure.

Another interesting research direction is that of dependency analysis. As our dataset stores the inter-dependencies among packages, one may apply social network analysis techniques and derive metrics (e.g. PageRank), or even employ association analysis to extract common patterns (e.g. find out packages that are more often used together). For this purpose, our index further supports graph-like queries, e.g. "Give me all the packages that are using 'lodash3' or 'underscore' and whose download count is more than 1000 downloads per week". Statistical patterns may also be extracted through crowdsourcing based on the abstract syntax trees provided by our complexity analysis. For example, one can extract common programming idioms by performing the analysis found in [5] for Ruby in JavaScript. Last but not least, `npm-miner` can be used as a recommendation engine for package quality, and thus allow developers to find a high quality package when they are making a decision as to which package to use. In that case, the star count maybe an effect of bias and time and may not reflect the quality characteristics of packages.

6 CONCLUSIONS

In this paper we presented `npm-miner`, a platform for mining the npm registry of packages used in the JavaScript community, as well as a dataset that contains analyses of 5 open source static analysis tools for 2000 popular packages in terms of stars. Throughout the creation of the dataset, we confronted several challenges in order to filter out any poorly-formed candidate packages for analysis and consider only useful packages. The result is a clean and complete dataset that can be used to pursue several interesting research questions, such as those outlined in Section 5.

As future work, we consider integrating more open source tools in order to be able to lint different types of files (e.g. support TypeScript via TSLint or JSON objects via JSONLint). Moreover, we plan to scale our platform to also maintain the analyses of all past versions of packages and not only the latest ones. This would further allow us to investigate the evolution of packages.

ACKNOWLEDGMENTS

Parts of this work have been supported by the European Union's Horizon 2020 research and innovation programme (project Mobile Age - grant agreement No 693319). The authors would like to thank GRNET for providing cloud resources to implement this project.

REFERENCES

[1] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. 2014. Co-evolution of Project Documentation and Popularity Within Github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 360–363.

[2] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. 2012. Standardized Code Quality Benchmarking for Improving Software Maintainability. *Software Quality Journal* 20, 2 (June 2012), 287–307.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, Piscataway, NJ, USA, 470–481.

[4] Valasia Dimaridou, Alexandros-Charalampos Kyprianidis, Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2017. Towards Modeling the User-Perceived Quality of Source Code using Static Analysis Metrics. In *Proceedings of the 12th International Joint Conference on Software Technologies (ICSOFT 2017)*. SciTePress, Setúbal, Portugal, 73–84.

[5] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, Crowd-scale Programming Practice in the IDE. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2491–2500.

[6] ISO25010. 2011. ISO/IEC 25010:2011. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. (2011). [Online; accessed October 2017].

[7] Ciera Jaspan, I Chen, Anoop Sharma, et al. 2007. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, New York, NY, USA, 963–970.

[8] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 102–112.

[9] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2017. *On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem*. Technical Report. arxiv.

[10] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas L. Symeonidis. 2016. User-Perceived Source Code Quality Estimation based on Static Analysis Metrics. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS 2016)*. IEEE Press, Piscataway, NJ, USA, 100–107.

[11] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Vol. 1. IEEE, Piscataway, NJ, USA, 314–324.

[12] Miltiadis G. Siavvas, Kyriakos C. Chatzidimitriou, and Andreas L. Symeonidis. 2017. QATCH - An Adaptive Framework for Software Product Quality Assessment. *Expert Systems with Applications* 86 (2017), 350 – 366.

[13] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 351–361.

[14] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR) (MSR 2017)*. IEEE, Piscataway, NJ, USA, 334–344.