npm-filter: Automating the mining of dynamic information from `npm` packages

Ellen Arteca and Alexi Turcotte*
{arteca.e,turcotte.al}@northeastern.edu

ABSTRACT

The static properties of code repositories, e.g., lines of code, dependents, dependencies, etc. can be readily scraped from code hosting platforms such as GitHub, and from package management systems such as npm for JavaScript; Although no less important, information related to the *dynamic* properties of programs, e.g., number of tests in a test suite that pass or fail, is less readily available. The ability to easily collect this dynamic information could be immensely useful to researchers conducting corpus analyses, as they could differentiate projects based on properties that can only be observed by running them.

In this paper, we present *npm-filter*, an automated tool that can download, install, build, test, and run custom user scripts over the source code of JavaScript projects available on npm, the most popular JavaScript package manager. We outline this tool, describe its implementation, and show that *npm-filter* has already been useful in developing evaluation suites for multiple JavaScript tools.

KEYWORDS

JavaScript, npm, corpus analysis, tool evaluation

ACM Reference Format:
Ellen Arteca and Alexi Turcotte. 2022. *npm-filter*: Automating the mining of dynamic information from `npm` packages. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528501

1 INTRODUCTION

Many code hosting platforms contain a wealth of useful metadata: e.g., GitHub lists code authors, commits, and general project history, and library repositories (such as npm for JavaScript) often contain information on dependencies and dependents. Although it can be readily scraped from the web, this metadata is *static*, and does not tell you much about running the actual code. We thus define *dynamic* metadata to be information gleaned from program executions: e.g., number of running tests, code coverage of tests, performance, memory usage, etc. Making said dynamic metadata available can enable new corpus analyses, focused on data pertaining to program executions—this is the purpose of our tool, *npm-filter*.

npm-filter is a tool for automatically installing, building, and testing sets of npm packages. npm is a major repository for JavaScript library code, and already contains a wealth of static metadata about JavaScript projects (and, if available, a link to the code). *npm-filter* complements this by generating information such as how a project is built, if and how it is tested, the number of passing/failing tests, and the list of transitive dependencies. *npm-filter* runs package code in a sandbox for added security and to ensure reproducibility of results. Users can also specify custom scripts to run over the source code of the package. As far as we know, there is no similar framework to automatically build, run, and test npm packages.

npm-filter has already been used to great effect in three projects; it was used to filter through huge lists of JavaScript projects in crafting evaluations for the *DrAsync* anti-pattern detection tool [27], the *Nessie* test generator for asynchronous JavaScript callbacks [2], and the *Desynchronizer* tool for automatically migrating from synchronous JavaScript APIs to their asynchronous equivalents [11].

2 BACKGROUND & MOTIVATION

Node.js [9] is an eminently popular JavaScript runtime, particularly for server-side JavaScript, and while JavaScript is best known as a front-end, client-side language, it is rapidly gaining in popularity for server-side development [1]. npm [20] is the most popular package ecosystem for Node.js applications: with the npm command-line interface (CLI) installed, a developer needs only navigate to the root of their project and `npm install <package-name>` to download and install any package they desire. JavaScript packages have a `package.json` file in which users can specify commands that can be run by npm, e.g., many developers will specify a `test` command that describes how a package's test suite is run, then a user can execute the package's tests with `npm run test`.

npm provides a wealth of metadata for all of the projects it hosts, including the number of weekly downloads, dependencies, dependents, and a link to the associated code repository. This said, running application code can reveal yet more useful information, such as if the package is equipped with a test suite, passing, failing, flaky tests, etc. But even though it is relatively straightforward to install, build, and test an npm package, in our anecdotal experience conducting JavaScript tool evaluations, we found that only (roughly) <5% of npm packages have running test suites with no failing tests.

npm-filter can be used in any scenario where metadata about the execution of JavaScript code is required. A list of npm projects or JavaScript repositories (e.g., from GHTorrent [12], CodeDJ [18], or scraping npm), can be fed into *npm-filter* to gather dynamic metadata by trying to install, build, and run package tests.

3 *NPM-FILTER* DESIGN

We will describe the overall design of *npm-filter* by way of describing the steps involved in analyzing a given npm package. Analysis

[*] Both authors contributed equally to this research.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528501

comprises various phases of execution, which correspond to the tasks required to set up and test an npm package, and running any user-specified scripts over the package's source code.

3.1 Package Setup and Installation

Supplied with an npm package name, *npm-filter* scrapes the repository link from the npm package page. The source code is then downloaded (with a `git clone`). If the user specified a particular commit to be analyzed, then the source code is checked out at this commit. If there is no repository link found on the page or if there is an issue with the cloning, then *npm-filter* bails out at this stage and reports the error to the user.

Once the code has been downloaded, the package dependencies are installed [1]. The list of transitive dependencies can be a useful piece of data: for example, [30] show that transitive dependencies can contain vulnerabilities that compromise the package itself. *npm-filter* computes this list by reporting the list of all packages in the `node_modules` directory after the install phase has completed. There is also an option to exclude `devDependencies`, which are dependencies excluded from production distributions of the package.

3.2 Building a Package

Once installed, some npm packages have additional commands that need to be run before the package is operational: we call this the *build phase*. For instance, packages written in TypeScript need to be compiled to JavaScript, and another common build step is the application of a bundler such as rollup [23] or webpack [28].

To determine the build commands, *npm-filter* looks at the package's `package.json` file and finds the available commands matching our tracked build commands. By default, these are "`build`", "`compile`", and "`init`" (the most common build commands in our experience). However, users can also customize the build commands tracked with a custom configuration file (discussed in Section 5.3).

If there is an error running a particular build command, the problematic command is added to the end of the command list; this way, the command can be run after potentially prerequisite commands. If all the build commands in a list have errors, then *npm-filter* bails out (to avoid infinite cycling) but continues to the testing phase anyway, reporting the build error in the results.

3.3 Testing a Package

Next, *npm-filter* determines if the package has a test suite, and if so computes some dynamic metadata—this is the *test phase*. `package.json` is further parsed, this time to find the test commands. By default, these are the common ones we observed: "`test`", "`unit' `', "`cov`", "`ci`", "`integration`", "`lint`", "`travis`", "`e2e`", "`bench`", "`mocha`", "`jest`", "`ava`", "`tap`", "`jasmine`"[2].

For each test command, *npm-filter* runs it and determines, by parsing the command itself and its output:

- if it is a linter or a coverage tool, and if so what tool is used;
- if not for linter/coverage, what testing infrastructure is used;
- whether or not it runs new user tests (this is false in test commands that only call other test commands, or that don't run any tests explicitly, e.g., linters, coverage tools);
- if it runs other test commands, then a list of these commands;
- if it does run new user tests, then the number of passing and number of failing tests.

npm-filter parses the output of running tests with the following tools, that were the most common we observed in practice: eslint [7], tslint [21], xx [29], standard [25], prettier [22], gulp lint [14] (linters); istanbul/nyc [16], coveralls [6], c8 [5] (coverage tools); mocha [19], jest [8], jasmine [17], tap [26], lab [15], ava [4], gulp [13] (test tools). Any test commands that run other infrastructures (such as custom Node.js scripts) will still be parsed on a best-effort basis, and whether or not the correct number of passing/failing tests is determined depends on the shape of the output.

3.4 Running Custom Scripts and CodeQL

In addition to the metadata collected about the package build and test suite, users can also specify shell scripts and CodeQL [10] static analysis queries to be run over the source code of the package. The scripts are run in the sequence specified, and any terminal output of each of them is included in the results, including errors.

CodeQL is a semantic code analysis language: with it, users can write static analyses for a variety of languages, including (most relevantly for *npm-filter*) JavaScript/TypeScript. In Section 6.2, we describe how this features was already used in an existing tool.

3.5 Results

The results of all phases of *npm-filter* are output to a JSON file. This JSON results object is organized in a hierarchical structure corresponding to the aforedescribed phases of execution. Any errors in an execution phase are reported in the corresponding field of the results. The output file is named `[package name]__results.json`.

If the user specifies CodeQL queries to be run over the package source code, the output of each of these queries is output to a CSV file, named `[package name]__[query name]__results.csv`. Any errors in the CodeQL query execution would be reported in the CodeQL field of the JSON results.

4 IMPLEMENTATION

npm-filter is written in Python. All the npm commands we run are done by dispatching with the Python `subprocess` library; this allows us to parse the output, and specify a timeout. It also doesn't crash *npm-filter* if there is any error in the subprocess.

The back end of *npm-filter*'s npm package analyzer is a web scraper: given the name of an npm package, it finds the associated repository link on the npm page so that it can analyze the package's source code. The scraper is built using Python's scrapy library [24], which allows us to include custom middleware to run if the scraper gets an error code as a response from the site. We implemented some middleware to deal with errors caused by the rate limiting on the npm site: if the site returns an error indicating that too many requests were received, the scraper pauses and then retries. This middleware ensures that the scraper will not miss package information because of the rate limiter, but if a user is analyzing a large number of packages they will see a significant performance hit compared to running on the GitHub repos directly. Thus, we also provide an option for users to pass a list of GitHub repos instead of npm packages to be analyzed, skipping the scraping entirely.

[1] *npm-filter* supports both `npm` and `yarn` package managers for installing dependencies.
[2] Many of these correspond to JavaScript testing infrastructures, such as `mocha`.

npm-filter is open source and includes a detailed Readme, with more examples than are included in this paper. *npm-filter* is available at https://github.com/emarteca/npm-filter/ [3].

5 *NPM-FILTER* USAGE

In this section, we explain how to use *npm-filter* and give some examples of usage. To follow along, clone the source code linked above; all example commands are run from the root of the repo. We have also included a minimal example usage tutorial here[3].

5.1 Safety first: Running in Docker

npm-filter can be run in a docker container that is provided on DockerHub[4], and we recommend this usage. The repository's Readme includes a list of all system requirements if you choose to run it locally or if you want to rebuild the docker container. To run *npm-filter* sandboxed, simply preface any commands with `./runDocker.sh`.

5.1.1 Input/Output from docker to host machine. Running *npm-filter* in docker allows all the code being analyzed to be run in a sandbox, protecting the host machine. To allow input to *npm-filter* and access to the results files from running in docker, we have some special directories that the docker container has access to. All input files to running *npm-filter* in docker must be in a directory `docker_configs` in the *npm-filter* home directory (any user scripts, CodeQL queries, or custom configuration files). Results files end up in the `npm_filter_docker_results` directory, which is also in the *npm-filter* home directory.

5.2 Basic usage

This tool can either take JavaScript packages specified as GitHub repository links, or as npm packages.

To run *npm-filter* over GitHub repo links, use the following:

```
./runDocker.sh python3 src/diagnose_github_repo.py
                [--repo_list_file [rlistfile]]
                [--repo_link [rlink]]
                [--repo_link_and_SHA [rlink_and_SHA]]
                [--config [config_file]]
                [--output_dir [output_dir]]
```

All arguments are optional, although *npm-filter* will not do anything if no repo links are specified.

- `repo_list_file`: a file containing a list of GitHub repo links to be analyzed. Each line of the input file must specify one repo link, with an optional whitespace delimited commit SHA to check the repo out at.
- `repo_link`: a link to a single GitHub repo to be analyzed
- `repo_link_and_SHA`: link to a GitHub repo followed by a space-delimited commit SHA to analyze the repo at
- `config`: path to a configuration file for the tool (config options explained in Section 5.3)
- `output_dir`: path to a directory in which to output the results files (note: this only works when not running in docker)

To run *npm-filter* over npm packages, use the following:

```
./runDocker.sh python3 src/diagnose_npm_package.py
                        --packages [list_of_packages]
                        [--config [config_file]]
                        [--html [html_file]]
                        [--output_dir [output_dir]]
```

- `packages`: list of npm packages to analyze. Required argument, and at least one package must be passed.
- `config`: path to a configuration file for the tool
- `html`: path to an html file that represents the npm page for the package that is specified to be analyzed. This option only works for one package, so if you want to use this option on multiple packages you'll need to call the tool in sequence.
- `output_dir`: path to a directory in which to output the results files (note: this only works when not running in docker)

5.2.1 Example Usage. What follows is an example of basic usage. This example runs on a single package, specified by GitHub repo and at a specific commit (to ensure consistency of expected output).

```
./runDocker.sh python3 src/diagnose_github_repo.py
  --repo_link_and_SHA https://github.com/streamich/memfs
        863f373185837141504c05ed19f7a253232e0905
```

The results file is `npm_filter_docker_results/memfs__results.json`, with contents (slightly redacted for length):

```
    "installation": {
        "installer_command": "yarn"
    },
    "build": {
        "build_script_list": [
            "build"
        ]
    },
    "testing": {
        "test": {
            "num_passing": 265,
            "num_failing": 0,
            "test_infras": [
                "jest"
            ],
------------------------------ REDACTED FOR LENGTH
    "metadata": {
      "repo_link": "https://github.com/streamich/memfs",
      "repo_commit_SHA": REDACTED FOR LENGTH
    }
```

From this we can see that at this commit `memfs` has a test suite with 265 passing tests and no failing tests, among other metadata.

More examples are included in the *npm-filter* GitHub repo Readme.

5.2.2 Batch dispatch. A common application of *npm-filter* is to analyze a large number of packages/repos. We provide a bash script that dispatches *npm-filter* in parallel across batches of inputs.

```
./runParallelGitReposDocker.sh repo_link_file
```

Results are in `npm_filter_parallel_docker_results`. Note that this parallel execution in performed in one docker container, and not multiple parallel docker containers.

5.3 Custom *npm-filter* configuration

Users can customize the behaviour of the tool by providing a custom configuration JSON file, organized by phases of *npm-filter* analysis. All fields are optional – if not provided, defaults will be used[5].

[3] https://github.com/emarteca/npm-filter/blob/master/Tutorial.md
[4] https://hub.docker.com/r/emarteca/npm-filter

[5] Default configuration: https://github.com/emarteca/npm-filter/tree/master/configs.

Install. package installation.

- `timeout`: number of millisections after which, if the install is not complete, the process bails with a timed out error

Dependencies. package dependency tracking (this is the libraries the current package depends on, both directly and transitively).

- `track_deps`: specifies to compute the package dependencies
- `include_dev_deps`: if true, this specifies to include the devDependencies in the dependency computation
- `timeout`: timeout in milliseconds

Build. package compile/build stage.

- `tracked_build_commands`: any npm script with one of these listed commands as a substring will be tested.
- `timeout`: timeout in milliseconds, per build command

Test. package test stage.

- `track_tests`: specifies to run this testing diagnostic stage
- `tracked_test_commands`: any npm script with one of these listed commands as a substring will be tested.
- `timeout`: timeout in milliseconds, per test command

Meta-info. any analysis-level configurations.

- `VERBOSE_MODE`: if true, include full output of all commands
- `ignored_commands`: commands to ignore: if these are present in the npm script name, then they are not run even if they otherwise fall into a category of commands to run.
- `ignored_substrings`: commands to ignore: if these strings are present in the command string itself, then these npm scripts are not run (same as `ignored_commands`, but for the command strings instead of the npm script names)
- `rm_after_cloning`: delete the package source code after the tool is done analyzing it. Strongly recommended if running over a large batch of packages.
- `scripts_over_code`: list of paths to script files to run over the package source code.
- `QL_queries`: list of paths to QL query files to run over the package source code.

6 *NPM-FILTER* IN PRACTICE

Now we describe three research papers that have used *npm-filter*.

6.1 DrAsync

Turcotte et al. used *npm-filter* to collect projects to evaluate their tool to detect anti-patterns in asynchronous JavaScript programs [27]. Their tool, called *DrAsync*, can statically detect asynchronous anti-patterns, and they found that many of these anti-patterns could be manually refactored; in order to confirm that these refactorings preserved behaviour, the authors ran application tests before and after refactoring (to confirm that refactoring did not introduce any failing tests). The tool also has a dynamic component that records promise lifetimes and displays them in a visualization.

Thus, the evaluation undertaken in the paper requires running test suites, and *npm-filter* was used to filter a list of 40K JavaScript Github repositories with asynchronous JavaScript code to a much more manageable 450 projects that had running/passing tests. This work is being presented concurrently at ICSE Technical Track.

6.2 Nessie

Arteca et. al built a test generator for JavaScript APIs with callback arguments [2]. In this project, they wrote a static analysis in CodeQL, to identify pairs of nested calls to functions that were part of the APIs the test generator was targeting. Then they used the CodeQL plugin feature of *npm-filter* to run this analysis on 13.6K JavaScript projects on GitHub. The results of this CodeQL query, amalgamated across all 13.6K projects, was used to inform the test generator of common pairs of nested API calls, to generate tests more representative of developers' use of the APIs. They also used *npm-filter* to select projects to evaluate the test generator. This work is being presented concurrently at ICSE Technical Track.

6.3 Desynchronizer

Gokhale et al. used *npm-filter* to collect projects to evaluate their tool for automatically migrating projects that use synchronous JavaScript APIs to use their asynchronous equivalents [11]. The tool, called *Desynchronizer*, statically detects calls to synchronous JavaScript APIs that have asynchronous equivalents (e.g., calls to `readFileSync`, rather than `readFile`)—then infers a call graph, and refactors the code. In the evaluation, authors applied *every* refactoring, and ran test suites post refactoring to establish any behavioural differences. Thus, runnable test suites with no failing tests were required in the evaluation, and *npm-filter* was used to filter a list of 50K JavaScript projects using APIs targeted by the tool down to a few hundred projects with passing test suites.

7 *NPM-FILTER* LIMITATIONS

We currently only support packages hosted on GitHub: if there is no GitHub repo link available on the package page, then *npm-filter* will not work. In our use cases we have found this to be rare.

If the package uses a testing tool that we have not implemented output parsing for, then it might not be properly tracked. That said, we have covered the most popular JavaScript test ecosystems. Also, if the package uses build/test commands that don't include the substrings we expect, then they won't be run. Note, however, that users can customize their *npm-filter* configuration to add or remove as many tracked commands as they want.

8 CONCLUSION

npm and GitHub contain a wealth of metadata related to static JavaScript project properties, but augmenting this static information with dynamic properties such as the number of tests in a test suite that pass or fail is immensely useful to researchers conducting corpus analyses or testing program transformation tools. In this paper, we presented *npm-filter*, an automated tool that can download, install, build, test, and run custom user scripts over the source code of JavaScript projects available on npm, the most popular JavaScript package manager. In addition to describing the implementation and usage of *npm-filter*, we also show that it has already been useful in developing evaluation suites for three separate JavaScript tools.

ACKNOWLEDGMENTS

Both authors were supported in part by National Science Foundation grants CCF-1715153 and CCF-1907727, and by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–36.

[2] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. 2022. Nessie: Automatically Testing JavaScript APIs with Asynchronous Callbacks. In *ICSE '22*.

[3] Ellen Arteca and Alexi Turcotte. 2022. *emarteca/npm-filter: 1.0.0*. https://doi.org/10.5281/zenodo.6374358

[4] ava. 2022. ava. https://www.npmjs.com/package/ava. Accessed: 2022-01-20.

[5] c8. 2022. c8. https://www.npmjs.com/package/c8. Accessed: 2022-01-20.

[6] coveralls. 2022. coveralls. https://www.npmjs.com/package/coveralls. Accessed: 2022-01-20.

[7] eslint. 2022. eslint. https://www.npmjs.com/package/eslint. Accessed: 2022-01-20.

[8] Facebook. 2022. jest. https://jestjs.io/. Accessed: 2022-01-20.

[9] OpenJS Foundation. [n.d.]. Node.js. https://nodejs.org/en/. Accessed 2020-08-27.

[10] GitHub. 2022. CodeQL. https://github.com/github/codeql. Accessed: 2022-01-20.

[11] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021-10-20. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proceedings of the ACM on programming languages.* 5, OOPSLA (2021-10-20).

[12] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[13] gulp. 2022. gulp. https://www.npmjs.com/package/gulp. Accessed: 2022-01-20.

[14] gulp eslint. 2022. gulp-eslint. https://www.npmjs.com/package/gulp-eslint. Accessed: 2022-01-20.

[15] hapi. 2022. lab. https://www.npmjs.com/package/@hapi/lab. Accessed: 2022-01-20.

[16] Istanbul. 2022. nyc. https://www.npmjs.com/package/nyc. Accessed: 2022-01-20.

[17] jasmine. 2022. jasmine. https://www.npmjs.com/package/jasmine. Accessed: 2022-01-20.

[18] Petr Maj, Konrad Siek, Alexander Kovalenko, and Jan Vitek. 2021. CodeDJ: Reproducible Queries over Large-Scale Software Repositories. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:24. https://doi.org/10.4230/LIPIcs.ECOOP.2021.6

[19] mocha. 2022. mocha. https://www.npmjs.com/package/mocha. Accessed: 2022-01-20.

[20] npm. [n.d.]. npm. https://www.npmjs.com/. Accessed 2020-08-27.

[21] palantir. 2022. tslint. https://www.npmjs.com/package/tslint. Accessed: 2022-01-20.

[22] prettier. 2022. prettier. https://www.npmjs.com/package/prettier. Accessed: 2022-01-20.

[23] Rollup. 2022. Rollup. https://www.npmjs.com/package/rollup. Accessed: 2022-01-20.

[24] scrapy. 2022. scrapy. https://scrapy.org/. Accessed: 2022-03-21.

[25] standard. 2022. standard. https://www.npmjs.com/package/standard. Accessed: 2022-01-20.

[26] tap. 2022. tap. https://www.npmjs.com/package/tap. Accessed: 2022-01-20.

[27] Alexi Turcotte, Michael D. Shah, Mark W. Aldrich, and Frank Tip. 2022. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In *ICSE '22*.

[28] webpack. 2022. webpack. https://www.npmjs.com/package/webpack. Accessed: 2022-01-20.

[29] xx. 2022. xx. https://www.npmjs.com/package/xx. Accessed: 2022-01-20.

[30] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.