`npm-follower`: A Complete Dataset Tracking the NPM Ecosystem

Donald Pinckney
pinckney.d@northeastern.edu
Northeastern University
USA

Federico Cassano
cassano.f@northeastern.edu
Northeastern University
USA

Arjun Guha
a.guha@northeastern.edu
Northeastern University
USA

Jonathan Bell
j.bell@northeastern.edu
Northeastern University
USA

ABSTRACT

Software developers typically rely upon a large network of dependencies to build their applications. For instance, the NPM package repository contains over 3 million packages and serves tens of billions of downloads weekly. Understanding the structure and nature of packages, dependencies, and published code requires datasets that provide researchers with easy access to metadata and code of packages. However, prior work on NPM dataset construction typically has two limitations: 1) only metadata is scraped, and 2) packages or versions that are deleted from NPM can not be scraped. Over 330,000 versions of packages were deleted from NPM between July 2022 and May 2023. This data is critical for researchers as it often pertains to important questions of security and malware. We present `npm-follower`, a dataset and crawling architecture which archives metadata and code of all packages and versions as they are published, and is thus able to retain data which is later deleted. The dataset currently includes over 35 million versions of packages, and grows at a rate of about 1 million versions per month. The dataset is designed to be easily used by researchers answering questions involving either metadata or program analysis. Both the code and dataset are available at https://dependencies.science.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**.

KEYWORDS

NPM, dependency-management, JavaScript, data mining, archiving

ACM Reference Format:
Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. 2023. `npm-follower`: A Complete Dataset Tracking the NPM Ecosystem. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613094

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613094

1 INTRODUCTION

Modern software development relies extensively on a complex network of reusable open-source software components (packages). The largest [6] repository of packages is the NPM repository, which contains over three million packages, and 35 million different versions of packages while serving tens of billions of downloads weekly. Practically every JavaScript application depends on packages from the NPM repository. Understanding the NPM ecosystem, including distribution properties, versioning, and dependency relations is an important component for reasoning about JavaScript software development practices [14, 36], security [3, 12, 21], program analysis [8, 18, 20, 30] and more.

Existing package repository datasets, such as `libraries.io` [35] and DaSEA [4], provide a wealth of information about dependency structure, author information, etc., even across multiple package ecosystems. However, these datasets are typically limited in two ways: (1) only storing metadata and not code of packages; and (2) not maintaining historic data when packages or versions of packages are unpublished or deleted from NPM. Unfortunately, packages are often deleted from NPM. Between July 12, 2022, and May 10, 2023, we have detected 335,325 versions of packages that have been deleted. This loss of data is problematic for data availability and artifact reproducibility for research which may use package data (regression testing [19], static analysis [8], training large-language models [15], etc.), and makes research areas that specifically examine deleted packages (such as malware analysis) nearly impossible without privileged access [32].

We present `npm-follower` as a platform to enable easier research on the NPM ecosystem. We believe that `npm-follower` offers two main benefits. First, `npm-follower` continually collects and archives packages, and thus retains data (including package code[1]) which is later deleted from NPM. Second, `npm-follower` scrapes and indexes multiple sources of data (developer-provided metadata, code, download metrics and security advisories), allowing researchers to easily write analyses which touch many aspects of the NPM ecosystem. The `npm-follower` dataset and source code is available at https://dependencies.science, and we hope that it will be useful to the research community.

[1] Note that `npm-follower` collects the code which is released by developers to NPM, which may be different from the source code of a project's GitHub repository (Section 2).

2 RELATED WORK

A variety of existing tools scrape data from software ecosystems. Roughly, these can be divided into two areas: those that include metadata only, and those that store source code.

The `libraries.io` [35] website hosts a dataset of metadata spanning multiple package managers, and is quite detailed. However, Buchkova et al. [4] report that it is not well maintained and does not include metadata for all versions of packages, and in response introduced the DaSEA dataset which is a cross-ecosystem dataset containing metadata for versions of packages. Unfortunately, DaSEA does not include NPM (the largest and fastest growing repository [6]). In addition, neither `libraries.io` nor DaSEA store package code themselves. Thus, to perform package code analysis one would have to download package code from NPM, which is time and labor intensive, and is impossible for packages that have been deleted from NPM.

On the other hand, large-scale projects exist which archive not only metadata but also source code. GHTorrent [11] and World of Code [17] collect and archive source code from VCS hosting platforms (GitHub, etc.). Unfortunately, GHTorrent appears to be unmaintained, and both focus on scraping VCS data rather than package manager repositories. Packages uploaded to NPM do not necessarily have an associated (public) VCS repository, and code uploaded to NPM may in fact be different from source code in a VCS repository, so these are related but complementary sources of data. The Software Heritage archive [13] collects the full source code of packages across multiple software ecosystems, including NPM. However, the Software Heritage archive performs intermittent scraping, similar to the Wayback Machine [1], so it is not able to download packages which are uploaded and deleted in-between scrapes. In contrast, `npm-follower` receives updates from the NPM repository and downloads new packages as they appear with low latency (Section 4.2).

Using software ecosystem datasets, researchers are able to examine many interesting research topics, such as technical lag [10], versioning [29, 37], micro packages [16], static analysis [8], malware analysis [25, 32, 38, 39], security vulnerability analysis [5, 7] and more, all of which need access to either metadata or package source code data. In our prior work, we used NPM ecosystem data to evaluate our technique for optimal dependency solving [27], and to understand how developers make use of semantic versioning and updates in practice [26]. The first version of `npm-follower` was born out of that work, and since then we have provided more built-in analyses, added scraping of package download metrics, and continued to improve reliability. In this paper we discuss key design decisions of `npm-follower`, as well as challenges for sustainability of the system.

3 USING NPM-FOLLOWER

The `npm-follower` dataset is useful for answering research questions involving metadata and/or source code analysis, such as evaluation of static analysis tooling, detection of malware, or training code large-language models. To illustrate how `npm-follower` could be used in such research, we present a hypothetical example of vulnerability impact analysis [28, 33], which has the goal of identifying client libraries that may be impacted by a security vulnerability in a dependency, and if the vulnerable code is in fact reachable from the client. We can use `npm-follower` to perform the first half of that task: finding pairs of clients and dependencies, where the dependency has a vulnerability. To do so, we will first work on building the set of dependencies, and then match them with dependent clients. A variant of this example is available as a video demonstration. [2]

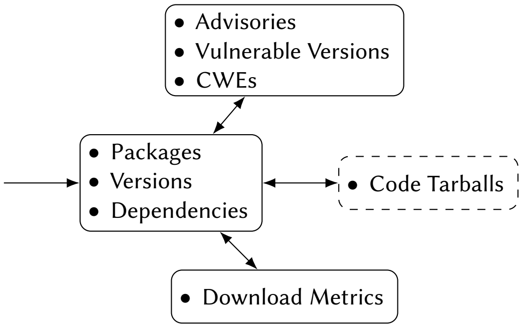

Figure 1: Conceptual structure of the `npm-follower` database.

3.1 Finding Packages with Vulnerabilities

In addition to package metadata, `npm-follower` also scrapes security advisories from the GitHub Advisory Database. We can find packages that have vulnerabilities by joining the table of packages (center of Figure 1) with the table of vulnerabilities (top of Figure 1):

```
select ...
from packages vuln_p
join vulnerabilities vuln on vuln_p.name = vuln.package_name
```

However, in order to obtain a smaller, more focused dataset we may wish to only select packages which also have a decent number of downloads. We may accomplish this by additionally joining scraped download metrics (bottom of Figure 1) and keeping only those with over 1 million weekly downloads:

```
...
join download_metrics m on m.package_id = vuln_p.id
and (m.download_counts[array_upper(m.download_counts, 1)]).counter
  > 1000000
```

Note that the code outlined here does not distinguish different versions of a vulnerable package, even though typically vulnerabilities only affect some versions. A more complex analysis that selects specific versions of packages that are vulnerable is possible with `npm-follower`, and in fact already has a reusable implementation (Section 4.1.2).

3.2 Determining Dependent Clients

Now that we have a set of packages that contain security vulnerabilities, we can find a corresponding set of dependent client package versions by using a relation describing the dependencies of each version of each package:

```
...
join metadata_analysis.version_direct_runtime_deps edge
  on edge.depends_on_pkg = vuln_p.id
join versions client on client.id = edge.v
```

[2] https://youtu.be/OgLYThRJhdc?si=V6krLg3LzUvUeH7u

This dependency relation table is not part of the core data model of npm-follower but is computed from the core tables with a provided analysis implementation. As above, this simple query does not take into account the version of the vulnerable package, so it may be that the clients depend on non-vulnerable versions. If needed, this may be addressed by writing a more complex query on the dependency version constraint data structure (Section 4.1.2).

Finally, if we aim to use dynamic analysis techniques to look at vulnerability impact in the clients, we may wish to only select clients which have tests. Since npm-follower stores in original JSON format all metadata it does not specifically extract, we may use this to filter for tests:

```
...
and client.extra_metadata->'scripts'->'test' is not null
```

We have now completed finding our set of client and vulnerable library pairs we wish to analyze, and can move on to retrieving package code for these pairs.

3.3 Obtaining Package Code

We now would like to obtain the code, say for the clients, to proceed with our vulnerability impact analysis. One way would be to read the `client.tarball_url` column from the query above and download each tarball. Unfortunately, some of these URLs might return 404 errors because developers could have unpublished those versions due to depending on a vulnerability, leading to obtaining a biased sample.

A different approach would be to use the npm-follower code store (Section 4.2), which attempts to archive tarballs before they are deleted. Doing so is easy, as all source code URLs map into the object store, which can then be read from using npm-follower's tooling. After obtaining datasets of lists of vulnerable packages and clients and associated source code, we are now in a good position to explore exciting research techniques to determine vulnerability impact.

4 DESIGN AND IMPLEMENTATION

When designing npm-follower, we had two primary design goals: (1) it should be a comprehensive and easy to use dataset for analysis of the NPM ecosystem; and (2) it should be able to be sustainably run using our available hardware resources. Specifically, we have available an academic Slurm-backed [31] HPC cluster with around 25 TB of networked file storage, which we use for downloading and storing tarballs of package code. The metadata portion of npm-follower is able to be run independently, on a single Linux VM with 4 CPUs and 128 GB of RAM, currently requiring about 700 GB of SSD storage.

4.1 Package Metadata

While the most obvious contribution of npm-follower is in the scraping and storing of package code data, we nevertheless designed the metadata scraping and analysis component of npm-follower to behave well in the presence of package deletions, and to use a richer data model than prior work to enable more complex analyses. The metadata of npm-follower is stored in PostgreSQL [34], which provides for a consistent and easy to use data analysis platform while providing sufficient throughput to index updates from NPM.

4.1.1 Streaming and Parsing Updates. NPM offers a changes streaming API [24] which allows us to stream updates (package / version creation and deletion operations) in a JSON format without needing to frequently crawl the NPM website. Unfortunately, the raw JSON updates have two major problems: (1) the data is poorly structured with little validation, making basic data analysis difficult; and (2) when a new version of a package is published, the corresponding change notification contains all previous versions in addition to the new version, which causes storage to grow quadratically with the number of updates if naively storing all change notifications.

To better parse and index the metadata, npm-follower first validates and cleans update events as it receives them, and inserts the data into the relational database while de-duplicating repetitive data. In addition, we carefully parse common metadata fields which are useful for data analysis into interpretable data structures, including version numbers, dependency version constraints, and GitHub repository data. Metadata fields which we don't specifically parse (e.g. author's names and emails) or fail to parse (e.g. invalid source code repository URIs) are retained in JSON format and available for querying. Additionally, data fields are not deleted when a package (or version of a package) is deleted from NPM. Instead, the entity is only marked with a deleted flag.

4.1.2 Querying Package Metadata. Three tables store the metadata obtained from the NPM changes API: `packages`, `versions` and `dependencies`, which collectively enumerate all versions of all packages, and the dependencies of each version. These tables make up the core data model of npm-follower (center of Figure 1), and are typically the starting point of analyses.

Since npm-follower performs parsing on many metadata fields, it is often possible to write SQL queries which directly interpret these fields. The most interesting example is dependency version constraints, which are parsed from their string format into disjunctive normal form (DNF) over the total ordering of version numbers. For example, the version constraint "`12 || 13.0.1`" would be parsed into $(X \geq 12.0.0 \land X < 13.0.0) \lor (X \geq 13.0.1 \land X < 13.1.0)$, and finding a matching version then corresponds to finding a satisfying assignment for X drawn from the set of versions of the dependency. Matching versions can be computed in SQL by matching candidate versions to each ordering term, and then aggregating conjunctions followed by aggregating disjunctions.

In contrast, prior work [4, 35] only provides version constraints as uninterpreted strings, and leaves it up to the user of the dataset to interpret the constraints if desired. Interpreting constraints correctly either requires substantial work [27], or forces the analysis pipeline to interoperate with a JavaScript package for interpreting version constraints [22].

Since writing queries that operate on these interpreted data structures is non-trivial, npm-follower includes a small library of common analyses that users may wish to use and build on. Some of these analyses include computing updates between versions of packages[3], resolutions of direct dependency version constraints, transitive dependency graph computation, and identifying versions of packages which contain a security vulnerability.

[3] tricky because version ordering and temporal ordering need not agree, see our prior work for details on this analysis [26].

4.2 Code Acquisition and Storage

When npm-follower receives metadata updates that contain URLs to new package code tarballs, npm-follower enqueues a download job to then be handled by the code data downloading and storage subsystem running on our HPC cluster.

Storing package code data is challenging, due to both the scale (tens of millions of tarballs, 20+ TB) and the need to handle sufficient concurrent writes. We did not explore using existing distributed file systems such as Hadoop [9] due to concern of Hadoop's scalability with regards to storing many small files [2] (our use case). In addition, we are unsure if Hadoop can run correctly and efficiently on top of the networked file system at our disposal.

Instead, we store tarball data in a custom-built object storage system stored on the networked file system. A manager node controls access to the object storage, keeping track of byte offsets and coordinating locks for writing, while individual worker nodes in the HPC cluster perform the networked disk I/O. Download jobs are dequeued from the work queue (enqueued by the metadata subsystem) and assigned to worker nodes in the HPC cluster.

We have observed that the download latency (tarball published to NPM → downloaded by npm-follower) has a bimodal distribution, with one group of tarballs having quite low latency (about 30 s), and another group of tarballs having higher latency (1 hour – 1 day). The higher latency downloads appear correlated with periods of higher load, and may be caused both by unavoidable latency in NPM sending change notifications, and latency within npm-follower itself. Overall, we are able to download 98.8% of tarballs within a latency of 24 hours, which is satisfactory for our purposes.

To allow data analysis jobs to read from the package code data (right side of Figure 1), the manager node exposes a mapping of tarball keys (derived from the downloaded_tarballs table) to underlying file system location information (file name, byte offset, number of bytes). To read a package code tarball, a worker node first queries the manager node for the location on disk, and then itself performs the read from the underlying (networked) file system. Fortunately, our software abstracts over this separation, allowing for a simple cp-like command to read a file out of the object store given a key. This system allows for large-scale concurrent reading from the object storage to perform code analysis.

4.2.1 Tarball Size Distribution. Currently the object storage system is about 24 TB in size and stores over 35 million tarballs. However, the distribution of tarball sizes is extremely skewed (median = 18.4 KB, mean = 730 KB), with the largest tarball being over 500 MB. Based on this skewed distribution, one could consider trading-off completeness for storage size. For instance, discarding all tarballs greater than 16 MB would cut the total size in half, while retaining 99.12% of all tarballs. While most of these oversize tarballs belong to obscure packages, sprinkled among them are popular packages, such as the NPM CLI (52 MB, 890+ million downloads) and gherkin (120 MB, 187+ million downloads). In the future we plan to investigate better discarding strategies by incorporating both tarball size and download metrics, in order to keep the storage requirements for npm-follower sustainable.

4.3 Scraping External Metadata

Additionally, npm-follower scrapes two other sources of metadata: security advisory metadata from the GitHub Security Advisory Database (top of Figure 1) and package download metrics (bottom of Figure 1).

The security advisory metadata lists security advisories for NPM packages, which versions of packages are vulnerable, and applicable CWEs. Our prior work used this security metadata in a prototype of npm-follower to analyze the relationships between semantic versioning and security effects [26]. The package download metrics provide weekly time-series data on the number of downloads each package receives, and are often useful for pre-filtering data prior to other analyses, such as to focus on the top N downloaded packages.

Unlike the metadata of packages, for both security metadata and download metrics we do not have a convenient changes API which can notify us of new data, and in particular the scraping of package download metrics [23] is challenging due to severe rate-limiting. To scrape download metrics for all packages in a reasonable amount of time, we must perform batch requests to the API, which unfortunately precludes scraping per-version download metrics. With this strategy, scraping download metrics for all packages takes about four days. We estimate that if we did not batch requests it would take about two weeks. If rate limits were to increase, we could consider scraping per-version metrics in the future.

4.4 Adaptability to Other Ecosystems

In addition to NPM, many other package repositories are important for software engineering, such as PyPI, APT and more. The general architecture of npm-follower may be applied to create comprehensive datasets of other ecosystems, depending on available APIs. The main requirement is a change notification API, which is crucial for enabling continual archiving of packages, and is central to the design of npm-follower. Designing a unified data model for multiple ecosystems could be challenging while maintaining easy querying and interpretable data structures, though prior work [4, 35] has partially tackled this. To have a unified interpretable format for version constraints, the DNF format of npm-follower may be able to used as a low-level target to which high-level constraints of various diverse syntaxes are parsed into. Investigating generalizing npm-follower to allow for other ecosystems could be an interesting direction for future work.

5 CONCLUSION

NPM is a quickly evolving and often unreliable archive of data, as packages are deleted frequently. In this demonstration, we have presented npm-follower, a scraper and dataset which continually downloads and archives metadata and code from NPM packages. We have further shown the utility of npm-follower for researchers working in the area of program analysis or software ecosystem analysis. The code and dataset, featuring a complete account of the metadata and code of all versions of all packages is available publicly at https://dependencies.science.

ACKNOWLEDGMENTS

This work was funded in part by NSF CCF-2102288, CCF-2100037 and NSF CNS-2100015.

REFERENCES

[1] Internet Archive. 2023. Wayback Machine. https://web.archive.org. Accessed May 5 2023.

[2] Szele Balint. 2009. The Small Files Problem. https://blog.cloudera.com/the-small-files-problem/. Accessed Mar 13 2023.

[3] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. 2011. Vetting Browser Extensions for Security Vulnerabilities with VEX. *Commun. ACM* 54, 9 (sep 2011), 91–99. https://doi.org/10.1145/1995376.1995398

[4] Petya Buchkova, Joakim Hey Hinnerskov, Kasper Olsen, and Rolf-Helge Pfeiffer. 2022. DaSEA: A Dataset for Software Ecosystem Analysis. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 388–392. https://doi.org/10.1145/3524842.3528004

[5] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (30 Mar 2021), 47. https://doi.org/10.1007/s10664-021-09951-x

[6] Erik DeBill. 2023. Modulecounts. http://www.modulecounts.com. Accessed May 5 2023.

[7] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401

[8] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE)*. 752–761. https://doi.org/10.1109/ICSE.2013.6606621

[9] The Apache Software Foundation. 2023. Apache Hadoop. https://hadoop.apache.org. Accessed Mar 13 2023.

[10] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*, Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli (Eds.). Springer International Publishing, Cham, 182–192. https://doi.org/10.1007/978-3-319-57735-7_17

[11] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 12–21. https://doi.org/10.1109/MSR.2012.6224294

[12] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the World Wide Web from Vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/2001420.2001442

[13] Software Heritage. 2023. Software Heritage. https://www.softwareheritage.org. Accessed May 5 2023.

[14] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. 2019. Tool Choice Matters: JavaScript Quality Assurance Tools and Usage Outcomes in GitHub Projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 476–487. https://doi.org/10.1109/ICSE.2019.00060

[15] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. https://doi.org/10.48550/arXiv.2211.15533 arXiv:2211.15533 [cs.CL]

[16] 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. https://doi.org/10.48550/ARXIV.1709.04638

[17] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data. *Empirical Softw. Engg.* 26, 2 (mar 2021), 42 pages. https://doi.org/10.1007/s10664-020-09905-9

[18] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 499–509. https://doi.org/10.1145/2491411.2491417

[19] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. 109 (2018), 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

[20] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 187 (nov 2020), 25 pages. https://doi.org/10.1145/3428255

[21] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.Js Applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3460319.3464836

[22] NPM. 2022. semver(1) – The semantic versioner for npm. https://github.com/npm/node-semver.

[23] NPM and Contributors. 2022. package download counts. https://github.com/npm/registry/blob/1c794110badd54b9fb08e7489746b6089c6648/docs/download-counts.md. Accessed Aug 19 2023.

[24] NPM and Contributors. 2023. registry-follower-tutorial. https://github.com/npm/registry-follower-tutorial. Accessed Mar 12 2023.

[25] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22)*. Association for Computing Machinery, New York, NY, USA, Article 127, 10 pages. https://doi.org/10.1145/3538969.3544415

[26] D. Pinckney, F. Cassano, A. Guha, and J. Bell. 2023. A Large Scale Analysis of Semantic Versioning in NPM. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 485–497. https://doi.org/10.1109/MSR59073.2023.00073

[27] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. 2023. Flexible and Optimal Dependency Management via Max-SMT. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1418–1429. https://doi.org/10.1109/ICSE48619.2023.00124

[28] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (01 Sep 2020), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

[29] S. Raemaekers, A. van Deursen, and J. Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Los Alamitos, CA, USA, 215–224. https://doi.org/10.1109/SCAM.2014.30

[30] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1806596.1806598

[31] SchedMD and Contributors. 2023. Slurm Workload Manager – Documentation. https://slurm.schedmd.com. Accessed Mar 12 2023.

[32] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious Npm Packages. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, 1681–1692. https://doi.org/10.1145/3510003.3510104

[33] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-Based Web Servers. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 361–376.

[34] The PostgreSQL Global Development Group. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org. Accessed Mar 12 2023.

[35] Inc Tidelift. 2023. Libraries.io – The Open Source Discovery Service. https://libraries.io. Accessed May 5 2023.

[36] Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie Van Deursen. 2020. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* 46, 8 (2020), 863–891. https://doi.org/10.1109/TSE.2018.2871058

[37] 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* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 351–361. https://doi.org/10.1145/2901739.2901743

[38] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What Are Weak Links in the Npm Supply Chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/3510457.3513044

[39] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 995–1010.

Received 2023-05-11; accepted 2023-07-20