A Large Scale Analysis of Semantic Versioning in NPM

Donald Pinckney*, Federico Cassano*, Arjun Guha*, Jon Bell*, Massimiliano Culpo†, Todd Gamblin‡
*Northeastern University, Boston, MA, USA
{pinckney.d, cassano.f, a.guha, j.bell}@northeastern.edu
†np-complete, S.r.l., Italy
massimiliano.culpo@googlemail.com
‡Lawrence Livermore National Laboratory, Livermore, CA, USA
tgamblin@llnl.gov

***Abstract*— The NPM package repository contains over two million packages and serves tens of billions of downloads per-week. Nearly every single JavaScript application uses the NPM package manager to install packages from the NPM repository. NPM relies on a "semantic versioning" ('semver') scheme to maintain a healthy ecosystem, where bug-fixes are reliably delivered to downstream packages as quickly as possible, while breaking changes require manual intervention by downstream package maintainers. In order to understand how developers use semver, we build a dataset containing every version of every package on NPM and analyze the flow of updates throughout the ecosystem. We build a time-travelling dependency resolver for NPM, which allows us to determine precisely which versions of each dependency would have been resolved at different times. We segment our analysis to allow for a direct analysis of security-relevant updates (those that introduce or patch vulnerabilities) in comparison to the rest of the ecosystem. We find that when developers use semver correctly, critical updates such as security patches can flow quite rapidly to downstream dependencies in the majority of cases (90.09%), but this does not always occur, due to developers' imperfect use of both semver version constraints and semver version number increments. Our findings have implications for developers and researchers alike. We make our infrastructure and dataset publicly available under an open source license.**

I. Introduction

Modern software development relies inextricably on open source package repositories on a massive scale. For example, the NPM repository contains over two million packages and serves tens of billions of downloads weekly, and practically every JavaScript application uses the NPM package manager to install packages from the NPM repository. As open source package repositories grow in scale, the maintenance, updating and distribution of packages represents a growing attack surface for malicious actors to target, and understanding the properties of the software supply chain is vital.

Historically, a critical concern of security is how easily downstream packages can update their dependencies to newer versions that patch vulnerabilities. NPM and other similarly-designed ecosystems (PyPi, etc.) offer a potential solution in the form of semantic versioning ("semver") and flexible version constraints. In semver, versions are numbered in the form `major.minor.bug`, where `major` denotes a breaking change, `minor` denotes a non-breaking change adding new functionality, and `bug` denotes a backward compatible bug fix. Flexible version constraints allow developers of downstream packages to specify which types of updates they are willing to automatically accept. Ideally, semver allows non-breaking important updates (such as security patches) to flow rapidly to downstream packages, while breaking changes are delayed until developers choose to accept them. For example, a developer may specify that they depend on the package `react`, with constraint `^18.1.1`, which will allow for any updates until version `19.0.0` to be installed. In essence, this constraint says "receive all updates to React that are unlikely to be breaking changes".

However, there are three significant complications with semver in practice. First, the positive properties of semver are predicated on both upstream developers labeling their updates with the correct semver increment type, and on downstream developers using constraints that are neither too flexible nor too strict. Second, dependencies in the middle of a transitive dependency chain affect the final received versions of dependencies. The downstream developer may list a constraint that allows the most up-to-date version of a package, but if a transitive dependency has a more restrictive constraint, the downstream developer may not receive the up-to-date version. Third, allowing for automatic (bug) updates to dependencies can be dangerous, as it introduces an attack vector for malware.

In this work, we aim to understand how developers make use of dependencies, semantic versioning, and flexible version constraints at the ecosystem scale, and how all these factors intersect to affect developer experience and supply chain security. Prior work on mining data from the NPM ecosystem has primarily focused on answering questions about NPM at a snapshot in time [1], [2], [3], [4]. In this work, we first understand how developers make use of semantic versioning by analyzing flexible constraint type frequency and semver increment type frequency over the history of NPM. Then to understand how updates flow in practice at the ecosystem scale, we run large-scale experiments where we solve packages' dependencies at multiple different snapshots in time and observe how long it takes for updates to be received by downstream packages. To enable these experiments, we built a

tool which allows for accurate time-travel dependency solving throughout the history of NPM.

In total, we have built the first dataset of NPM that includes (as of October 31, 2022):

1) every package on NPM (2,663,681 packages)
2) every version of every package (28,941,927 versions)
3) both metadata ($\approx$ 40 GB compressed) and packaged code ($\approx$ 19 TB compressed) for every version of every package,
4) full data of security advisories issued for NPM packages, downloaded from the Github Security Advisory database.

This dataset is indexed to allow for easily writing queries and large-scale distributed computations against, oriented towards use in HPC clusters. To gather this data, we designed and implemented a distributed system for downloading, archiving and retrieving packages from NPM. We release our crawler and dataset under the BSD 3-Clause license [1].

We use our dataset to answer several about the NPM ecosystem, in particular how developers use of semantic versioning, and how this affects supply chain security. Specifically, we answer the following research questions:

- **RQ1**: Do developers specify dependency version constraints to allow for automated updates?
- **RQ2**: Do developers use semantic versioning in their package releases to allow for automated updates to downstream packages?
- **RQ3**: Do packages frequently contain out-of-date dependencies? And when updates are published, how long does it take for those updates to be received by downstream packages?
- **RQ4**: Do code changes differ between different categories of semver updates? How often do developers only change metadata in updates?

We believe our results are impactful for both developers and researchers, and show that generally the NPM ecosystem is effective in terms of efficient distribution of non-breaking updates, but most packages end up with out-of-date dependencies anyways due to the sheer volume of dependencies and updates to deal with. In addition, we found evidence that some developers use semver non-optimally when releasing patches to security vulnerabilities.

II. Methodology

At a high-level, we answer our four core research questions using different aspects of our dataset and analysis systems. RQ1 and RQ2 are answered purely via analysis of our scrapped metadata. Answering RQ3 is more challenging as it requires reasoning about how dependencies are solved across time, which we answer by using our time-travelling dependency solver in large-scale experiments. Finally, to answer RQ4 we compute diffs between tarballs of package versions.

[1]Due to hosting requirements, we are unable to *anonymously* post the 19TB dataset before publication, but will share it after double-blind review, working with partners to make this resource easily available for researchers

A. RQ1: Version Constraint Usage

Within NPM's rich language for specifying version constraints on dependencies [5], [6], it is unclear which of the many constraint types developers frequently make use of and how loose or restrictive those constraints are.

We classify version constraints in the following mutually exclusive categories:

1) Exact constraints (`"=1.2.3"`) accept no versions other than the specifically listed one;
2) Bug-flexible constraints (`"~1.2.3"`) accept any updates to the bug semver component, so `1.2.4`, etc.;
3) Minor-flexible constraints (`"^1.2.3"`) accept any updates to the minor semver component, so `1.3.0`, etc.;
4) Geq constraints (`">=1.2.3"`) accept any versions greater than or equal to the specified version;
5) Any constraints (`"*"`) accept any versions; and
6) Other constraints, consisting of all other constraints, such as disjunction, conjunction, GitHub URLs, etc.

and then examine frequencies of these constraint categories across all of NPM, segmented by year so we can observe how constraint usage has evolved historically. In addition, one challenge with analyzing data from NPM is that some packages publish a massive number of versions (React has over 1,000 versions), so aggregating across all versions may produce results that are biased towards packages that upload more often. In RQ1, when we segment by year, we select only the most recent version of every package that was uploaded witin that year. This enables us to segment by time while avoiding this bias.

B. RQ2: Semantic Versioning in Updates

We now turn to examine the use of semantic versioning on the supply side: how do developers increment their semantic version numbers when publishing updates? To answer this question, we first find all of the package updates that have occurred in NPM's history, observing for each if the developer incremented the bug component (e.g. `5.4.8` $\rightarrow$ `5.4.9`), the minor component (e.g. `5.4.8` $\rightarrow$ `5.5.0`), or the major component (e.g. `5.4.8` $\rightarrow$ `6.0.0`).

Given the full version history of all packages, one would expect that an update can trivially be identified as two consecutive versions of the same package. NPM however allows version numbers to be published non-chronologically. This feature is often used by packages to maintain multiple parallel version branches at once, for example uploading `1.0.0`, then `2.0.0`, `1.0.1`, and `2.0.1`. In this example, the mined updates should consist of `1.0.0` $\rightarrow$ `2.0.0`, `1.0.0` $\rightarrow$ `1.0.1`, and `2.0.0` $\rightarrow$ `2.0.1`, as these reflect chronologically and numerically ordered update paths where each destination version is based most closely on the source version. To determine the set of updates, we group versions by the equivalence relation of same major component (`1.0.0` and `1.0.1` are a group), and expect groups to be ordered within themselves chronologically. We then have updates between versions within each group, and between different groups.

From observing real packages, we believe this algorithm best reflects how developers use semantic versioning.

With all updates and version increment types identified, we examine the distribution of the three update types across the whole population, and then compare to the subgroups of updates which introduce and patch vulnerabilities. Updates which patch vulnerabilities are identified directly in the scraped advisory database, while we identify updates which introduce vulnerabilities as the update which creates the version which is the minimal version containing that vulnerability. To avoid the bias introduced by some packages have a large number of updates, our top-level aggregation is among packages rather than updates. For each package, we identify which percentage of its updates are of each type (segmenting by security effect), and then visualize this percentage across all the packages. This enables us to make conclusions about how packages and package developers generally handle incrementing semver numbers during updates. In addition, note that when segmenting by updates which introduce vulnerabilities, we are *not* attempting to study malware, rather updates that (probably inadvertently) create a vulnerability.

C. RQ3: Out-of-Date Dependencies and Update Flows

The properties examined thus far have been local properties of each package, in that each package has been analyzed individually. We now wish to answer how out-of-date NPM packages typically are, and how long it takes updates to flow to downstream packages. Both of these properties rely on all the packages in the transitive dependency closure of a downstream package. However, reasoning about how dependencies are solved is challenging both because NPM's dependency solving algorithm is complex, and because we wish to parameterize this over time.

In order to compute solves accurately and at different points in time, we use vanilla NPM's solver combined with a proxy that emulates the world state at any given point in history. With this key tool, we then perform two experiments: first we solve the dependencies of the most recent version of every package in NPM and observe how many packages have out-of-date dependencies; we then explore how updates flow to downstream packages by solving the dependencies of the downstream package at different points in time until it receives the update.

D. RQ4: Analyzing Code Changes in Updates

After having examined how developers use constraints and version numbers in isolation, we next align that with a high-level characterization of what updates actually change. For every identified update, we decompress the packaged code from both versions, and look for file changes. We then classify changes as modifying dependencies, code (`.js`, `.ts`, `.jsx`, `.tsx`), both, or neither. We then examine the distribution of these types of changes segmented by semver increment type, again normalizing per-package to avoid biasing towards packages with more updates.

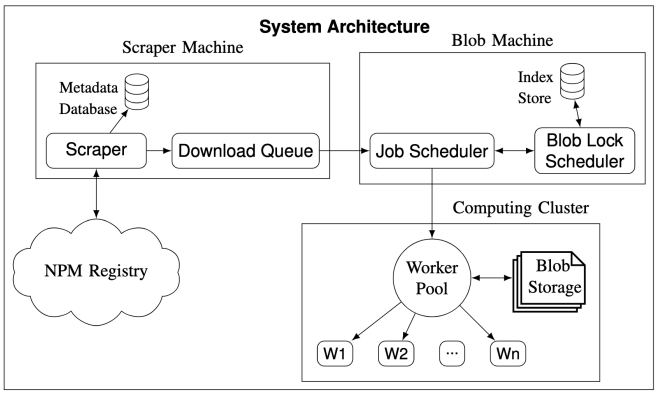


Fig. 1: Overview of our system architecture.

Analyzing at a deeper level is possible with our dataset, but is beyond the scope of this paper. Note that many packages upload compiled or minified JavaScript code, which makes it difficult to even look at simple line-by-line diffs. In addition, we could have chosen to count other file types as code (`.sh`, etc.), but we chose to focus on JavaScript code.

III. System Architecture

In order to construct the dataset described, facilitate computations over it, and carry out our experiments we developed several key components:

1) The crawler, which both downloads old updated and continually streams new updates in realtime from NPM and queues them for processing;
2) The metadata database, which indexes all the changes from the crawler to enable fast and easy access to package metadata, including version history and dependencies;
3) The tarball blob store, which is a distributed storage system to store all tarballs of every version of every package; and
4) A time-travelling variant of the NPM command line tool, which enables solving a package as if the state of the ecosystem were in the past.

We now explain the implementations of these core pieces.

A. Crawler and Metadata Database

NPM stores their metadata in a CouchDB database. CouchDB is a document-oriented database that stores data in JSON documents. CouchDB is a good fit for NPM because it is schemaless and allows for arbitrary nesting of JSON objects, such as the `package.json` file. For performing data analysis we find it to be a poor fit due to the extremely loose structure. There is almost no enforced validation of the format of `package.json` files in the CouchDB, making it impossible to use for analyses without first cleaning and better structuring the data.

We found it easiest to design our own metadata database using PostgreSQL, a relational database management system. We've designed the database to be highly structured and easy for researchers to query, while being an accurate historical

representation of the NPM database, keeping any data that NPM may delete or modify. Finally, we designed it to allow for continual scraping of data from NPM as packages are published, with sufficient throughput to keep up with the rate of package publication. The continual scrapping is accomplished through the use of NPM's `changes` endpoint, which pushes new metadata information as well as URLs of tarballs to download to the other components in the system.

B. Tarball Blob Storage

Storing tarballs of every version of every package is challenging, as there are over 28 million tarballs to store. This leads to a large number of small files, which is a problem for many file systems. For instance, when using the ext4 file system, the maximum number of inodes is quickly reached when storing millions of tarballs. Additionally, we don't have the flexibility of choosing our own file system, as we are using a high-performance computing cluster's file system, NFS.

Because of these limitations, we chose to store tarballs in a custom-built distributed blob storage system. This system is composed of three components: 1) a directory, with a thousand files, which is stored on the cluster's NFS system; 2) a Redis-based blob-index store, which maps from package names to the file and byte index in the directory; and where the tarball is stored. 3) a lock scheduler, which is responsible for locking a blob file in the directory for writing. This system allows us to store millions of tarballs, and to efficiently retrieve them for computation on worker nodes inside the cluster. An overview of how the system interacts together is shown in Figure 1.

C. Time-Travelling Dependency Solver

In order to carry out our experiments outlined in Section II, we needed to be able to observe how a package's dependencies would have been solved at arbitrary points in NPM's history. We built a simple proxy server than can be used with vanilla NPM to enable time-travel dependency solving.

NPM's command line tool enables the user to specify a custom package registry to use in place of `npmjs.com`. To use our time-travelling solver, we specify a registry base URL pointing to our proxy server that includes in the URL the timestamp at which we wish to solve the dependencies. The proxy server then receives both the timestamp as well as a request to fetch metadata for a given package. The proxy server can then simply rewrite the response from `npmjs.com` to remove versions of packages prior to the timestamp.

IV. Results

At a high level, we would consider a package ecosystem to be healthy with regards to update distribution when updates that are positive (performance improvements, bug fixes, security patches, etc.) can be quickly and easily adopted by downstream dependencies, while disruptive changes (breaking changes, security vulnerabilities, malware, etc.) flow more slowly and/or require explicit downstream opt-in. In NPM, the flow of updates is generally determined by two factors:

- **RQ1**: Do downstream developers specify version constraints for dependencies such that updates can be received automatically?
- **RQ2**: Do upstream developers increment their version numbers appropriately when releasing updates?

We start by explaining the overall structure and general properties of the dataset. Then we move on to discuss RQ1 and RQ2 separately, and finally we consider how RQ1 and RQ2 intersect in practice in the ecosystem (**RQ3**), and how they are related to the actual contents of the updates (**RQ4**).

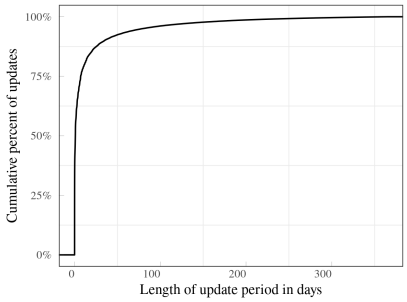
A. Dataset Structure and General Properties

As discussed in Section I our collected data is split into two parts: 1) **Ecosystem Metadata**: This includes the full list of packages (2,663,681 packages), versions of every package (28,941,927 versions), and metadata for every version including version upload times, version numbers, dependencies, descriptions, links to repositories, and more. We also have a full scrape of all security advisories for NPM packages, scrapped from the GitHub Security Advisory Database, which specify which versions of packages are vulnerable, and which versions patch the vulnerability. 2) **Tarballs of published packages**: The full source tarball of every version of every package (other than deleted content) has been downloaded and indexed by our system.

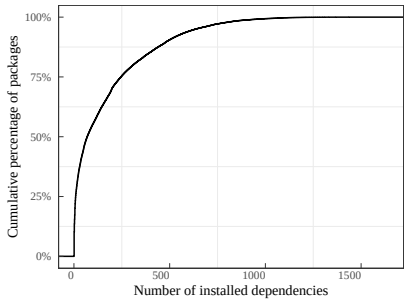
Before diving into the core research questions, we first discuss the general properties of the dataset, which we hope will contextualize the results shown afterwards. Figure 2 displays three distributions regarding our main objects of interest: updates and dependencies.

Figure 2a displays an ECDF of the distribution of the time between updates of packages, excluding updates to prerelease (e.g. `1.0.2-beta3`) as those are generally not intended for downstream packages to consume. A surprising finding is how quickly updates are pushed out in many cases, with 25% of updates spanning only 39.87 *minutes* or less, and 50% of updates spanning 22.71 hours or less. However, a long tail of updates exists, with the top 25% of updates spanning 7.78 days or longer, and 10% spanning 40.12 days or longer. On average, updates span 21.03 days. A manual inspection of the data suggests that update behavior is quite bursty, with developers releasing multiple updates in rapid succession, and then going silent for long periods of time, however this hypothesis should be investigated more thoroughly.

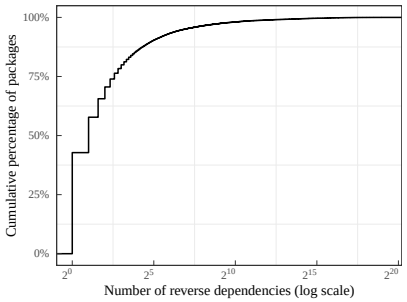
Determining the transitive dependencies of a package statically is difficult to do, as ultimately it relies on the specifics of NPM's solving algorithm and is inherently non-local to each package. Figures 2b and 2c display ECDFs of the distributions of the numbers of (transitive) dependencies and downstream packages (i.e. transitive reverse dependencies), respectively, harvested from actual executions of NPM's dependency solver on over 690,000 packages. We excluded packages which never received an update and thus are considered unmaintained, and some solves failed or timed-out. Our data suggests that on average packages have 167.87 dependencies, and 95% of packages have solution sizes of 636 or fewer dependencies,



(a) An ECDF of the time in days between the publication of two versions of a package. Note that this plot specifically excludes updates for non-prerelease versions of packages.



(b) An ECDF of the number of (transitive) dependencies of each package. This was collected from solving the latest version of every package on NPM as part of the experiment in Figure 5.



(c) An ECDF of the number of reverse (transitive) dependencies of each package. Note that the x-axis is log-scaled. This was collected from solving the latest version of every package on NPM as part of the experiment in Figure 5.

Fig. 2: ECDFs plots visualizing distributions of general properties of the NPM ecosystem with regards to versioning and dependencies.

with the largest solutions reaching up to 1641 dependencies. We hope that this gives an approximate sense of the scale of dependency solving in NPM, and hints at what sizes of solutions research on tooling for package management must support. When turning to downstream packages however, the situation is quite asymmetrical, as there is a vastly longer tail of packages with massive amounts of downstream packages. The top 3 depended-upon packages that we observed were: 1) `supports-color` (does a terminal support color?, 624,883 downstream packages), 2) `debug` (logging library, 571,547 downstream packages), 3) `ms` (time conversion library, 515,684 downstream packages). On the other hand, a large amount of packages are unused except by a handful of downstream packages, with 50% of packages having 2 or fewer downstream packages, and 90% only being used by 30 or fewer downstream packages.

B. RQ1: Version Constraint Usage

We now look at how developers tend to specify constraints. Figure 3 shows the frequency of each main type of version constraint in each year since 2010, the year that NPM launched.

There are several interesting trends in constraint usage over time. First, about 78.36% of all initial dependencies were specified as accepting any versions greater than some particular version (Geq, purple bars), such as `"react" : ">= 1.2.3"`, but developers abandoned using Geq constraints within the first 3-4 years of NPM. Geq constraints generally become unmaintainable as they will automatically update to major new versions of packages, which typically contain breaking changes. Developers in the early days of NPM likely realized this after a few years of building the ecosystem. Second, even though constraints which are flexible in the minor component (Minor, green bars) currently represent a majority of dependencies, the phenomenon of using minor flexible constraints only started in 2014, and then rapidly

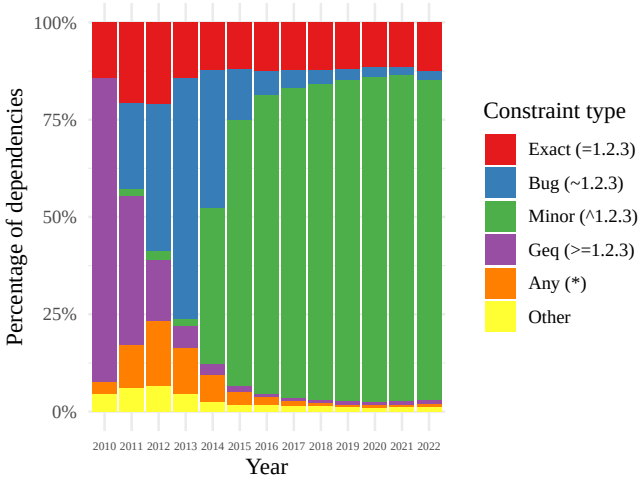


Fig. 3: The relative popularity of each version constraint type across time. For each year, we extract the dependencies from the most recent version of every package within the year, and compute the frequencies of constraint types across the year. The percentages are thus *not* cumulative over past years, but only reflect published dependencies within each year.

expanded after. The expansion of minor flexible constraints coincides with the decreased usage of bug component flexible constraints (Bug, blue bars). Third, in the last four years of the NPM ecosystem, developers have gravitated towards using only two types of constraints almost exclusively: constraints which require exact versions (Exact, red bars) and minor component flexible constraints. Together, those types represent over 94.85% of constraints in 2022. Finally, the percentage of dependencies which are potentially able to automatically receive updates (everything below the red bars) has stayed relatively stable throughout the entire life of NPM, and is

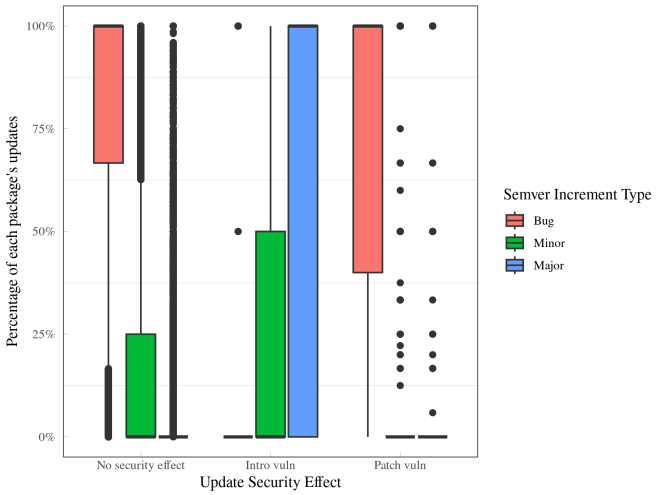


Fig. 4: A boxplot visualizing the distribution of percentages of packages' updates by semver increment type, segmented across security effects. Within each security effect the percentages across semver increment types are normalized.

currently about 87.32% of all dependencies.

C. RQ2: Semantic Versioning in Updates

We now examine if developers increment the semantic versioning of their packages so as to enable easy automated updates for downstream packages. Figure 4 displays boxplots where each observation represents what percentage of a package's updates are one of the three semver increment types, normalized across security effect. We find that in the no security effect category (the vast majority of updates), the most common update type by far are bug semver increments, with 75% of packages having 66% or more. Next most popular are minor semver increments, and finally least most popular are major semver increments.

However, when we consider updates which introduce vulnerabilities, we see a different story. Most updates which introduce vulnerabilities are major semver increments, indicating that vulnerabilities are often introduced when packages developers release major new versions possibly consisting of many new features and significant structural changes to the code base. We did however find 29 outlier packages which introduced a vulnerability in at least one bug update (either it was the only update introducing a vulnerability, or it was one of two updates). A particularly interesting example is an update to the `ssri` package (a cryptographic subresource integrity checking library, 23M weekly downloads) from version 5.2.1 to 5.2.2. The update attempted to patch a regular expression denial of service vulnerability, but inadvertently increased the severity of the vulnerability by changing the worst-case behavior from quadratic to exponential complexity [7]. This highlights the challenge package developers face in needing to quickly release patches to vulnerabilities, while needing to be extremely careful when working on security-relevant code and releasing it through bug updates that will be easily distributed to downstream packages.

Finally, in the case of vulnerabilities being patched, almost all patches are released as bug semver increments, which means that the 87.32% of non-exact constraints shown in Figure 3 would potentially be able to receive them automatically. However, a handful of outlier packages have released vulnerability patches as non-bug updates (we found 358 such updates). From manual inspection, it appears that many of these updates include the fix for the security vulnerability mixed in with many other changes, rather than the vulnerability fix being released independently. For example, update 1.6.0 to 1.7.0 of the `xmlhttprequest` package (1.2M weekly downloads) fixed a high-severity code injection vulnerability [8]. The security-relevant part of the update is only 1 line, but 892 lines were modified in the update. Without further investigation we do not know why some developers have chosen to package vulnerability patches as part of larger updates rather than as standalone updates.

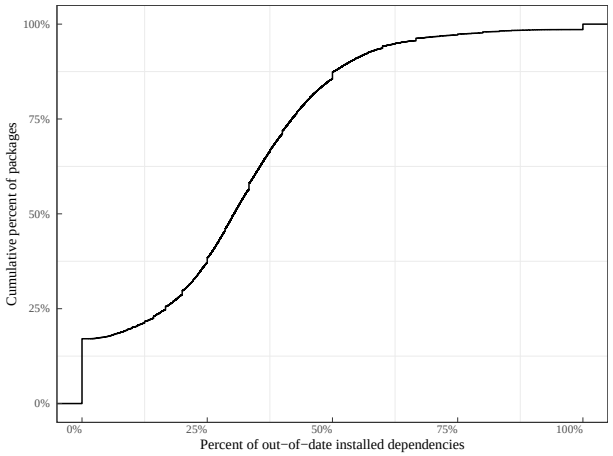
D. RQ3: Out-of-Date Dependencies and Update Flows

1) How out-of-date are packages' dependencies?: Version constraints and semver update types work in tandem to control the flow of updates to downstream packages, across many chains of transitive dependencies. Whether a downstream package receives up-to-date packages depends not only on the constraints at the downstream package and the type of semver increment at the reverse dependency, but also on packages in the middle.

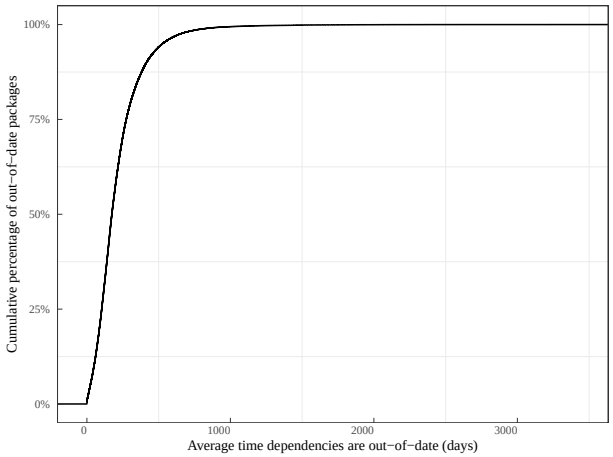
We first take a look at generally how out-of-date packages' dependencies are on NPM. Figure 5a displays an ECDF of the distribution across packages of the percentage of each package's dependencies that are out-of-date. There is a group of packages, about 17.08% of the population, that have fully up-to-date packages. However, these are almost all packages that have very few dependencies, only 3.17 dependencies on average compared to 167.87 dependencies for the whole population. In other words, these fully up-to-date packages are packages that live primarily on the far left side of the ECDF in Figure 2b.

Moving beyond the spike of up-to-date packages, most packages have at least some out-of-date dependencies, with 62.94% of packages having 25% or more of their dependencies out-of-date. Not only do packages often have out-of-date dependencies, but they are often out-of-date for quite a while. Among packages that have at least one out-of-date dependency, Figure 5b displays an ECDF of on average how out-of-date each package's dependencies are. Half of all packages with out-of-date dependencies have on average dependencies that are 173.87 days old or order, with a long tail of 5% of packages with dependencies that are on average 527.38 days old or older. In contrast, updates are released within 21.03 days on average, and 50% are released within only 22.71 hours (Figure 2a).

There can be a variety of reasons why packages have out-of-date dependencies, many of which are benign, such as



(a) ECDF of percentage of each package's dependencies that are out-of-date.



(b) ECDF of average amount in days that dependencies are out-of-date by for each package with at least one-out-of-date dependency.

Fig. 5: In this experiment, we select the latest version of every package on NPM, and use our time-travelling NPM to simulate solving the package's dependencies at the time the latest version was uploaded (T_P). We then observe which of its installed dependencies are out-of-date, where a dependency is out-of-date if the upload time (T_D) of the selected dependency version (V_D) is older than another higher-version-numbered (non-beta) version existing prior to time T_P, that is, another version V'_D of the dependency has an upload time T'_D such that $T_D < T'_D < T_P$ and $V_D < V'_D$. We then define the out-of-date time as $T'_D - T_D$ for the largest such T'_D. We plot in Figure 5a an ECDF for the distribution of percentage of each package's dependencies that are out of date, and in Figure 5b an ECDF for the distribution of mean out-of-date times for each package's dependencies, for packages that have at least one out-of-date dependency. In total, 696,419 solves were performed successfully.

developers choosing to stay on older versions of libraries rather than rewrite code to handle breaking changes. Let's now filter away cases where packages already depend on old dependencies, and focus on how rapidly updates flow to downstream packages which are up-to-date.

2) How rapidly do updates flow downstream?: Figure 6 gives an overview to understand qualitatively what happens when an update is published to NPM, and how it flows to downstream packages. First, the update developer publishes the update with a certain semver increment type (major, minor or bug). As seen in Section IV-C, developers almost always categorize updates that patch vulnerabilities as semver bug updates, while updates that introduce vulnerabilities are typically marked as minor or major. Note that Figure 6 shows that more vulnerability-introducing updates are marked as minor (68.42%) than major (10.53%), while Figure 4 shows that more packages mark vulnerability-introducing updates as primarily major. This occurs because the number of versions per-package varies widely. As for security-neutral updates, they are majority bug updates (65.90%), somewhat frequently minor updates (29.41%), and occasionally major updates (4.69%).

Once the update is marked as bug, minor or major and uploaded to NPM, it can then be consumed by downstream packages that depend on it, possibly transitively. This can happen in several ways. Most commonly, downstream packages can receive the update instantly and with no human intervention needed ($\cdots \rightarrow$ no intervention $\rightarrow$ instant update).

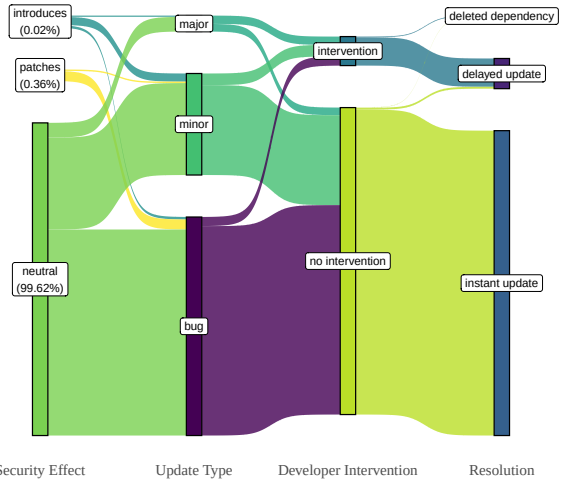


Fig. 6: This experiment examines how updates flow mechanically to downstream packages, and how often developer intervention is required in order to receive the update. For the most recent update prior to 2021 of every package, we randomly selected 50 potential downstream packages. Using our time-travelling we then solve the downstream package immediately before the update, immediately after the update, and in 1 day increments afterwards until the dependency on the old version of the dependency has been updated or deleted. In total, 888,294 solves with NPM were performed successfully.

This occurs when the package that declares the constraint on the updated package uses a constraint which is at least as flexible as the type of semver increment. Note that the package declaring the constraint, and thus responsible for allowing or inhibiting the update flow, could be either the final downstream package or a package in the middle of the dependency chain. This type of flow occurs for the majority of bug and minor updates, which is induced by the distribution of constraint types (Figure 3) typically allowing both bug and minor updates. As this type of flow is 90.09% of all analyzed update flows, it is by far the most common, indicating overall positive health of the NPM ecosystem.

The second largest update flow consists of updates that require intervention from the developer of the downstream package (and possibly developers of other packages as well), and thus is delayed ($\cdots \rightarrow$ intervention $\rightarrow$ delayed update). This occurs in 9.01% of all analyzed update flows, and involves a major update 28.11% of the time, a minor update 40.27% of the time, and a bug update 31.62% of the time. Updates requiring intervention are due to developers constraints which are more restrictive than the semver increment type of the update. Intervention thus involves developers either switching to more flexible constraint types or incrementing the constrained version range. Additional investigation could be done to see how common each type of dependency change is.

A small fraction (0.60%) of updates are resolved not by the developer of the downstream package performing an intervention, but by a developer(s) in the middle ($\cdots \rightarrow$ no intervention $\rightarrow$ delayed update). For this to occur, the developer of the package in the middle must have specified a constraint that is too restrictive, while the developer of the downstream package specified a flexible enough constraint to allow for the intervention of the package in the middle to be adopted. Since this type of flow happens very rarely, this indicates that downstream packages typically have constraints that are equally or more restrictive than their (transitive) dependencies. This makes sense from a software engineering perspective, as the more downstream packages (those closer to applications rather than libraries), have more of an incentive to keep dependencies fixed at versions they know work, while library developers (upstream) don't want to force their consumers onto old versions of dependencies.

The final type of flow is when the out-of-date dependency is eventually deleted rather than updated ($\cdots \rightarrow$ intervention $\rightarrow$ deleted dependency). This occurs in only 0.29% of all analyzed update flows, indicating that developers do not generally delete dependencies. More investigation could be done to understand why developers choose to delete dependencies in a small number of cases.

Among the update flows which are blocked due to restrictive constraints, almost all update flows are unblocked via manual intervention quite rapidly. Figure 7 shows an ECDF of the distribution of how many days it takes for each update flow to be unblocked.

The majority of blocked update flows (91.74%) are unblocked within 1 day, with a tail trailing off to 25 days or more. The surprising speed of update flows being unblocked is due largely to the fact that many packages that depend on each other are developed by the same contributors, and they will often bump version numbers and update dependencies of their packages nearly simultaneously.

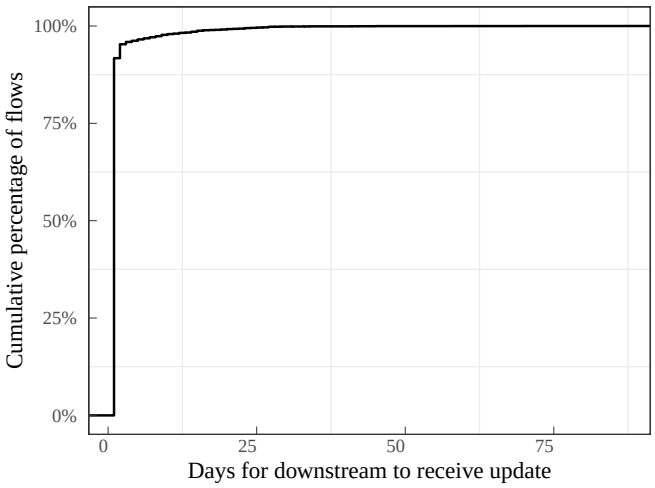


Fig. 7: An ECDF plot of how long it takes for an update flow which is blocked to be resolved.

Combined with Figure 5, we see that from the update point of view, updates generally flow quite effectively to downstream packages, and yet from the downstream package point of view, most downstream packages have at least some out-of-date dependencies. More investigation should be carried out on this phenomonon, but we suspect this is due in part to the number of dependencies per package (Figure 2b) and rate of updates (Figure 2a). With average packages having around 167 dependencies, and updates being released every 21 days on average, we would expect that for an average package, every day multiple dependencies are releasing updates and potentially going out-of-date, so even with many updates being adopted instantly or quickly, some dependencies will become stale. In addition, in this experiment we selected for packages that are already up-to-date to measure rate of update flow in a more controlled manner.

E. RQ4: Analyzing Code Changes in Updates

We now turn to inspecting the *contents* of package updates rather than metadata analysis. Semantic versioning can only be useful if package developers release updates which are in accordance with what downstream packages expect from bug, minor, or major semver increments. In this paper we focus on providing a high-level characterization of what updates generally consist of in the NPM ecosystem, across the different update types. While fine-grained checking of adherence to semver semantics would be interesting and useful [9], it is beyond the scope of this paper. However, we believe that our dataset may be a useful building block for evaluation within

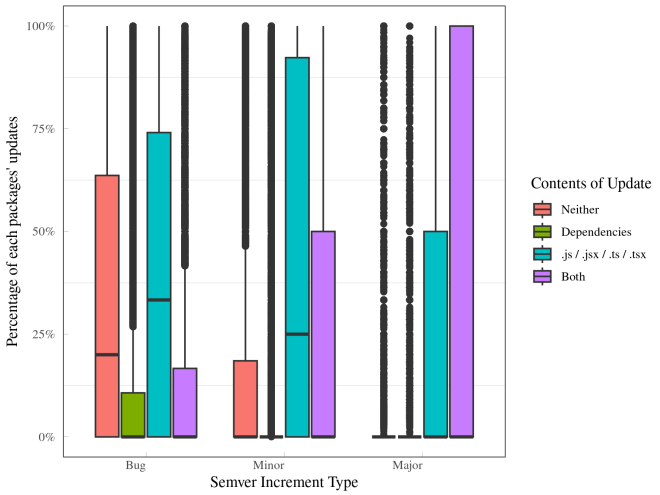


Fig. 8: A boxplot displaying the distribution of the percentage of packages' updates grouped by semver increment type that change only code (`.js, .ts, .jsx, .tsx`), only dependencies, both, or neiter.

the active research area of update analysis systems [10], [1], [11].

Figure 8 displays a boxplot where each observation is the percentage of a package's updates within each semver increment type that change only code (`.js`, `.ts`, `.jsx`, `.tsx`), only dependencies, both, or neither. Note that updates categorized as neither may include other changes such as modifications to other file types (README, CSS, etc.) or other metadata changes besides dependencies.

First, we see that bug updates somewhat often contain no changes to code or dependencies. 50% of packages change neither code nor dependencies in about 20% or more of their bug updates, while 25% of packages change neither in a majority (64%) of their bug updates. A manual inspection of the data suggests that some of these updates consist of changes to metadata (listed contributors, descriptions, READMEs) or to configuration files (`.json`, `.yaml`, etc.), while other updates truly change nothing. However, more investigation on our data could be done to quantify this more precisely. Second, while it is not common to do so, 25% of packages do occasionally release bug updates which only modify dependencies (11% or more of bug updates). As we shift to looking at minor and major updates, the frequency of packages modifying neither or only one or the other decreases, and when looking at major updates, most packages modify both code and dependencies simultaneously.

With growing concern over malware in the software supply chain, analysis tools will likely wish to scan package updates in open-source ecosystems. We hope that the results presented here characterize what the contents of updates usually are, and the type of load such tools would need to handle. Existing work on malware detection in NPM [11], [10], [1] operates primarily at the metadata level, and sometimes incorporating lightweight syntactic features. However, the significant portion of packages which publish bug updates that change neither `js`, `ts`, `jsx`, `tsx` files, nor dependencies suggests that a sizeable portion of updates may be changing other types of files, and such changes may be an effective place to hide malicious changes. Malware analysis may need to analyze changes to not only JavaScript code but any other code or data that is loaded by the package, such as shell scripts, embedded binaries, or even configuration files.

Regardless, the recent work has shown that practical malware detection for NPM is indeed possible, indicating that there may be a fertile design space of malware detection tooling for open-source package ecosystems. However, more fine-grained analyses that consider code in addition to metadata will be challenging, particularly for checking minor and major updates for potential security violations, as they frequently change many things at once (Figure 8). On the other hand, minor and major update types are both more rare than bug update types (Figures 4 and 6) and take longer to flow to downstream packages (Figure 6), so hypothetical security analyses may have infrequent load from complex updates, and may not need to respond fully in realtime.

V. Related Work

The NPM package ecosystem has been studied in a number of ways. Wittern et al. [12] studied dependencies between packages in the NPM ecosystem, and found that the number of dependencies between packages is increasing over time. Kula et al. [2] analyzed the size of packages in the ecosystem, and found that a significant portion of packages are very small and provide few features. Despite the small size of these packages, they are often depended on by many other packages, making them a crucial part of the ecosystem and a potential target for malicious actors, as shown in the Chowdhury et al. study [3]. The Abdalkareem et al. study [4] explored the reasons why these small packages are depended on by so many other packages, and found that they are often used as a dependency for a single function or class, and that developers consider these packages to be well tested and maintained, which was disproven by the authors' analysis. Zahan et al. [1] examined the security of packages in the ecosystem, and proposed six signals that can be used to identify packages that are likely to be malicious. However, these signals are entirely based on metadata, and do not consider the contents of packages. Sejfia et al. [10] designed a malware detection tool for NPM which uses lightweight machine learning models trained on metadata and shallow syntactic features to classify updates as malware. Zimmermann et al. [13] studied threats to the NPM ecosystem, and found that mitigating threats to the NPM ecosystem is challenging due to the large number of packages and the lack of a vetting process for packages before they are published to the NPM registry.

The NPM ecosystem has also been studied in the context of semantic versioning. Cogo et al. [14] analyzed the phenomenon of *downgrades*, which are updates that decrease the version of a dependency. They didn't find any evidence

that downgrades are introduced due to security vulnerabilities, indicating that security vulnerabilities are often fixed by upgrading the dependency that is vulnerable, rather than downgrading it. We suspect that this is because of the `npm audit fix` command, which automatically upgrades vulnerable dependencies to the latest non-vulnerable version if it exists, instead of downgrading them, as shown by Pinckney et al. [6]. Zerouali et al. [15] measured the technical lag between the release of a new version of a package and the adoption of that version by downstream packages. They found that the technical lag is significantly higher for major updates than for minor and bug updates, and that the technical lag is higher for packages with more dependencies. Moreover, Bogart et al. [16] interviewed developers on the stability of dependencies in the NPM and CRAN ecosystems. They found that semantic versioning is being used by developers, but that developers are not always aware of it's implications, and therefore many packages depend on unstable dependencies. However, these works inspected a subset of either the metadata or the code of packages in the NPM ecosystem, whereas our study analyzes both the metadata and the code of packages and packages' updates. Furthermore, we provide infrastructural support for future studies of the ecosystem, allowing researchers to replicate these studies utilizing a more comprehensive dataset.

Lastly, the NPM ecosystem is not the only software ecosystem to have been studied when it comes to semantic versioning. Raemaekers et al. [17] examined the usage of semantic versioning in the Maven ecosystem, and found that semantic versioning rules are not always followed. However, they have showed that the adherence to semantic versioning is increasing over time, indicating that developers are becoming more aware of it's importance. Decan et al. [18] studied the compliance of semantic versioning in multiple ecosystems (Cargo, NPM, Packagist and Rubygems) and found that Cargo is the most compliant ecosystem, followed by NPM, Packagist and Rubygems. However, they did not study the contents of updates, and therefore were not able to determine whether the updates were actually changing the code or not. We believe that our study highlights the importance of semantic versioning in the NPM ecosystem, and that our dataset can be used to further study the stability of dependencies.

VI. Threats to Validity

A. External Validity

While our experiments were ran on all packages currently available on the NPM registry, there are some packages which are not included in our dataset. For example, packages that were unpublished or deleted from the registry are by nature not included in our dataset. Our results may be skewed by the fact that we are only looking at packages that are currently available on the NPM registry. However, we believe that this is a reasonable assumption to make, as we wouldn't be able to analyze packages that were made unavailable on the NPM registry before we started our scraping process. Additionally, we do not make any claims about malware specifically in our analysis, as those are often packages which get deleted. Future work could look at packages that were unpublished or deleted from the registry after our scraping process began, and compare them to packages that are still available on the registry.

A significant portion of the NPM ecosystem consists of small packages [2] containing just a few functions and dependencies. While some may consider these "toy" packages to be an inessential part of the ecosystem, they have in fact played a vital role in it [3]. For this reason, we decided to include toy packages in our dataset. A proposed alternative would be to only look at packages that have a certain number of dependencies, or that have a certain number of downloads per month. Se believe that this would be a mistake, as the results of our study would not represent the ecosystem as a whole, but rather a subset of it. However, some questions only make sense to ask for packages with at least one dependency, such as when considering update flows in Figure 6, and in these cases we filter out such packages.

B. Internal Validity

Our system described in Section III-A and Section III-B has a lot of moving parts, and it is possible that there are bugs in our system that could affect the results of our experiments. For example, we may have missed some packages in our scraping process, or we may have incorrectly downloaded some packages. We believe that this is unlikely, as we have written unit tests for our system and have tested it on a small subset of packages, and have not found any bugs.

While running millions of package installations for RQ3 (Section IV-D) we sometimes failed to install some packages due to intermittent hardware failures on our HPC cluster. However, we believe these failures were evenly distributed across our samples and did not bias our results.

C. Construct Validity

While gathering data for RQ4, we only looked at files that matched the file extensions `.js`, `.ts`, `.jsx`, and `.tsx`. This is because we were only interested in files that contained JavaScript and TypeScript code, and we wanted to avoid analyzing files that contained files which were written in other languages. However, this may have caused us to miss some files that contained JavaScript or TypeScript code, as some packages may use other file extensions or may not use any file extensions at all. Additionally, shell scripts are often used in the NPM ecosystem to run scripts before or after a package is installed, we did not analyze these scripts, but a change in such files may be considered a code change. In general, the notion of what counts as "code" is not well-defined at this large scale, and we leave room for future work to make other choices in analyzing the data.

VII. Conclusion

We present a large-scale analysis of semantic versioning in NPM, and a full, reusable dataset of complete package metadata and tarball data from NPM. We analyze the usage

of flexible version constraints, the patterns of semver increments in updates, how out-of-date packages typically are, how easily updates flow to downstream packages, and how the contents of package updates align with their metadata updates. We segment many of these questions over security effects (introducing and patching vulnerabilities) to understand their relation to supply chain security.

VIII. Data Availability

Our artifact [19] contains: 1) the full implementation of our system, 2) and the full indexed metadata database. Due to hosting requirements, we are unable to *anonymously* post the 19TB dataset of tarballs before publication, but will share it after double-blind review, working with partners to make this resource easily available for researchers.

References

[1] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 331–340. [Online]. Available: https://doi.org/10.1145/3510457.3513044

[2] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "On the impact of micro-packages: An empirical study of the npm javascript ecosystem," 2017. [Online]. Available: https://arxiv.org/abs/1709.04638

[3] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the untriviality of trivial packages: An empirical study of npm javascript packages," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2695–2708, 2022.

[4] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 385–395. [Online]. Available: https://doi.org/10.1145/3106237.3106267

[5] "package.json," https://docs.npmjs.com/cli/v9/configuring-npm/package-json#dependencies. Accessed Jan 20 2023, 2023.

[6] D. Pinckney, F. Cassano, A. Guha, J. Bell, M. Culpo, and T. Gamblin, "Flexible and optimal dependency management via max-smt," 2022. [Online]. Available: https://arxiv.org/abs/2203.13737

[7] B. Caller, "Security advisory: Regular expression denial of service (redos) in npm/ssri," 2021. [Online]. Available: https://doyensec.com/resources/Doyensec_Advisory_ssri_redos.pdf

[8] "Cve-2020-28502 detail," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-28502

[9] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the semantics into semantic versioning," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 157–179. [Online]. Available: https://doi.org/10.1145/3426428.3426922

[10] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1681–1692. [Online]. Available: https://doi.org/10.1145/3510003.3510104

[11] "Snyk open source," https://snyk.io/product/open-source-security-management/. Accessed Jan 20 2023, 2023.

[12] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 351–361. [Online]. Available: https://doi.org/10.1145/2901739.2901743

[13] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Smallworld with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 995–1010.

[14] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2457–2470, 2021.

[15] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse*, R. Capilla, B. Gallina, and C. Cetina, Eds. Cham: Springer International Publishing, 2018, pp. 95–110.

[16] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015, pp. 86–89.

[17] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 215–224.

[18] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2021.

[19] A. Anonymous, "Artifact For A Large Scale Analysis of Semantic Versioning in NPM," Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7552551