Visualising software package dependency graphs for Cyber-Supply Chain Risk Management

Candidate Number 1034803

Degree Course: Honour School of Computer Science
Word count: 9,980

Abstract

Software development teams are increasingly reliant on open-source software libraries (packages), which can, and have, introduced severe security vulnerabilities into private projects. By considering the package dependency network as a supply chain of program code, Cyber Supply Chain Risk Management (C-SCRM) can identify and mitigate these risks. In response to developer surveys, I create a C-SCRM visualisation system which takes a project's current dependencies as input and returns a visualisation of various dimensions of risk, along with revealing how the risk factors would change if a new library were depended upon. I use the NPM registry as a working example due to the large set of freely-available data, and to provide context through previous incidents. I argue my visualisation achieves many features requested by developers, but should have better integration into existing workflows, and is lacking offline functionality.

This report explains the steps required to construct such a visualisation system, both in general and specific to NPM, then discusses the output of my implementation and its applicability to the software development process. I also use the underlying database to reveal interesting properties of the NPM registry, such as 35% of packages having no dependencies, 13% having more than 100 transitive dependencies, and 80% of packages having no dependents in the registry. On average, NPM packages have 15 contributors, 78 transitive dependencies, and a dependency codebase 219x larger than this package's source code.

Contents

List of Figures

List of Tables

1 Introduction

Open-source software (OSS) is software whose source code is freely available to access, modify, and contribute to via pull/merge requests (Open Source Initiative, 2007). OSS is developed by a large international community, many of whom choose to be anonymous, but can usually be trusted thanks to code review processes and general reputation (Murdoch and Leaver, 2015).

Software development teams are increasingly reliant on third-party software libraries, many of which are open-source. The average enterprise application contains over 100 OSS libraries, comprising over 35% of the codebase (Pittenger, 2016). Thousands of software vulnerabilities are added to NIST's National Vulnerability Database each year (NIST, 2021), many of them belonging to OSS. The compatibility issues involved in updating an already-integrated library mean software developers often delay updating an insecure library (Pashchenko, Vu and Massacci, 2020), so the act of choosing between libraries carries significant risk. Furthermore, the warranty disclaimers often found within OSS licenses (Pearson, 2000), and the anonymity granted to developers, mean that the usual security guarantees or background checks employed by critical infrastructure developers cannot apply to OSS (see Section 2.1).

The installation and maintenance of a collection of libraries (or packages) is simplified by package managers: when a package is installed to a local project, it becomes a *dependency* of that project. Packages often have dependencies themselves which must also be installed, creating a global network of package dependencies, of which a small subset is used by one project. Lauinger et al. (2017) show that software vulnerabilities can be introduced through transitive dependencies[1], which could have us believe the (in)security of a codebase is out of the developer's hands entirely. By considering the dependency network as a supply chain of program code, with vulnerabilities in upstream packages affecting all downstream packages, supply chain risk management (SCRM) can be employed to identify and mitigate these risks. This is one instance of Cyber Supply Chain Risk Management (C-SRCM), which can be applied to various types of risk in the computer industry (NIST, 2016).

Pashchenko, Vu and Massacci (2020) interviewed developers about their dependency management and selection practices; they show that developers often 'avoid updating dependencies as they lack resources to cope with the breaking changes' (p. 1520). To aid with this, the developers suggested usage of high-level dependency management tools (p. 1521):

> ***Observation 13:*** *Developers recommend introducing high-level metrics that show that a library is safe to use (security badge), mature, and does not bring too many transitive dependencies.*

In response to these suggestions, and with further motivation from examples in

[1]A *transitive dependency* of package A is a package that A on through transitive closure of the immediate dependencies - i.e. any upstream package in the dependency network.

Section 2, my project aims to provide tools to support developers with C-SCRM. This can be divided into three aims, with the second being the main focus of the project:

- Create a relational dataset of software packages and associated entities (source code contributors, security advisories etc.)
- Create a visualisation tool to help developers understand the risks associated with their projects' dependencies
- Calculate & present aggregate statistics on the dataset to reveal trends in open-source development.

From these aims, I created a C-SCRM visualisation interface, which takes a project's current dependencies as input and returns a visualisation of various dimensions of risk, along with revealing how the risk factors would change if a new library were depended upon. The system required the construction of a data acquisition and processing pipeline, from which visualisations can be made for any combination of packages on the given registry. For example data, I used NPM (Node Package Manager)[2], a package manager bundled with the Node.js runtime, which allows developers to download and manage JavaScript packages hosted on the NPM registry. I used graph database software (Neo4j) to ease the calculation of graph-related attributes such as transitive dependencies and ratios of source code sizes, and constructed numerous visualisation graphs in a web application (using D3.js) for rapid prototyping.

The following report will describe the development of this system and its applicability in the software engineering industry. Further definitions and context will be given in Section 2, and Section 3 contains a brief collection of research surrounding software security. I give an abstract design of a C-SCRM visualisation system in Section 4, which is applied to specific technologies in Section 5. Various outputs of the system are given in Section 6 and discussed in Section 7, and I assess the successes and limitations of such a system in Section 8. Finally, Section 9 recaps the contributions and gives avenues for further research. In some sections, I found it helpful to split the discussion into subsections for Data acquisition (sourcing data for packages and relevant entities), Processing (the manipulation of the acquired dataset) and Visualisation (presenting the information to the end user).

2 Background

To provide some background and motivation for the project, I will briefly discuss the relevance of C-SCRM in the software engineering industry, and why NPM is a good example for this field.

[2]The package manager and the organisation are officially titled 'npm', which I write as NPM to disambiguate between the organisation and the `npm` program.

2.1 C-SCRM

The US National Institute of Standards and Technology (NIST) runs a research project for C-SRCM, which mainly focuses IT systems rather than software, but presents some useful terminology. In their project overview (NIST, 2016), they categorise C-SCRM threats as either 'adversarial (e.g. tampering, counterfeits)' or 'non-adversarial' (e.g. poor quality, natural disasters)'. They categorise vulnerabilities as 'internal (e.g. organizational procedures)' or 'external (e.g. part of an organisation's supply chain)'. This project concerns both adversarial and non-adversarial *external* threats, examples of which are given later in this section.

In 2014, NIST first released their Framework for Improving Critical Infrastructure Cybersecurity (NIST, 2018) to 'help an organization to align and prioritize its cybersecurity activities with its business/mission requirements', which has a specific category for Supply Chain Risk Management. They make the following recommendations (p. 28):

> ***ID.SC-2:*** *Suppliers... of information systems, components, and services are identified, prioritized, and assessed using a cyber supply chain risk assessment process.*
>
> ***ID.SC-4:*** *Suppliers... are routinely assessed using audits, test results, or other forms of evaluations to confirm they are meeting their contractual obligations.*

The growing adoption of OSS makes NIST's recommendations more difficult to enforce: contributors to OSS aren't under any 'contractual obligations', and usually aren't even known to the software developers using the packages.

2.2 NPM

I chose to focus on NPM for a variety of reasons. It's the default package manager for an already-popular programming language[3], and as a result the NPM registry is the largest package manager registry in the world (Brown, 2017). Additionally, data about the packages hosted on the registry is easy and free to acquire, through various REST API endpoints and data streams (see Section 5).

NPM also offers sobering examples of supply chain risks becoming major incidents, as documented by Petrik (2020). In March 2016, a developer unpublished all of their packages from the registry, making them inaccessible to any dependent projects. One such package, `left-pad`, was depended upon by thousands of projects (@izs, 2016), which after this un-publishing were unable to operate, causing major disruption. After a controversial 'un-un-publishing' by the administrators, dependent systems could download the package and continue

[3]In their annual survey, Stack Overflow (2020) find that 52% of professional developers are using Node.js to some extent in their work.

operation. This is an example of a non-adversarial risk, according to NIST.

For an example of an adversarial risk: in September 2018, the new owner of the widely-used `event-stream` package pushed an update[4] that introduced a new dependency, which was found in November 2018 to contain hidden functionality to steal cryptocurrency wallet keys (GitHub, 2018), (@adam-npm, 2018). Regardless of whether the contributor was knowingly introducing this malware, it shows the relative ease of spreading malware to large numbers of projects through transitive dependencies.

3 Related work

This section is divided into a few topics of relevance to the project: vulnerability prediction (assessing the security of source code), C-SCRM (assessing the security of third-party packages), and visualisations of various kinds.

3.1 Vulnerability prediction

While only a small aspect of C-SCRM, vulnerability prediction in program source code is a well-researched field, resulting in numerous static code analysis tools being available to software developers. Díaz and Bermejo (2013) assessed 9 such static analysis tools against a dataset of insecure code shared by NIST (2009). Díaz found the tools had an average precision ('ratio of correctly detected vulnerabilities to the number of all detected vulnerabilities') of 70%, with the best tool having 93% precision.

Bilgin et al. (2020) describes the construction of a vulnerable code classifier by reducing source code to an abstract syntax tree (AST), then training models on the Draper VDISC dataset (Kim, 2018). The performance/recall of the models varied depending on the CWE examined, but is comparable to Díaz's findings. Hovsepyan et al. (2012) also trained a vulnerability prediction model, instead converting source code into a 'feature vector' of code keywords (monograms) with their frequencies. This conversion gave their model a prediction accuracy of 87%.

However, all of the above methods rely on having a local copy of the source code available. Such methods might be infeasible for large programs (or indeed for examining an entire package registry), so methods have been explored that instead use metrics of the code.

Sultana (2018) compared vulnerability prediction models using 'traceable code patterns' (coding patterns specific to one programming language) and 'software metrics' (coarse attributes of a program's methods/classes/files), and found that *"class-level metrics and method-level metrics outperform micro patterns and nano-patterns respectively in terms of precision"* (p. 88).

[4] `https://github.com/dominictarr/event-stream/commit/e3163361`

Similarly, Shin (2011) predicted vulnerabilities from ‘complexity metrics’, such as the number of lines of code (LoC/SLoC), comment density and the number of conditional statements in each function. They found that 25 of their 28 complexity, code churn and developer activity metrics *“supported the hypothesis for discriminate power between vulnerable and neutral files”* (p. 80). Gegick, Rotella and Williams (2009) also predicted vulnerable program components using two candidate metrics (warning frequency from existing static analysis tools, and SLoC); their models have an average accuracy of 90%.

Lastly, Nguyen and Tran (2010) predicted vulnerable components in Firefox’s JavaScript Engine. Instead of using metrics or textual analysis, they constructed dependency graphs of Members (functions or variables of a component) and Components (a class of variables and member functions). Training a model on the Component graph greatly reduced the false negative rates compared to Shin and Laurie Williams (2008)’s classifier (based on nesting level), showing promise for the use of dependency graphs in risk management.

3.2 C-SCRM

Next, I look at work undertaken to analyse vulnerabilities and risk in third-party packages. To firstly justify my concern about dependency networks, I refer to Lauinger et al. (2017)’s study of the behaviour of client-side JavaScript libraries and transitive dependencies. As well as finding 37% of websites serving outdated libraries with known vulnerabilities, they find these are more likely to be transitive dependencies of other libraries (e.g. for ad tracking). They claim that as well as the web developers being at fault, *“the dynamic architecture and developers of third-party services are to blame for the Web’s poor state of library management”* (p. 1).

Neuhaus and Zimmermann (2009) demonstrate that new dependencies could both increase and decrease the risk of vulnerability, using Red Hat packages as an example. Observing the dependencies of a package, their model could predict vulnerability with a precision of 83%. Unlike other studies, Neuhaus’ model of dependencies is context-aware, in the sense that depending on some package has different effects based on the other dependencies already present.

Zhang et al. (2015) investigated the ‘attack surface exposed through package dependency’, and developed an algorithm to calculate a project’s attack surface for an individual vulnerability. Knowing that vulnerabilities in dependencies can affect the security of the base project, they show that the risk of such an attack can be measured, allowing developers to prioritise risks.

Decan, Mens and Constantinou (2018) ran a quantitative study of technical lag in the NPM dependency network; they found that the average technical lag (‘the time during which a dependency prevents the use of a newer version of its target package’) of all package releases is 7 to 9 months, which is ample time for vulnerabilities to be discovered and impact a downstream package.

Silic and Back (2016) investigated risk factors when adopting OSS[5] by interviewing IT professionals. They found source code integrity to be an important risk factor in the decision-making process (p. 173):

> *It is clear that the risk behind the use of OSS and the fact that by its nature, OSS source code can be accessed and modified by unknown individuals, directly influences the IT decision-maker.*

Considering this research, it's encouraging to find a useful tool for C-SCRM already contained within the `npm` CLI: `npm audit`. According to the documentation (npm Inc., 2021), the tool generates a report of known vulnerabilities in a project's dependencies, and offers the ability to automatically fix them. Security advisories in the NPM registry[6] are classified as either low, moderate, high, or critical, which can help a developer to understand the risk a vulnerability poses, and balance that against potential breakages when updating.

While this research shows that vulnerabilities can be predicted quite effectively, and that package dependencies introduce vulnerability in varying amounts, little work appears to have been done to apply these ideas within the software development process. Tools to predict vulnerabilities in packages are not widespread, and while `npm audit` can identify when a current dependency has a vulnerability, it cannot offer advice about the long-term risk of a dependency, nor offer any advice when choosing a new package to depend upon. My project aims to fill this gap, by producing an intuitive tool that helps software developers understand the risk of existing and new dependencies in their projects.

3.3 Visualisation

When developing the web interface, I wanted to find examples of services with similar goals of informing users of the risks associated with some entity, without deciding the entity is objectively good or bad.

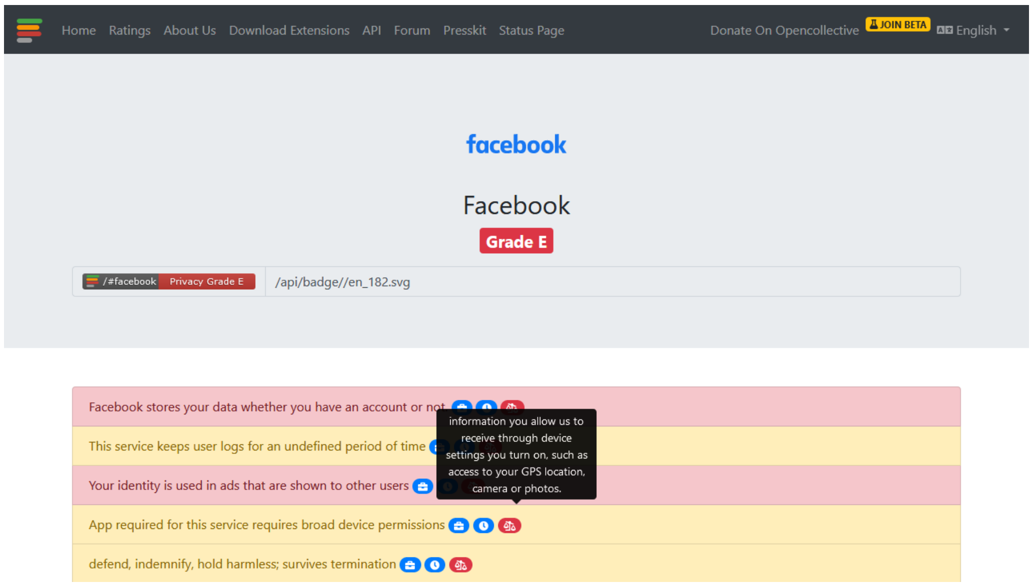
One such website is ToS;DR (Terms of Service; Didn't Read), a non-profit project started in 2012 that attempts to better-inform the public about the often worrying contents of Terms of Service documents (ToS;DR, 2012). For each service, they give a list of positive/negative features found in the Terms, and assign a qualitative grade (from A to E) to give users an understanding at a glance (see Figure 1).

Another website that performs a risk assessment of sorts is ReviewMeta, developed by Tommy Noonan in 2016 (ReviewMeta, 2016). This service analyses the reviews given to products on Amazon, runs various statistical tests to check their authenticity, then attempts to filter out 'unnatural' reviews from the total rating. Similarly to ToS;DR, the service offers an overall grade (this time Pass, Warn or Fail). Unlike ToS;DR, it relies on a multitude of tests that analyse

[5] A majority of package managers only serve open-source packages, so risks associated with OSS also apply to package dependencies.

[6] `https://www.npmjs.com/advisories`

[8] `https://tosdr.org/en/service/182`



facebook

Facebook

Grade E

/#facebook Privacy Grade E /api/badge//en_182.svg

Facebook stores your data whether you have an account or not

This service keeps user logs for an undefined period of time

Your identity is used in ads that are shown to other users

App required for this service requires broad device permissions

defend, indemnify, hold harmless; survives termination

information you allow us to receive through device settings you turn on, such as access to your GPS location, camera or photos.

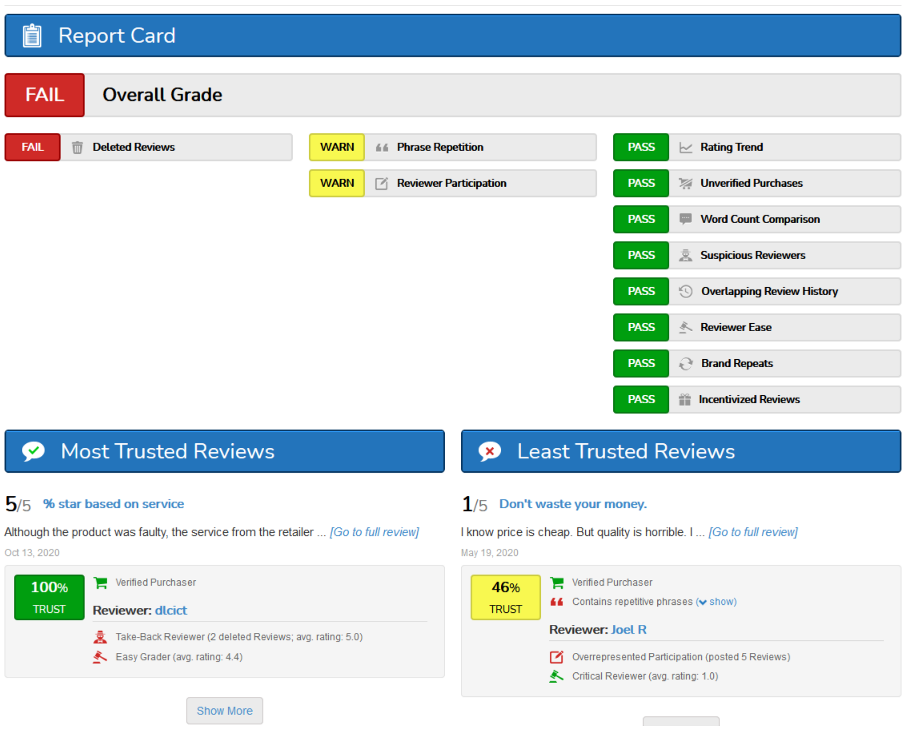
Figure 1: A screenshot of the ToS;DR website, showing a report for a specific service[8]. Note in particular the qualitative grade, the list of positive or negative factors, the tooltips with further explanation, and the extensive colour coding throughout.

the same reviews and metadata in different ways, and each get their own grade. This creates a hierarchy of information available on the report page - the overall grade, then the per-test grades, then the quantitative outputs from each test - allowing the user to read only the amount and type of information they're interested in (see Figure 2).

On first visiting ReviewMeta, a full-screen disclaimer is presented, making it clear to the user that the analysis *"is only an ESTIMATE"*, and the grades do *"NOT indicate presence or absence of 'fake' reviews."*. This is important not only to remove liability, but also for users to understand that the grades are subjective - though it seems his 'Pass/Fail' terminology slightly contradicts this intention.

In terms of visualisations specifically in the field of C-SCRM, I found Synk's Advisor interface, which also examines the NPM dependency network and presents graphs of various risk factors (see Figure 3). Advisor gives an overall score ('Package Health Score'), calculated from the scores in each category (Popularity, Maintenance, Security, Community), which in turn are calculated from empirical measurements of the repository and package registry. I only discovered this interface quite late into the project, so as a result I will wait until Section 9 to discuss its features and compare against my solution.

[10]`https://reviewmeta.com/amazon/B07W9DVHVS`


Report Card

FAIL Overall Grade

FAIL Deleted Reviews	WARN Phrase Repetition	PASS Rating Trend
	WARN Reviewer Participation	PASS Unverified Purchases
		PASS Word Count Comparison
		PASS Suspicious Reviewers
		PASS Overlapping Review History
		PASS Reviewer Ease
		PASS Brand Repeats
		PASS Incentivized Reviews

Most Trusted Reviews

5/5 % star based on service

Although the product was faulty, the service from the retailer ... [Go to full review]

Oct 13, 2020

100% TRUST

Verified Purchaser

Reviewer: dlcict

Take-Back Reviewer (2 deleted Reviews; avg. rating: 5.0)

Easy Grader (avg. rating: 4.4)

Show More

Least Trusted Reviews

1/5 Don't waste your money.

I know price is cheap. But quality is horrible. I ... [Go to full review]

May 19, 2020

46% TRUST

Verified Purchaser

Contains repetitive phrases (show)

Reviewer: Joel R

Overrepresented Participation (posted 5 Reviews)

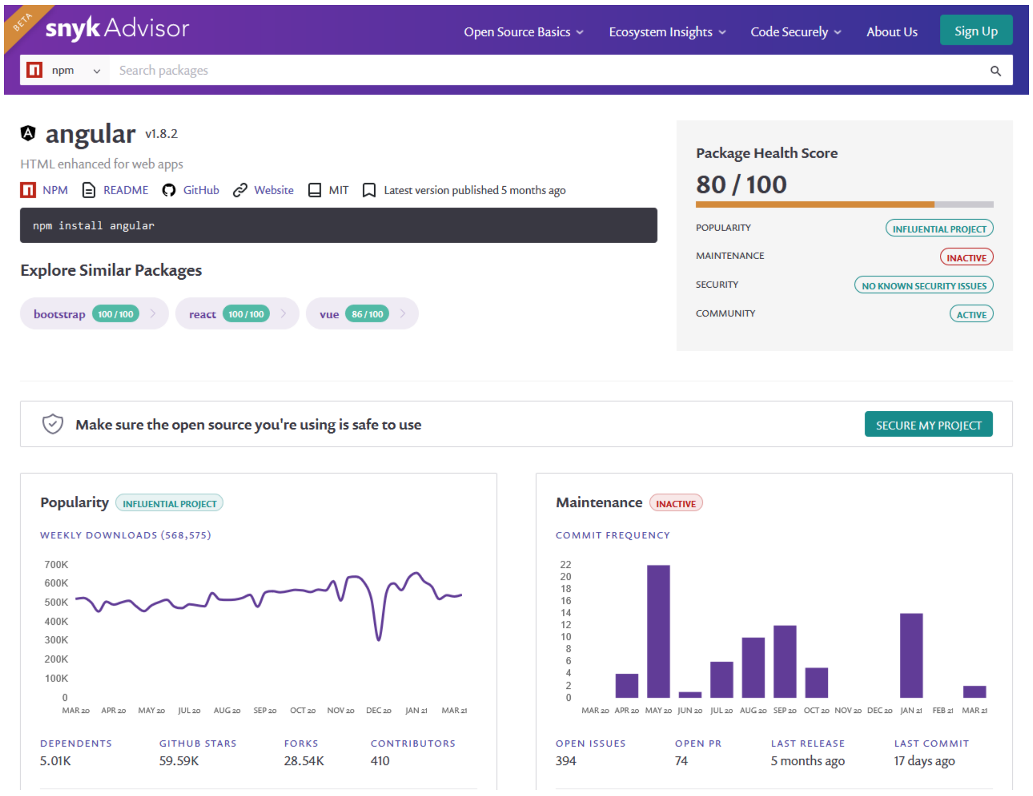
Critical Reviewer (avg. rating: 1.0)

Figure 2: A screenshot of the ReviewMeta website, showing a report for a product that failed the analysis[10]. Note in particular the pass/fail states of each test, the prominence of the overall grade, some examples of particularly good/bad reviews, and the colour coding throughout. More detailed information for each test is found lower on the report.

4 Methodology

This section describes the steps needed to produce the system, without giving specific technologies to use; this will come in Section 5. It's also written to be nonspecific to one package registry and version control server, so the technique can be applied to services other than NPM and GitHub (see Section 9.1). Figure 4 introduces the components and connections required to construct the system and produce the visualisations, and is further explained in the subsections.

[12]https://snyk.io/advisor/npm-package/angular



angular v1.8.2

HTML enhanced for web apps

NPM README GitHub Website MIT Latest version published 5 months ago

```
npm install angular
```

Explore Similar Packages

bootstrap 100 / 100 · react 100 / 100 · vue 86 / 100

Package Health Score

80 / 100

POPULARITY	INFLUENTIAL PROJECT
MAINTENANCE	INACTIVE
SECURITY	NO KNOWN SECURITY ISSUES
COMMUNITY	ACTIVE

Make sure the open source you're using is safe to use — SECURE MY PROJECT

Popularity — INFLUENTIAL PROJECT

WEEKLY DOWNLOADS (568,575)

DEPENDENTS	GITHUB STARS	FORKS	CONTRIBUTORS
5.01K	59.59K	28.54K	410

Maintenance — INACTIVE

COMMIT FREQUENCY

OPEN ISSUES	OPEN PR	LAST RELEASE	LAST COMMIT
394	74	5 months ago	17 days ago

700K
600K
500K
400K
300K
200K
100K
0
MAR 20 APR 20 MAY 20 JUL 20 AUG 20 SEP 20 OCT 20 NOV 20 DEC 20 JAN 21 MAR 21
22 20 18 16 14 12 10 8 6 4 2 0
MAR 20 APR 20 MAY 20 JUN 20 JUL 20 AUG 20 SEP 20 OCT 20 NOV 20 DEC 20 JAN 21 FEB 21 MAR 21

Figure 3: A screenshot of Synk Advisor, showing an analysis for the 'angular' package[12]. The top right box contains the Package Health Score with grades from each category. These are expanded upon in the boxes below, with line charts and histograms. The interface also offers 'Similar Packages', along with with their scores (top left).

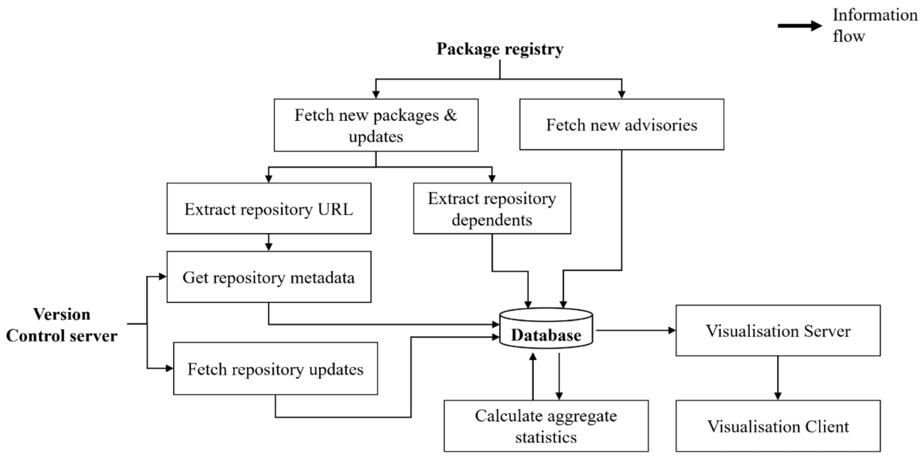


Figure 4: A flowchart of abstracted information flows between components of the system. The design is suitable for both a live service and a static dataset implementation. See Figure 5 for an implemented version.

(Note that it's possible for the package registry and version control server to be the same service, but I have not come across an example of this setup.)

4.1 Data acquisition

The system will need to have a record of all the packages available in a registry. This can either be a static snapshot of a registry at some timestamp, or a live copy that inserts new packages periodically or in real-time. In order to provide useful information to the end user, the system will also need properties of the packages' source code or version control repositories. Visualisations could be made out of any such properties of a package, but Table 1 describes some properties I identified as useful.

Entity	Property	Description	Required?
Package	`name`	The unique name of the repository	Yes
Package	`dependencies`	A list of names of packages this package depends on	Yes
Package	`repo_url`	A URL for a version control repository containing this package.	Yes, for repository metadata
Package	`date_updated`	A timestamp for when the package was last updated on the registry	No
Package / Repository	`date_created`	A timestamp for when the package/repository was first uploaded	No
Repository	`date_modified`	A timestamp for when the repository was last contributed to, indicating active development	No
Repository	`num_contributors`	The number of contributors with write-access to this repository	No
Repository	`total_contributions`	The lifetime number of commits to this repository	No
Package / Repository	`bytes_of_code`	The size of the source code files, measured in bytes	No
Contributor	`name`	The username of this contributor, to link contributions between repositories	No
Contributor	`contributions`	The number of contributions made to this repository, by this user	No
Advisory	`advisory_affects`	The identity of the package this advisory affects	Yes
Advisory	`date_published`	A timestamp for when the security advisory was published	No
Advisory	`severity`	A quantitative measure of the severity of the advisory. Values depend on registry convention	No

Table 1: A list of mandatory and optional properties of entities in the dataset.

Some of the rows above indicate the property can be found either in the package, or in its repository. While broadly similar, the values can differ on each[13], so deciding which to use will be based on ease of access and their usefulness in later processing. Furthermore, the `bytes_of_code` property could be swapped with lines of code (LoC/SLoC), depending on which is easier to acquire, as they can be converted based on a rough estimate of average characters per line of code.

4.2 Processing

After the sources of these properties have been identified, a database will be constructed to host the data. The entries will be relational, so a package can be easily linked to its contributors and security advisories. The most frequent operation on the database will be finding the tree of dependencies from a base package, so it should be suitably indexed to aid in selecting transitive dependencies. This requirement suggests a graph database would be well-suited, where data and relationships are stored as nodes and edges in a graph structure.

To accommodate low-latency requests for data, it would be worthwhile precomputing some results, and storing them as new properties, or in separate files. I didn't initially know which data would need to be precomputed, but these properties became useful to precompute during the implementation:

- **Number of dependent packages**: The number of packages that depends on a certain package is a useful property, particularly as a proxy for popularity. A package can have hundreds of thousands of dependents, with many hops between it and the dependents, making the calculation cover large portions of the dataset. Dependencies can also be cyclic, which for a badly-written query can cause a combinatorial explosion.

  Once these results are calculated, one should take care to keep them up to date when receiving updates to the dependency network; any packages upstream of a new package (or a new edge in the network) will need recalculation.

- **Percentile data of scoring categories**: An aim of the interface is to convey how one packages' scores relate to other packages in the dataset. It would be too difficult to calculate the score for every package at runtime, and it would be space-inefficient to store every score, so I compromise by sampling these scores at certain percentiles of the distribution. The number of samples to take depends on our desired resolution, but 100 samples is generally suitable.

  Some of these samples also required more precomputed results: for example, calculating the bytes of code in all the upstream packages (required for the *Dependency size ratio* score) is as difficult as identifying all the

[13]For example, a package might be in development a long time before it's released on the registry, so the `date_created` property would be earlier on the repository.

upstream packages. By precomputing the 'transitive bytes of code' and storing it as a property of each package, I can run a simpler percentile query on just those properties.

4.3 Visualisation

The visualisation serves two purposes: to examine a project's current supply chain risk, and to reveal how the risk would change if a new dependency were added. To this end, I'll create two pages: a single-project examination page (named internally as 'details'), and a comparison page for a new dependency ('compare'). The technology/medium of the visualisation is unimportant, but should support rich text and the presentation of various graphs in colour. As seen in Figure 4, a server is also needed to provide read-only access to the database through a small number of API endpoints.

The most prominent visualisation will be a network graph of the package's dependencies. Many visual variables are available to be controlled in a network graph, such as x/y positioning, node size and node colour. These can be assigned based on any acquired property (see Section 4.1), but some variables are better suited to particular properties:

- Vertical positioning should be controlled by the distance this dependency is from the base package (the minimum number of hops to the base). This indicates how directly the package is depended upon, and will produce a tree-like layout.
- Node size should be controlled by the package's source code size, to indicate which packages contain a significant amount of code and which are more lightweight.
- Many options are available for node colour, but unsafe packages (e.g. packages associated with security advisories) could be coloured in varying shades of red, so they stand out as risk factors.

On the comparison page, the network graph will change to indicate the separation between dependencies exclusive to the original project, dependencies exclusive to the new package, and dependencies shared between both.

As well as graphs, the output will contain a textual summary of the dependency tree, such as the number of transitive dependencies, the total size of the codebase, and the number of contributors with write-access to dependent packages. Similarly, it will output a shortlist of the top n contributors with the most *influence* over the dependency tree, where influence is calculated as

$$Influence(contributor) = \sum_{p \in T} \frac{p.commitsBy(contributor)}{p.totalCommits}.$$

As mentioned previously, the interface will present a series of scores that can be easily compared against scores of other packages. The particular scoring categories chosen will again depend on the properties acquired, but should be justifiable with regards to supply-chain risk.

Lastly, the visualisation will combine the scores from each category into an overall grade (e.g. A-F). To determine the goodness/badness of a score, we can check which percentile it belongs to (calculated earlier in Processing), then invert the percentile ($p \leftarrow 100 - p$) if lower is better. To combine into one grade, an average of these percentiles is taken (using mean, root-mean-square or another combining function), then convert to a letter.

5 Implementation

To implement the methodology described above, I return to using the NPM registry and GitHub for the data sources. I then choose technologies for the data storage and presentation, and identify scoring categories from the acquired and processed data. Figure 5 shows the system flowchart again, with concrete endpoints and components.

Following my previous suggestion of graph databases for efficiency, I used Neo4j, an open-source Graph DBMS (Database Management System), to store the data. A Neo4j database holds a graph of arbitrarily many *nodes*, each with an arbitrary number of associated *keys* (properties). Nodes can be linked together by *relationships* (edges) that can also have keys. The graph is accessed and modified using Cypher, a declarative graph query language with syntactic and semantic similarities to SQL.

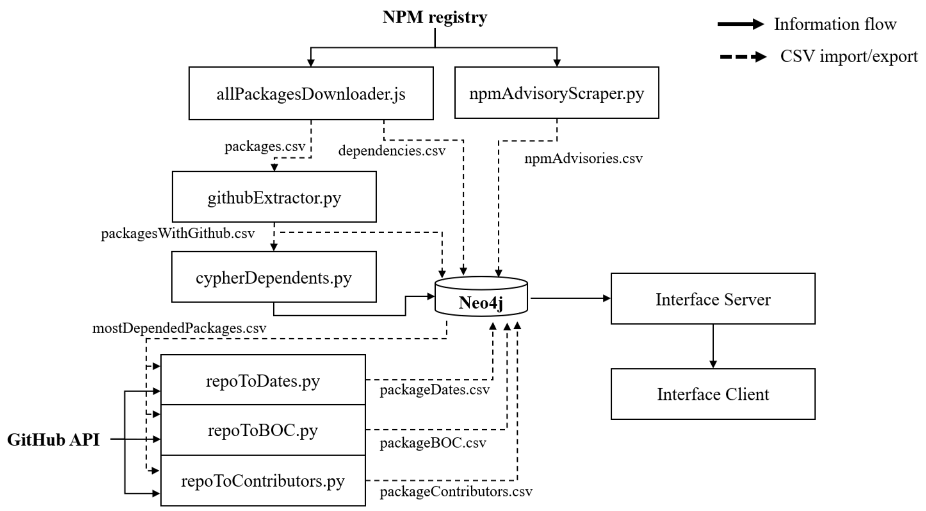


Figure 5: A flowchart of the various data stores and scripts used in my system, with a small key. Due to the experimental nature of the project, the system is quite disorganised, and relies on intermediary CSV files where direct communication would suffice. See Figure 15 for a more structured architecture suitable for a live service.

5.1 Data acquisition

Data acquisition and processing was divided into a series of tasks, described below. Note that these tasks were more interdependent than the separate lists suggest; for example, I needed to calculate the most popular packages before accessing the GitHub data.

- **Discovering all packages in the NPM registry**: Constructing the dependency network requires at least the name and direct dependencies of every package in the registry. In NPM packages, this data is contained in a `package.json` manifest file[14]. The `package-stream` package[15] provides a stream of the `package.json` files of all packages uploaded to the NPM registry. So I wrote a straightforward Node.js script to access this stream and write the data to CSV files.

  Node.js makes a distinction between dependencies a project relies on in production and dependencies that are only useful for development purposes, known as ‘devDependencies’. I made the script store both production dependencies and devDependencies, but when importing to Neo4j I chose to ignore the devDependencies (see Section 7 for a justification).

[14] I also collected the repository URLs at this stage, as this is also included in the manifest file.
[15] `https://github.com/nice-registry/package-stream`

- **Fetching package metadata**: Given a list of package names and repository URLs, we wish to find information about the size of a package, dates of creation and modification, and the contributors responsible for the package. Each of these features are accessible with different endpoints in the GitHub REST API (GitHub, 2021):
  - Dates: the '`/repos/{owner}/{repo}`' endpoint returns properties `created_at` and `pushed_at`, which are ISO timestamps for when the repository was created and last pushed to, respectively.
  - Package size: the '`/repos/{owner}/{repo}/languages`' endpoint gives a list of the programming languages used in each repository, along with their total size in bytes. After filtering non-programming languages such as HTML or CSS, these sizes are summed to create the 'bytes of code' property.
  - Contributors: the '`/repos/{owner}/{repo}/contributors`' endpoint gives a list of contributors and their total number of commits to this package, ordered by commit amount.

  Three Python scripts access each of these endpoints for each package, and write the results to CSV files for import into Neo4j (see below).

- **Fetching NPM security advisories**: I was not able to find an API endpoint that returned a list of security advisories in the registry, so I had to scrape the public advisory list[16] using Python and BeautifulSoup. The advisory name, their affected package, disclosure date and severity are written into a CSV file for Neo4j import.

5.2 Processing

The processing stage comprised of filtering, loading and computing statistics on the acquired data, and is again divided into tasks:

- **Filtering the package list**: in attempt to make the input data more consistent, I filtered the full package list to only packages containing a repository URL under the `github.com` domain.
  I chose GitHub because it was the most popular website for hosting repositories, but similar API endpoints probably exist for other services such as GitLab or BitBucket.

- **Loading CSV packages and dependencies into Neo4j**: I started by importing the `packages.csv` file, creating a '`Package`' node for each package. I then imported `dependencies.csv` by inserting a '`DEPENDS_ON`' relationship between packages:

```
:auto USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///.../dependencies.csv' AS row
MATCH (p1:Package {name: row.From})
```

[16]`https://npmjs.com/advisories`

```
MATCH (p2:Package {name: row.To})
CREATE (p1)-[rel:DEPENDS_ON]->(p2)
RETURN count(rel);
```

Listing 1: A Cypher query to load the dependencies CSV file into Neo4j.

Since there's over 3 million dependency relationships in the database, I had to use the '`USING PERIODIC COMMIT`' directive to commit the new relationships every n rows of the CSV, to avoid running out of memory.

- **Identifying most depended-upon packages** (as a proxy for popularity): To speed up the acquisition process, I wished to only download GitHub metadata for the most popular packages, but `package-stream` does not give any suitable statistics to indicate popularity. Since downloading a package is equivalent to making it a dependency, I argue that the number of dependents in the dataset is a suitable proxy for download counts, and thus a package's popularity. So this query was made to find all packages depending on a given Package and store it as a property of the Package:

```
MATCH (p:Package{name:'package_name'})
MATCH (p)<-[:DEPENDS_ON*..10]-(p2:Package)
WITH p, count(distinct(p2)) as c
SET p.dependents = c;
```

Listing 2: A Cypher query to calculate the number of packages depending on the package '`package_name`'.

In an earlier version of this query, the transitive closure of `DEPENDS_ON` was not capped at 10 hops and was running out of memory before terminating. Since the number of dependents was mostly stable when experimenting with different `maxHops` values, I suspect the evaluator was stuck in a cycle of dependencies.
Since I was also having out-of-memory issues when running this query on all packages at once, I used a Python script to run the query on each individual package. In retrospect, it seems that correct usage of the `UNWIND` clause (i.e. UNWINDing on the list of package names) would fix the memory issue and improve efficiency.

Using these counts, I selected the top 5,000 packages and exported them to a CSV file:

```
MATCH (p:Package)
RETURN p.name, p.repo_url
ORDER BY p.dependents DESC
LIMIT 5000;
```

Listing 3: A Cypher query to fetch the 5,000 most depended-upon packages in the database.

- **Calculating percentile data for scoring categories**: To be able to draw histograms of the scoring categories without downloading the whole dataset to the client, I precalculated histogram data for each property to send instead. I used a Cypher query to return the value of the property at percentiles 0-100 and used these arrays to construct the histograms.

```
MATCH (p1:Package) where exists (p1.bytes_of_code)
UNWIND p1 as x
MATCH p = (x)-[:DEPENDS_ON*1..9]->(p3:Package)
WITH
  x, SUM(distinct p3.bytes_of_code) as transBoC
SET x.TRANSITIVE_BYTES_OF_CODE = transBoC;

MATCH (p:Package)
WHERE exists (p.TRANSITIVE_BYTES_OF_CODE)
WITH
  p.name as name,
  sum(p.TRANSITIVE_BYTES_OF_CODE)/toFloat(p.bytes_of_code) as
     code_size_ratio
UNWIND range(0,100) as x
RETURN x as percentile, percentileCont(code_size_ratio, toFloat(x)/100)
      as value;
```

Listing 4: A Cypher query that returns 100 percentiles of the ratio between the size of the package and its dependencies. Because of double counting errors, I needed to write to a temporary property ('TRANSITIVE_BYTES_OF_CODE') before UNWINDing.

> I exported the query result as a CSV and loaded it into the interface JavaScript manually. In a live service, these queries should be intermittently executed on the up-to-date graph, and the results loaded into the interface through an API endpoint.

5.3 Visualisation

As mentioned earlier, my implementation uses a server/client model for the interface. The visualisation is constructed in a web application, using JavaScript for logic and HTML/CSS for rendering. The application consists of a single HTML document which sections are made visible depending on the URL accessed: '`/`' for the index page, '`/details`' for the single-project visualisation and '`/compare`' for the new dependency visualisation.

The web application and API endpoints are served by a Node.js server. URLs starting with '`/api`' are passed to an API handler that responds with JSON objects, and for all other URLs the webapp HTML is served.

To draw the network graphs and histograms, I used D3.js (a.k.a. D3), an open-source data-driven graphing library. It provides useful interfaces to draw graphs into <`svg`> elements, along with an implementation of particle force simulations to naturally spread-out nodes of a network graph, which alleviates some layout work.

For the styling of other elements on the interface, I used the Bootstrap CSS framework, which defines styling rules for various HTML elements. I added additional styling for elements such as the grade badges and D3 graph elements. As recommended by Bootstrap, I used the Popper JavaScript library to show tooltips with more information when `(i)` symbols are hovered over.

For the scoring sections, I defined 6 scoring categories based on the properties I had available, which are listed in Table 2.

Category Name	Description (tooltip)	Higher is better?	Combining function
Dependency count	The number of transitive dependencies this project relies on	No	Count
Contributors per package	The average number of contributors listed against each package	Yes	Average
Dependency size ratio	Dependency kLoC ÷ your package's kLoC	No	Sum ÷ Base
Historical Vulnerabilities	Sum of past vulnerabilities, weighted by severity	No	Sum
Commits per day	Number of commits to each package ÷ its age	Yes	Average
Dependency age	Mean number of years since each package was created	Yes	Average

Table 2: A list of the scoring categories I included on the interface, along with the combining function used to aggregate that score. The *Higher is better?* attribute is used when combining the scores into the overall grade.

To access a visualisation, the index page provides a HTML form to direct to both pages. A user inputs either the name of a package already on the NPM registry, or uploads their projects' `package.json` file[17]; for the comparison page, they must also provide the name of a new package to depend upon. Submitting a form redirects the browser to a URL with the package names written in GET variables, and the correct visualisation is presented.

5.3.1 Command-line interface (CLI)

In addition to accessing reports via the index page, I developed a small command-line interface in Node.js that, once installed globally[18], is able to present a report for the Node.js project in the current working directory, or a comparison report between this project and a hypothetical new dependency.

```
> npm-scrm --help
Usage: npm-scrm [details | compare <package_name>]
```

Listing 5: The '`--help`' output of the `npm-scrm` CLI.

Running the program with valid inputs reads this directory's `package.json` file, then opens the default web browser with a URL containing the correct package names. This is evidently a reuse of the web interface rather than true implementation of a CLI that produces static HTML reports. See Section 8 for a further discussion of CLIs and static HTML reports.

[17]The JavaScript FileReader API can be used to read the `package.json` files in the client, then redirect to the correct visualisation URL. My implementation instead uploads the file to the server, which returns a visualisation based on the dependencies in the file. FileReader would probably be more efficient, as well as more secure (see Section 8).

[18]using '`npm install -g npm-scrm`'

6 Results

This section will show the output of each stage of the implementation described above. Discussion of the results is reserved for Section 7.

6.1 Data acquisition

Acquiring the data took about a week, with most of this time spent fetching package details through `package-stream`. In total, this script downloaded and stored details of 1,469,824 packages[19], and identified 4,548,778 direct dependency links between packages. Filtering the list to only packages with GitHub repository URLs this down to 931,225 packages, and 3,007,109 direct dependencies between them.

After selecting the top 5,000 most depended-upon packages, the repository scripts downloaded metadata for the dates, codebase sizes and contributors, which took a few hours to complete on 16 Nov 2020. The scripts identified 19,738 unique contributors appearing in the top 5,000 repositories, and 75,756 links between a contributor and a repository. So on average, a contributor appearing in the dataset has contributed to 3.84 packages in the top 5,000 NPM repositories.

Scraping the NPM advisory page was faster than the previous steps, due to there only being 1,310 advisories on record when the script was executed on 8 Dec 2020.

6.2 Processing

In this section I'll present some statistics of the acquired data; these statistics are not visible to the end user of the interface, but some of them appear in the Summary and the Scoring sections of the interface (see Section 6.3). To begin, I calculated the transitive dependents[20] count of each package, shown in Figure 6.

[19] i.e. all of the packages in the NPM registry as of November 2020.

[20] a *transitive dependent* of package A is a package that relies on A, through transitive closure of the direct dependencies - i.e. any downstream package in the dependency network.

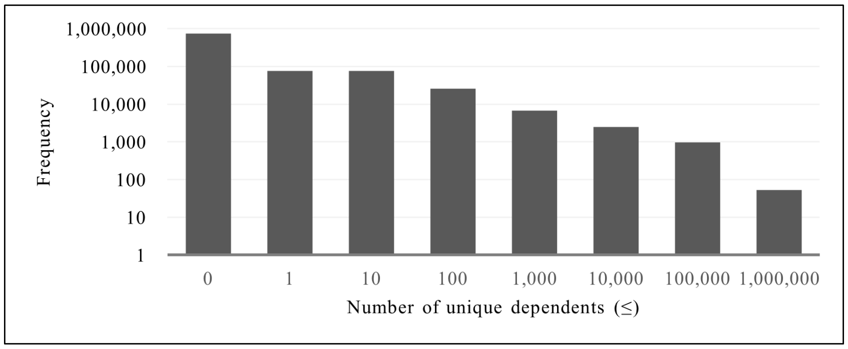


Figure 6: A histogram of the 'transitive dependents' count of each package in the dataset. The bucket labels indicate the upper bound of that bucket.

The histogram shows that 80% of the packages in the dataset have no dependents at all, and 96% of packages have at most 10 dependent packages[21].

In the other direction, a package has on average 5.0 direct dependencies, and 12.4 transitive dependencies - this distribution is shown in Figure 7.

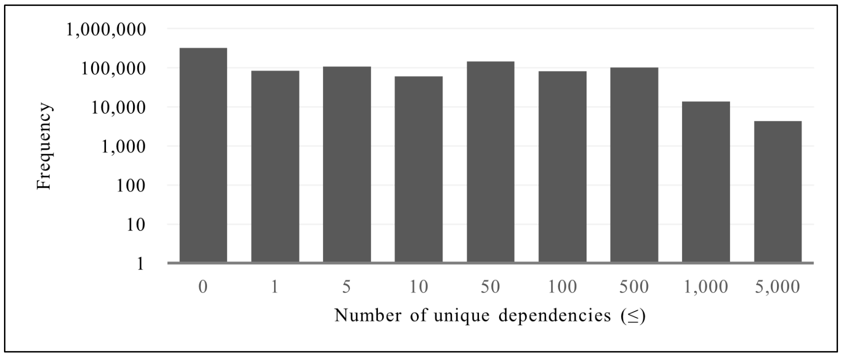


Figure 7: A histogram of the 'transitive dependencies' count of each package in the dataset. Again, the bucket labels indicate the upper bound of that bucket.

The dependencies histogram shows a surprising range of development practises in the NPM community. 35% of packages in the dataset ($\approx 328,000$ packages) have no dependencies at all, and 56% ($\approx 521,000$) have at most 5 transitive dependencies; on the other hand, 13% ($\approx 120,000$) have over 100 transitive dependencies. This distribution is used as one of the scoring categories ('Dependency count'), shown in the next section.

Precomputing the percentile data for the scoring categories also reveals interesting statistics about the distribution of values, shown in Table 3.

[21]Note that this data considers only public packages in the NPM registry - it's possible that many closed-source projects depend on some package that few packages in the registry do.

Category Name	Mean	Median	Standard Deviation
Dependency count	78.5	1	109
Contributors per package	15.2	11	11.9
Dependency size ratio	219	1.0	1240
Historical Vulnerabilities	0.0546	0	6.50
Commits per day	0.898	0.0616	2.39
Dependency age	5.89 years	6.10 years	2.24 years

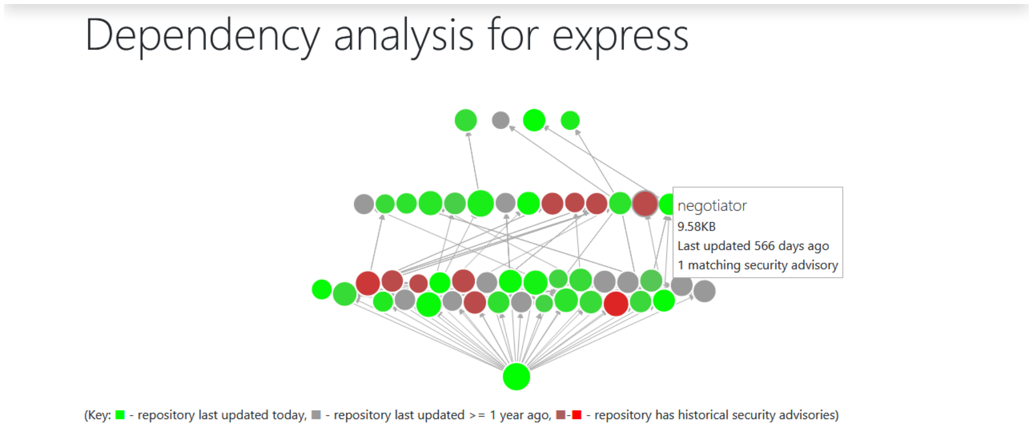
Table 3: A table showing the mean, median and standard deviation for each scoring category, to 3 significant figures.

6.3 Visualisation

As mentioned previously, the visualisation is split between '`/details`' and '`/compare`' pages, though the overall components are quite similar and described in the sections below.

6.3.1 Dependency graphs

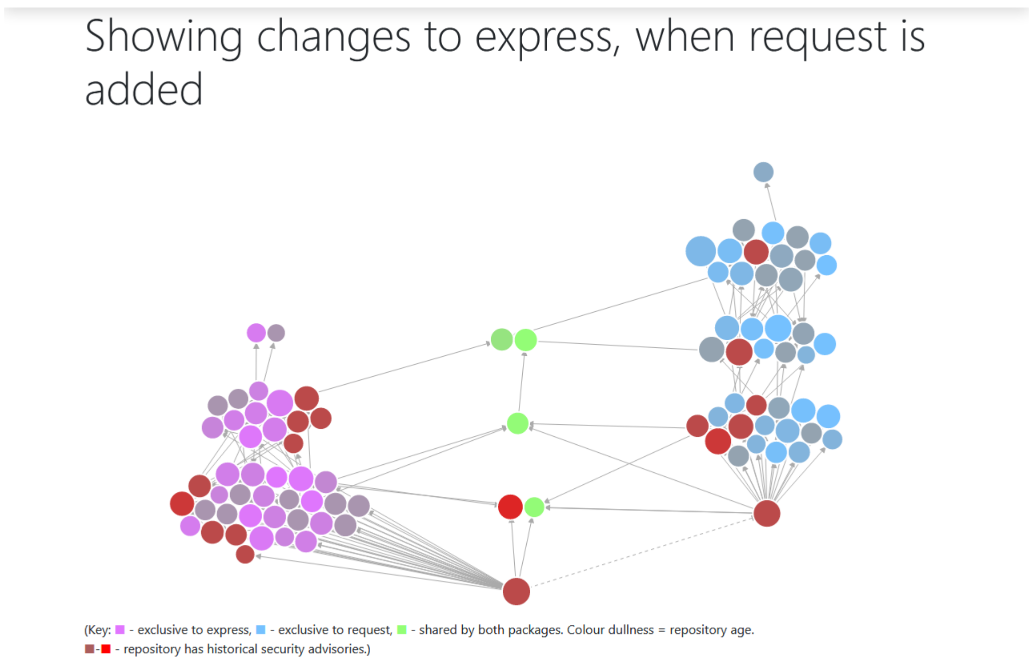
Both pages contain a network graph of the project's dependencies, where packages are organised in a tree shape (with y-position determined by dependency hops), scaled based on their source code size, and coloured based on their package age and historical advisories (See Figures 8 & 9). Each package has a tooltip containing these variables in textual form, appearing when the node is hovered over.


Dependency analysis for express

negotiator
9.58KB
Last updated 566 days ago
1 matching security advisory

(Key: 🟩 - repository last updated today, ⬜ - repository last updated >= 1 year ago, 🟥-🟥 - repository has historical security advisories)

Figure 8: A screenshot of the dependency graph presented on the single-project page, with a colour key. A tooltip is shown for package '`negotiator`'.

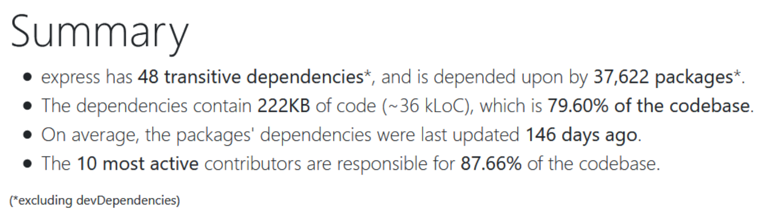

Showing changes to express, when request is added

(Key: 🟪 - exclusive to express, 🟦 - exclusive to request, 🟩 - shared by both packages. Colour dullness = repository age.
🟥-🟥 - repository has historical security advisories.)

Figure 9: A screenshot of the dependency graph presented on the comparison page, with a colour key. The colouring differs from the first graph to incorporate the ‘existing/shared/new dependency’ information, but all previous features remain.

6.3.2 Summary points

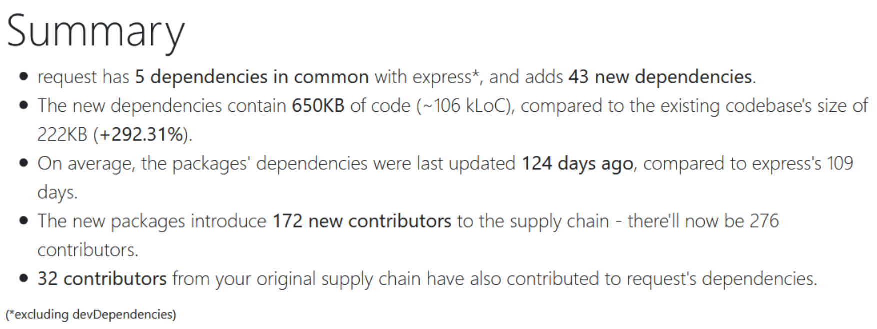
Both pages give a textual summary of the main properties of the dependency tree, with the compare page containing slightly more information regarding the changes to these properties (See Figures 10 & 11).


Summary

- express has **48 transitive dependencies***, and is depended upon by **37,622 packages***.
- The dependencies contain **222KB** of code (~36 kLoC), which is **79.60% of the codebase**.
- On average, the packages' dependencies were last updated **146 days ago**.
- The **10 most active** contributors are responsible for **87.66%** of the codebase.

(*excluding devDependencies)

Figure 10: A screenshot of the Summary section on the single-project page for ‘`express`’.



Summary

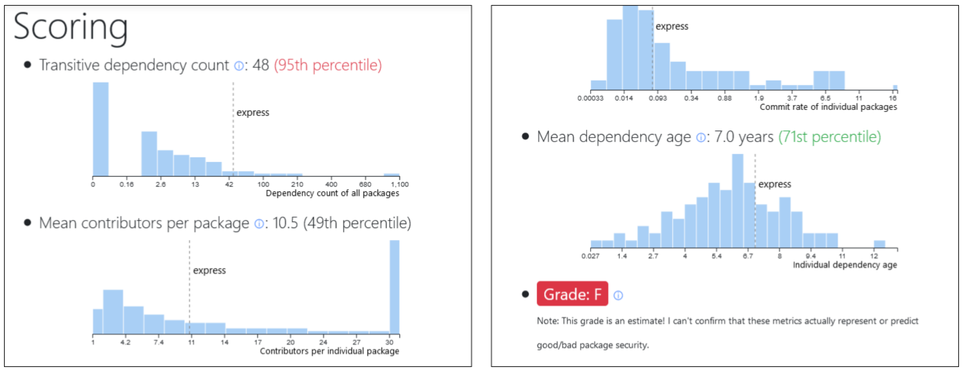
- request has **5 dependencies in common** with express*, and adds **43 new dependencies**.
- The new dependencies contain **650KB** of code (~106 kLoC), compared to the existing codebase's size of 222KB (**+292.31%**).
- On average, the packages' dependencies were last updated **124 days ago**, compared to express's 109 days.
- The new packages introduce **172 new contributors** to the supply chain - there'll now be 276 contributors.
- **32 contributors** from your original supply chain have also contributed to request's dependencies.

(*excluding devDependencies)

Figure 11: A screenshot of the Summary section on the comparison page between '`express`' and '`request`'.

6.3.3 Scoring categories & grade

Both pages give a series of score values that contribute to the overall grade, with histograms plotted for each category, and markers indicating this package's position in the distribution (See Figures 12 & 13). On the details page, a colour-coded percentile is given next to the score value, but on the compare page a score change is displayed instead.



Scoring

- Transitive dependency count ⓘ: 48 (95th percentile)

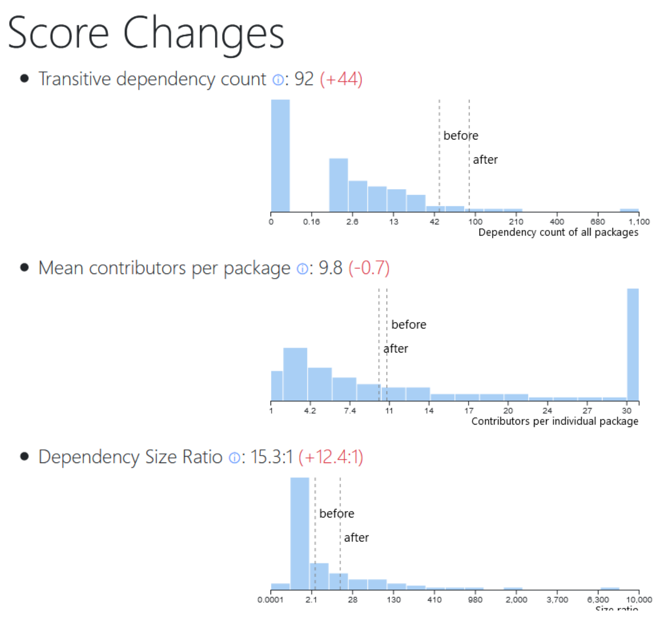
- Mean contributors per package ⓘ: 10.5 (49th percentile)

- Mean dependency age ⓘ: 7.0 years (71st percentile)

- Grade: F ⓘ

Note: This grade is an estimate! I can't confirm that these metrics actually represent or predict good/bad package security.

Figure 12: A screenshot (wrapped) of the Scoring section on the single-project page. The overall grade badge is found below the scores, along with a disclaimer.

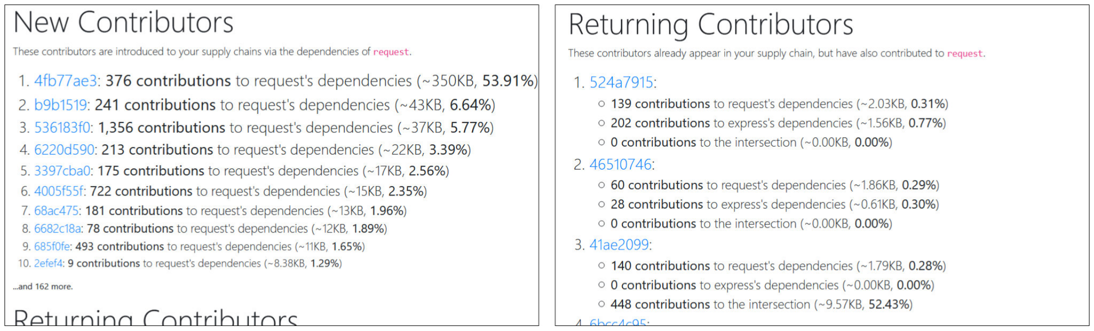

Score Changes

- Transitive dependency count ⓘ: 92 (+44)
- Mean contributors per package ⓘ: 9.8 (-0.7)
- Dependency Size Ratio ⓘ: 15.3:1 (+12.4:1)

Figure 13: A screenshot of the Score Changes section on the comparison page. Two markers are inserted on the histogram to visualise the change.

6.3.4 Contributor statistics

Both pages have a list of the contributors with the highest influence over the dependencies (See Figure 14). In this figure, the username have been scrambled for anonymisation, and the GitHub profile hyperlinks have been disabled.


New Contributors

These contributors are introduced to your supply chains via the dependencies of request.

1. 4fb77ae3: **376 contributions** to request's dependencies (~350KB, **53.91%**)
2. b9b1519: **241 contributions** to request's dependencies (~43KB, **6.64%**)
3. 536183f0: **1,356 contributions** to request's dependencies (~37KB, **5.77%**)
4. 6220d590: **213 contributions** to request's dependencies (~22KB, **3.39%**)
5. 3397cba0: **175 contributions** to request's dependencies (~17KB, **2.56%**)
6. 4005f55f: **722 contributions** to request's dependencies (~15KB, **2.35%**)
7. 68ac475: **181 contributions** to request's dependencies (~13KB, **1.96%**)
8. 6682c18a: **78 contributions** to request's dependencies (~12KB, **1.89%**)
9. 685f0fe: **493 contributions** to request's dependencies (~11KB, **1.65%**)
10. 2efef4: **9 contributions** to request's dependencies (~8.38KB, **1.29%**)

...and 162 more.

Returning Contributors

Returning Contributors

These contributors already appear in your supply chain, but have also contributed to request.

1. 524a7915:
   - **139 contributions** to request's dependencies (~2.03KB, **0.31%**)
   - **202 contributions** to express's dependencies (~1.56KB, **0.77%**)
   - **0 contributions** to the intersection (~0.00KB, **0.00%**)
2. 46510746:
   - **60 contributions** to request's dependencies (~1.86KB, **0.29%**)
   - **28 contributions** to express's dependencies (~0.61KB, **0.30%**)
   - **0 contributions** to the intersection (~0.00KB, **0.00%**)
3. 41ae2099:
   - **140 contributions** to request's dependencies (~1.79KB, **0.28%**)
   - **0 contributions** to express's dependencies (~0.00KB, **0.00%**)
   - **448 contributions** to the intersection (~9.57KB, **52.43%**)
4. 6bcc4c95:

Figure 14: A screenshot (wrapped) of the 'New/Returning Contributors' sections of the comparison page, with scrambled usernames.

7 Discussion

This section will contain an analysis of the results found at each stage, and a brief justification of the impact of the results. A discussion of limitations relating to the methodology or implementation will be deferred to Section 8.

7.1 Data acquisition

To recap, the results from this stage consisted of counts of entries acquired of each type (packages, contributors, advisories etc.).

A consequence of downloading a snapshot of a live dataset is that the data quickly becomes outdated. Between November 2020 (when the data was downloaded) and March 2021 (the time of writing), $\sim 84,000$ new packages have been published, and version updates to existing packages may have greatly changed the graph landscape, which would affect the scores/grades for those packages. The security advisories are also out of date, meaning malicious packages could still be considered safe in this visualisation.

Another decision mentioned in Section 5 was to ignore devDependencies (dependencies only used as development tools). This was done to reduce the size of the database and to remove some confusion in the visualisation (how should the difference between dependencies/devDependencies influence the graphs/scores?), but also because it's unclear how much devDependencies have a security impact on the final product. Synk operate a C-SCRM service for NPM that ignores devDependencies by default, arguing they 'matter little when searching for vulnerabilities' (Podjarny, 2016). On the other hand, it seems that both the `left-pad` availability incident and the `event-stream` injection attack (see Section 2.2) would still have impacted the project if these packages were included as devDependencies.

Other issues arise from the extraction of repository metadata. For example, the script that fetches the size of code in a repository might be counting non-program text as code[22], and cannot determine which code is/isn't bundled in the built package, both of which could inflate the computed size of a package. Additionally, the histogram for contributors shows a large spike at 30 and no values after that, which revealed an oversight in my acquisition script (`repoToContributors.py`), which was failing to make subsequent requests for packages with more contributors than would fit in one response. I fixed the script after identifying this error but decided not to re-run the analysis pipeline, as it would only have a small impact on the results. Contributor identities returned on this endpoint are sorted by contribution count (descending), so the missing contributors had little influence on the packages relative to the included ones.

On the topic of missing data, the proportion of packages lost when filtering by

[22] I excluded HTML and CSS files from the counting after briefly checking the script output, but other non-program file formats are likely to persist.

repository URLs containing `github.com` (37%) was higher than expected. But examining the original list of packages shows that 30% of the packages actually had no repository URL at all (or did, but were not following the standard format of `package.json` files). This leaves 7% of repositories being hosted on other services, such as GitLab and BitBucket - but these services only represent 18% and 12% of the remaining packages, respectively. Therefore, it seems infeasible to write handlers for all of the remaining repository locations. It's possible that some of the repository metadata could be acquired directly from the NPM registry (or through proxy of other NPM metadata), which would allow the rest of the packages to be included in the dataset.

7.2 Processing

Calculating the distribution of properties gave revealing insights into the state of the NPM registry, but also revealed some incorrect expectations. With 80% of the packages having no dependents in the registry, it suggests that many packages are incomplete, redundant, or stale - meaning they would realistically never be chosen by software developers. It's still useful to include them in the dataset (so the visualisation can mark them as security risks if necessary), but it has the effect of heavily skewing the scoring category percentiles.

For example, *Historical vulnerabilities* is heavily skewed towards 0, which is arguably because a large number of these packages have never been popular enough to warrant a security review. The average number of dependents over all the packages is 251, but the average number of dependents for packages with at least one associated security advisory is 6,493 ($\sim$ 25x larger). Interestingly, 52% of packages with associated advisories (578/1,121) also have 0 dependents, likely because the packages were later replaced with a 'security holding package' which contains no code[23]. This suggests that 'dependent count' is not an accurate measure of package popularity as these packages are still regularly downloaded, so acquiring the actual download statistics would be more reliable. It would also reveal the package's usage outside of the NPM registry graph, which the system does not currently consider.

7.3 Visualisation

In general, I believe the visualisations and the web application work as intended, and are effective in their task of conveying information from the dataset. The page loads in good time[24] and has good performance[25]. I would be eager to

[23]See for example `https://www.npmjs.com/package/angluar-cli`; 'angluar-cli' is a mistype of popular package 'angular-cli' containing malware (`https://www.npmjs.com/advisories/918`).

[24]The majority of time is spent downloading scripts and data, which could be improved by enabling HTTP/2 and gzip compression.

[25]Aside from the edge case of packages with $\sim 1,000$ dependencies, for which D3.js struggles running the force simulation on so many nodes. The resulting graph is also too wide to fit on a standard laptop screen, so perhaps a condition to omit some nodes when the count gets too high would be useful.

conduct a survey of developers' opinions about this system and incorporate their feedback, but that is outside the scope of this project.

One aspect with a few areas for improvement are the histograms under each scoring category. Firstly, the power scaling (i.e. quadratic and quartic) on the x-axis creates sub-integer buckets for integer-valued domains, so the *Transitive dependency count* histogram has a visible gap between 0 and 1. Some bars are drawn wider than others for no obvious reason; this seems to be a quirk of D3 that can probably be overriden. Additionally, strange tick values are generated for the x-axis because I had to reimplement the generator function (D3 usually generates the tick values before applying the scaling, resulting in non-uniform ticks). Since the main intent of these histograms is to show the general shape of the distribution and this project's position in that distribution, these issues are forgivable. In fact, the desired information could be conveyed in a graph without any labels or scaling at all, akin to how sparklines are commonly used in financial visualisations (Francy, 2005).

The scoring categories seem to be effective at differentiating the packages, as it's easy to come across packages with scores on the low end / high end / middle of the distribution for a certain category. However, my selection of scoring categories, and whether 'higher is better' for each, is not sufficiently justified, on the interface nor in this paper. Indeed, for a category such as *Commits per day*, it's not immediately clear which direction is correct: a low score could suggest inactive development or a slow reaction time to bug reports, but a high score could suggest a volatile development cycle in which breakages are frequent. It's also an unintuitive unit, meaning an end user can only really hope their score is near the middle of the distribution.

There are also categories I could have included using my existing data that may have been more informative than the current ones, such as the total number of dependents of all transitive dependencies of a project (the idea being that more dependents implies more scrutiny). The selection of scoring categories is clearly a complicated one, and warrants much deeper research into vulnerability prediction from package metadata.

Lastly, while the overall grade is a useful feature in theory, I noticed it was reporting low grades (F/E) much more commonly than high grades. Upon observation, it appeared that scoring well in some categories correlated with scoring badly in other categories: one example being an often-negative correlation between *Dependency age* and *Commits per day.* Since users have the option to disregard categories they're not interested in, the correlation doesn't seem to be an issue. But perhaps one way to boost the grades would be to use a root-mean-square average of the score percentiles rather than a simple mean, so that higher scores are more distinguished and lower scores have less impact on the grade.

8 Reflection

Now that the capabilities of the visualisation system have been demonstrated, it's worth reconsidering the value of C-SCRM in open-source software development. While software bugs and vulnerabilities are likely to never disappear from an evolving codebase (Pressman, 2005), availability risks such as unpublishing of packages can be addressed - and in NPM's case, have been fixed. Packages can no longer be unpublished after 24 hours of being published (Brown, 2017), so it seems unlikely for another instance of the `left-pad` incident to occur.

Returning to Pashchenko, Vu and Massacci (2020)'s study, and Observation 13 (repeated below):

> ***Observation 13:*** *Developers recommend introducing high-level metrics that show that a library is safe to use (security badge), mature, and does not bring too many transitive dependencies.*

I would argue my visualisation achieves many of the features requested by the developers: it has a high-level security metric presented in a badge (the Grade) and has measures for library maturity and the number of transitive dependencies, so it seems that the system would be of use to developers. Observation 16 is also worth discussing (p. 1521):

> ***Observation 16:*** *Developers recommend dependency analysis tools to report only relevant alerts, work offline, be easily integrated into company workflow, and report both recent and early safe versions of vulnerable dependencies.*

I would say my visualisation fails at this specification. The current architecture does not work offline, doesn't integrate well into an existing workflow (see later discussions regarding a CLI), and doesn't take package versions into account. Making the dependency network version-aware would be quite difficult: I would need to acquire the version history of every `package.json` file, then have dependency edges associated with a range of version numbers. The additional computation required to compare version numbers might also have a significant performance impact on the database queries.

Another significant limitation of my implementation was the decision to use a static dataset rather than a live service that observes and acts upon package/advisory updates. A static visualisation still has value to developers; it successfully shows trends in the dependencies of each package and can give an estimate for the security of a new dependency. But this data will quickly go out of date and become less reflective of the true nature of the packages. I didn't implement a live service as it would require overcoming too many technical hurdles outside the scope of this project, but I would recommend doing this for a production version. Figure 15 shows a suitable design for a live service, along with other improvements.

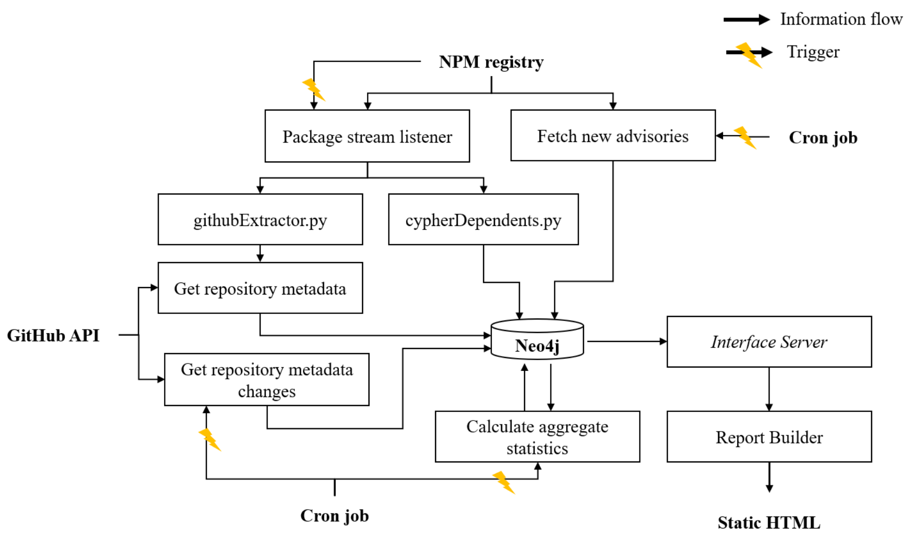


Figure 15: A flowchart of data acquisition & processing stages, with various structural improvements and adjusted to support a live service. It's unclear whether services such as GitHub offer hooks/messages to notify a change, so a cron job trigger is used for periodic invocations of the script.

8.1 Data acquisition

The dataset in this implementation is intended to be a snapshot of the state of the packages at a certain date. But some scripts took days to complete, and other scripts were executed some time apart from each other, so the initial data could be out of date even by the time the acquisition is finished. This would be fixed in the live-service design, but the extent of this issue in the static design is yet to be realised.

When writing the acquisition scripts, I designed most of them to output the data to CSV files, which could be imported to Neo4j or used in subsequent scripts. This was useful for understanding the data and debugging script failures, but resulted in a lot of manual work when importing/exporting to/from Neo4j. Having each script communicate directly with Neo4j would increase automation and efficiency of the system.

As mentioned in the Discussion, filtering the packages by repository URL was done too early in the pipeline; filtering should have been done after importing the packages into Neo4j, so as to not skew the histogram data (see Section 6.2). But a more significant filter was choosing only the top 5,000 most depended-upon packages to acquire repository metadata for. This step was necessary as the GitHub API was heavily rate-limited; acquiring data for more packages would delay my project by a large amount. It might be worth exploring ways to expand the 5,000 limit without sending too many requests to the GitHub API. I imagine that in a live service, the rate of package updates would be considerably lower than the GitHub API rate limit, so the challenge would only apply to acquiring the initial set of data.

8.2 Processing

On the topic of a live service, it's worth noting that many of the pre-processing steps in my methodology will become more complex. When a package adds a new dependency in a version update, the dependents/dependencies counts for every upstream/downstream package will need to be updated, along with any other stored aggregate statistics. As long as the number of transitive dependencies stays much smaller than the size of the whole dataset, this should be cheap enough to execute in real-time.

One thing I realised when testing the interface is that it's possible, and normal, for multiple NPM packages to be developed in (and therefore, point to) the same version control repository, meaning the bytes and contributions in these repositories are double counted. As a particularly bad example, the `@types/...` collection of packages is operated by a bot which stores all 6,500+ packages in the same repository[26]. It's unlikely to see more than a couple of these packages in a project's dependencies, but the database counts contributions to one package as contributions to all the others, resulting in bloated package sizes. It's unclear how this could be rectified (perhaps uniformly-distribute the contributions between packages sharing repositories?), but it's worth bearing in mind when considering other package managers to visualise (see Section 9.1).

Choosing Neo4j for database management seems to be the right decision. The Cypher query language took some getting used to, and its relative infancy compared to SQL makes online documentation more difficult to come by, but having a direct correspondence between the abstract operation and the actual graph layout of the data makes the resulting queries fairly intuitive. Unfortunately, Cypher queries that accessed/modified the whole graph would often fail from running out of memory on my machine. I fixed this by instead using a script to update one node at a time, committing the result in between each query.

8.3 Visualisation

I'm happy with the presentation and contextualisation of data on the interface, though as mentioned the feature selection is lacking in justification. The scoring categories are plausible but unsubstantiated, and calculating the overall grade as an average of each score's 'goodness' is especially subjective. I believe there's a trade-off between academic, evidence-backed correctness and usefulness to developers. To satisfy both sides, I try to present quantitative values where possible, but contextualised with simplified statistics like the grade and the

[26]`https://github.com/DefinitelyTyped/DefinitelyTyped`

unlabelled histograms. This is reminiscent of Shneiderman (1997)'s mantra (*"Overview first, zoom and filter, then details-on-demand"*): the grade would represent an overview, and the individual statistics and tooltips would be details-on-demand. The mantra doesn't apply so well to the application's UI; the high-level statistics are buried further down the page than they should be (compared to ReviewMeta and ToS;DR, which both present their grade at the top of the page). It would be easy and worthwhile to reorganise the HTML output to improve 'at-a-glance' readability.

Making a web application was decided for easy implementation, but for a product that better aligns with developers' workflows, it would be preferable to generate static HTML report files from a CLI, similarly to the `powercfg /batteryreport` tool on Microsoft Windows. A CLI already exists in my implementation, but making it the main entry point would be advantageous for a number of reasons: a majority of software development tools are CLIs (Unwin and Hofmann, 1999), so it would fit in well with existing workflows; developers might be working on virtual machines without a web browser and need to copy the report elsewhere; and it would open the opportunity to export reports in alternative, machine-readable formats such as JSON files. The client-server model could still apply, but calculating the scores and summary points could be done in the CLI instead of in the web browser[27].

Regardless of using a web application or outputting static pages, the client-server model poses a security risk when requesting the dependency data from the server. Disclosing the dependencies of a closed-source project reveals a large attack surface and may even reveal current vulnerabilities in the project. The `npm audit` tool uses 'non-reversible identifiers' to scrub sensitive information from its web requests (npm Inc., 2021), so a similar technique could be used to prevent information leakage from secure projects.

Speaking of privacy, anonymisation of contributors is worth considering. In my implementation, an optional flag replaces the contributors' username strings with their MD5 hashes in the client JavaScript[28]. It's unclear whether the contributors' identities should be disclosed in a production version of this system. The underlying data is readily available on GitHub, but it's possible that the context of risk and influence in this visualisation implies that some contributors should be trusted above others. Singling out individuals as a security risk compared to singling out packages seems more controversial, and the potential impacts should be greatly considered before releasing this feature.

[27]Considering visualisation libraries such as D3.js would still be needed, browser scripts are still required to load the correct data into the graph, but these can be embedded in the HTML output.

[28]This is of course insecure; anonymisation should be done on the server, but this feature was only added to produce those screenshots.

9 Conclusions

In this project, I have created a novel visualisation of open-source software package dependencies in the NPM registry. Starting with research of risk factors within and surrounding third-party source code, I identified and acquired necessary sets of data and linked them together in a graph database model. I developed a web application to construct reports about the security of an existing project, and of that project with a new dependency. End users are given contextualised scores that indicate risk, and an abstracted grade for easy comparison between similar packages. Interesting results were also discovered and documented during the processing of the data; namely that a large proportion of packages in the NPM registry have no dependencies, nor any dependent packages in NPM.

While the web application was successful in generating visualisations of the dependencies and associated risks of any package in the NPM registry, it faced a number of limitations, which were mostly due to the size of the dataset, the limited access to repository metadata (i.e. the API rate-limiting), and the continuously changing nature of the dependency graph. Recommendations for overcoming these limitations include implementing a live service that updates the database in real-time and replacing the web application with a command-line interface that generates static HTML reports.

To further assess the project's success, I compare and contrast my result with existing solutions - specifically Synk's Advisor, as shown in Section 3.3. Most obviously in common, Advisor gives a qualitative overall score, calculated from the scores of sub-categories, based on data accessed from the source code repositories and package registries. In particular, they specifically use statistics like package age, commit frequency, contributor counts and package size, as I do. The 'Dependents' statistic in the Popularity category (see Figure 3) reveals that they too are operating on the whole NPM dependency network, but not presenting it graphically as I am.

For other differences, I notice they feature download statistics (a data point I was not able to acquire) quite prominently as a proxy for package popularity, which was considered earlier in my report. Their Security category is much more detailed than my statistics on advisories, instead giving a breakdown of high/medium/low-severity vulnerabilities for each 'significant version' of the package and its dependencies. This feature relies on their proprietary dataset of package vulnerabilities, which is likely more detailed than the NPM security advisories I used. Finally, Advisor has a greater focus on the operations side of software development, detailing aspects like the package's licence and Code of Conduct.

While I feel my visualisation is commensurate with Synk's, theirs has many features that outperform mine (along with a more compact and readable UI), and it's a useful insight into properties they consider valuable to developers.

9.1 Future work

One obvious direction of future work is expanding the visualisation, taking on the suggestions presented throughout Sections 7 and 8. The suggestion of evidence-backed scoring categories would be of particular academic interest, as the system would then be able to suggest a definitive ordering of the quality of software packages, with literature to justify the claims.

Another worthwhile research direction would be a field study with software developers, both to assess whether such a visualisation would be effective in influencing dependency choices, but also to collect feedback and suggestions about the visualisation. Similarly to how I used Pashchenko, Vu and Massacci (2020)'s observations to inspire this visualisation, the feedback would give valuable insights for improving the applicability of the system and reducing harms, as argued by proponents of Responsible Innovation (Stahl et al., 2014).

Lastly, it might be beneficial to apply this methodology to services other than NPM and GitHub. The methodology is kept suitably abstract such that other package managers and version control servers can be used, such as PyPI (the Python Package Index). But with enough abstraction of the input data, it would be possible to apply the visualisation techniques (and possibly the web application itself) to supply chains outside of software packages. One idea is Docker Images, which are able to inherit packages and operating systems from a base Image (Merkel, 2014). Synk actually offers security analysis for Docker Images, but focuses more on the OS packages in an Image rather than the chain of base Images (Armstrong, 2020). It could also theoretically be applied to aspects of physical supply chains, or academic paper citations, or any large directed graph of dependent entities with associated risk factors.

References

@adam-npm (Nov. 2018). *npm Blog Archive: Details about the event-stream incident.* URL: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident (visited on 18/03/2021).

@izs (Mar. 2016). *npm Blog Archive: kik, left-pad, and npm.* URL: https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm (visited on 09/01/2021).

Armstrong, Jim (Nov. 2020). *COntainer Security Guide —What is Container Security —Synk.* URL: https://snyk.io/learn/container-security/ (visited on 25/03/2021).

Bilgin, Zeki et al. (2020). 'Vulnerability Prediction From Source Code Using Machine Learning'. In: *IEEE access* 8, pp. 150672–150684. DOI: 10.1109/ACCESS.2020.3016774.

Brown, Paul (Jan. 2017). *State of the Union: npm.* URL: https://www.linux.com/news/state-union-npm/ (visited on 27/02/2021).

Decan, Alexandre, Tom Mens and Eleni Constantinou (Sept. 2018). 'On the Evolution of Technical Lag in the npm Package Dependency Network'. In: IEEE. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00050.

Díaz, Gabriel and Juan Ramón Bermejo (2013). 'Static analysis of source code security: Assessment of tools against SAMATE tests'. In: *Information and software technology* 55 (8), pp. 1462–1476. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2013.02.005.

Francy, George (2005). 'Sparklines as effective graphics'. In: *Theses* 402. URL: https://digitalcommons.njit.edu/theses/402.

Gegick, M, P Rotella and L Williams (2009). 'Predicting Attack-prone Components'. In: IEEE, pp. 181–190. ISBN: 9781424437757. DOI: 10.1109/ICST.2009.36.

GitHub (Nov. 2018). *Backdoored dependency? flatmap-stream-0.1.1 and flatmap-stream-0.1.2 · Issue #115 · dominictarr/event-stream · GitHub.* URL: https://github.com/dominictarr/event-stream/issues/115 (visited on 09/01/2021).

— (2021). *Repositories - GitHub Docs.* URL: https://docs.github.com/en/rest/reference/repos (visited on 16/02/2021).

Google (Aug. 2019). *Google JavaScript Style Guide (4.4 Column limit: 80).* URL: https://google.github.io/styleguide/jsguide.html#formatting-column-limit (visited on 28/03/2021).

Hovsepyan, Aram et al. (2012). 'Software vulnerability prediction using text analysis techniques'. In: ACM, pp. 7–10. ISBN: 9781450315081. DOI: 10.1145/2372225.2372230.

Kim, Louis (Nov. 2018). *OSF —Draper VDISC Dataset - Vulnerability Detection in Source Code Wiki.* URL: https://osf.io/d45bw/wiki/home/ (visited on 16/01/2021).

Lauinger, Tobias et al. (2017). 'Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web'. In: *Proceedings 2017 Network and Distributed System Security Symposium.* DOI: `10.14722/ndss.2017.23414`. URL: `http://dx.doi.org/10.14722/ndss.2017.23414`.

Merkel, Dirk (2014). 'Docker: lightweight linux containers for consistent development and deployment'. In: *Linux journal* 2014.239, p. 2.

Murdoch, Stuart and Nick Leaver (2015). 'Anonymity vs. trust in cyber-security collaboration'. In: *Proceedings of the 2nd ACM Workshop on Information Sharing and Collaborative Security*, pp. 27–29.

Neuhaus, Stephan and Thomas Zimmermann (2009). 'The Beauty and the Beast: Vulnerabilities in Red Hat's Packages.' In:

Nguyen, Viet and Le Tran (2010). 'Predicting vulnerable software components with dependency graphs'. In: ACM, pp. 1–8. ISBN: 9781450303408. DOI: `10.1145/1853919.1853923`.

NIST (Feb. 2009). *SAMATE:About - SAMATE.* URL: `https://samate.nist.gov/Main_Page.html` (visited on 16/01/2021).

— (May 2016). *Cyber Supply Chain Risk Management —CSRC.* URL: `https://csrc.nist.gov/projects/cyber-supply-chain-risk-management` (visited on 09/01/2021).

— (Apr. 2018). 'Framework for Improving Critical Infrastructure Cybersecurity'. In:

— (2021). *NVD - CVSS Severity Distribution Over Time.* URL: `https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time` (visited on 18/03/2021).

npm Inc. (2021). *npm-audit —npm Docs.* URL: `https://docs.npmjs.com/cli/v6/commands/npm-audit` (visited on 03/01/2021).

Open Source Initiative (Mar. 2007). *The Open Source Definition (Annotated).* URL: `https://opensource.org/docs/definition.html` (visited on 28/03/2021).

Pashchenko, Ivan, Duc-Ly Vu and Fabio Massacci (2020). 'A qualitative study of dependency management and its security implications'. In: pp. 1513–1531.

Pearson, Hilary E (2000). 'Open source licences: Open source—the death of proprietary systems?' In: *Computer Law & Security Review* 16.3, pp. 151–156.

Pittenger, Mike (Dec. 2016). *Open Source Security Analysis: The State of Open Source Security in Commercial Applications.* URL: `https://sq-software.com/wp-content/uploads/2016/12/2016-12-OS-Security-Analysis.pdf` (visited on 18/03/2021).

Podjarny, Guy (June 2016). *The 5 dimensions of an npm dependency —Synk.* URL: `https://snyk.io/blog/whats-an-npm-dependency/` (visited on 16/02/2021).

Pressman, Roger S (2005). *Software engineering: a practitioner's approach.* Palgrave macmillan.

ReviewMeta (Apr. 2016). *How it Works —ReviewMeta Blog.* URL: `https://reviewmeta.com/blog/how-it-works/` (visited on 15/02/2021).

Shin, Yonghee (2011). 'Investigating Complexity Metrics as Indicators of Software Vulnerability'. ISBN: 978-1-124-47833-3.

Shin, Yonghee and Laurie Williams (2008). 'An empirical model to predict security vulnerabilities using code complexity metrics'. In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 315–317.

Shneiderman, Ben (Nov. 1997). 'A Thousand-fold increase in human capabilities'. In: *Educom Review*, pp. 4–10.

Silic, Mario and Andrea Back (2016). 'The influence of risk factors in decision-making process for open source software adoption'. In: *International Journal of Information Technology & Decision Making* 15 (01), pp. 151–185.

Stack Overflow (May 2020). *Stack Overflow Developer Survey 2020 (Other Frameworks, Libraries, and Tools).* URL: `https:/%20Other%20Frameworks,%20Libraries,%20and%20Tools%20-%20/insights.stackoverflow.com/survey/2020#technology-other-frameworks-libraries-and-tools` (visited on 28/03/2021).

Stahl, Bernd Carsten et al. (2014). 'From computer ethics to responsible research and innovation in ICT: The transition of reference discourses informing ethics-related research in information systems'. In: *Information & Management* 51.6, pp. 810–818.

Sultana, Kazi Zakia (2018). 'Towards a software vulnerability prediction model using traceable code patterns and software metrics'. In: IEEE, pp. 1022–1025. ISBN: 9781538626849. DOI: `10.1109/ASE.2017.8115724`.

ToS;DR (2012). *About Us · ToS;DR.* URL: `https://tosdr.org/en/about` (visited on 15/02/2021).

Unwin, Antony and Heike Hofmann (1999). 'GUI and Command-line-Conflict or Synergy?' In: *Computing Science and Statistics*, pp. 246–253.

Zhang, Su et al. (2015). 'Assessing attack surface with component-based package dependency'. In: pp. 405–417.