

Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM

Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, Emad Shihab

Data-driven Analysis of Software (DAS) Lab

Concordia University

Montreal, Canada

{jasmine.latendresse,suhaib.mujahid,diego.costa,emad.shihab}@concordia.ca

ABSTRACT

Modern software systems are often built by leveraging code written by others in the form of libraries and packages to accelerate their development. While there are many benefits to using third-party packages, software projects often become dependent on a large number of software packages. Consequently, developers are faced with the difficult challenge of maintaining their project dependencies by keeping them up-to-date and free of security vulnerabilities. However, how often are project dependencies used in production where they could pose a threat to their project's security?

We conduct an empirical study on 100 JavaScript projects using the Node Package Manager (npm) to quantify how often project dependencies are released to production and analyze their characteristics and their impact on security. Our results indicate that most project dependencies are not released to production. In fact, the majority of dependencies declared as runtime dependencies are not used in production, while some development dependencies are used in production, debunking two common assumptions of dependency management. Our analysis reveals that the functionality of a package is not enough to determine if it will be shipped to production or not. Findings also indicate that most security alerts target dependencies not used in production, making them highly unlikely to be a risk for the security of the software. Our study unveils a more complex side of dependency management: not all dependencies are equal. Dependencies used in production are more sensitive to security exposure and should be prioritized. However, current tools lack the appropriate support in identifying production dependencies.

KEYWORDS

third-party packages, dependencies, security, npm

ACM Reference Format:

Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, Emad Shihab. 2022. Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM. In *Proceedings of The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. ACM, New York, NY, USA, 11 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE 2022, 10 - 14 October, 2022, Michigan, United States

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06...\$15.00

<https://doi.org/XXXXXXXX.XXXXXXX>

1 INTRODUCTION

The vast majority of modern software systems are built by using modular functionalities provided by open source packages. Reports estimate that more than 90% of open source and proprietary projects rely substantially on reusing open source packages [2, 6]. As a testament to the popularity of open source, popular package managers such as npm, host more than 2 million reusable packages, covering all sorts of software functionalities [21].

While the use of open source packages significantly reduces development time and costs [16, 30, 40], it also exposes software applications to vulnerabilities. In the 2020 State of the Octoverse security report, GitHub reveals that active repositories with a supported package ecosystem have a 59% chance of getting a security alert in the next 12 months [6]. This problem is even more widespread in the JavaScript ecosystem, where nearly 40% of all npm packages rely on code with known vulnerabilities [1]. Software vulnerabilities may lead to significant financial and reputation loss. A popular example is the 2017 Equifax cybersecurity incident caused by a web-server vulnerability in the Apache Struts package. The incident led to a data breach of millions of american citizens, costing Equifax 1.8 billion USD in security upgrades and lawsuits [22].

The problem is that developers struggle to identify what vulnerabilities may affect their software application [28]. Current security scanners report on the severity of a vulnerability, but lack a support to identify if the dependency is 1) used in the code and 2) is part of the production software the project delivers. Developers constantly complain that security alert tools report too many false positives [32, 33], as even the most critical vulnerability may be unexploitable if the vulnerable dependency is never released in the production software.

In this paper, we study how often dependencies are actually part of a production software and their impact on security based on their characteristics, usage, and context. We perform this study on 100 JavaScript projects in npm, the largest and fastest growing software ecosystem to date [31], to answer the following three research questions:

- RQ1. How many installed dependencies are production dependencies?
- RQ2. What are the characteristics of production dependencies?
- RQ3. How often are npm security alerts emitted for production dependencies?

Findings show that that production dependencies represent a very small fraction of the total number of dependencies in each project. While projects tend to depend on hundreds of dependencies (both direct and transitive), 51 projects did not have any production

dependencies, and 49 have a median of 5 production dependencies. Contrary to common assumptions, most dependencies declared as runtime are not shipped to production while some development dependencies are included in the production software. Consequently, we find that dependency usage and context gives better insight at determine if a dependency will be used in production than the nature of a dependency itself. Furthermore, our results show that not all security vulnerabilities reported by npm are an actual threat to the security in production. Our paper makes the following contributions:

- To the best of our knowledge, this is the first study to investigate the discrepancy between installed and production dependencies in open source projects.
- We report on results that challenge the assumptions of dependency management and should be revisited by researchers and practitioners.
- We investigate the support of current tools in providing better information for developers regarding the scope and context of vulnerable dependencies.
- We make our dataset of 100 projects available¹, including all scripts used to collect and pre-process data, to facilitate replication and foment more research in the field.

The rest of the paper is organized as follows: we start by motivating our problem with an example in Section 2. We describe and justify our methodology in Section 3 and explain our results in Section 4. Implications of our findings are discussed in Section 5. We present the related work in Section 6, and discuss the limitations to our study in Section 7. Finally, we conclude our study in Section 8.

2 MOTIVATION & BACKGROUND

To motivate our study and illustrate the terminology used in this paper, we walk the reader through the creation of a simple application using create-react-app [7]. This example application is a single-page "Hello World" application that is provided by React when initializing a Create React App project. We create our application by simply running the command `npm create-react-app my-app`.

How many dependencies in our project? To achieve this single-page React application without further programming, our generated application reuses several open source packages published in npm. We refer to each of the packages as a *dependency* of our project. The dependency configuration of our project is stored in the `package.json` file, shown in Figure 1. Dependencies are grouped into two groups: *runtime dependencies* ("dependencies") and *development dependencies* ("devDependencies"). Runtime dependencies are dependencies required by the application to function, e.g., as our we build a React application, our project depends on react version 17.0.2. Development dependencies, on the other hand, are needed to develop the project, e.g., to format the code (prettier 2.5.1), and are not required by the software to run. As it can be seen in Figure 1, our small application has 7 runtime dependencies and 9 development dependencies.

Once we install these dependencies locally to build and test our application (`npm install`) we may be surprised to see that a total of 1,764 dependencies were installed. The dependencies

Figure 1: A snippet of the `package.json` file listing the dependencies of our example project.

```

1  "dependencies": {
2    "@testing-library/jest-dom": "^5.16.2",
3    "@testing-library/react": "^12.1.3",
4    "@testing-library/user-event": "^13.5.0",
5    "react": "^17.0.2",
6    "react-dom": "^17.0.2",
7    "react-scripts": "5.0.0",
8    "web-vitals": "^2.1.4"
9  },
10 "devDependencies": {
11   "@webpack-cli/generators": "^2.4.2",
12   "css-loader": "^6.6.0",
13   "html-webpack-plugin": "^5.5.0",
14   "prettier": "^2.5.1",
15   "style-loader": "^3.3.1",
16   "webpack": "^5.69.1",
17   "webpack-cli": "^4.9.2",
18   "webpack-dev-server": "^4.7.4",
19   "workbox-webpack-plugin": "^6.5.0"
20 }

```

shown in Figure 1 are *direct dependencies* of our project, each of which have dependencies of their own. For instance, the package `loose-envify` is a dependency of `react`. These are called *transitive dependencies* and represent the vast majority of dependencies installed. As such, `loose-envify` is a transitive dependency of our example application. We use the term *installed dependencies* to refer to all dependencies of a project, both direct/transitive and development/runtime dependencies.

Is our application vulnerable? Security vulnerabilities are a widespread problem in npm [1]. Given that our application depends on 1,764 installed dependencies, is our application affected by vulnerabilities? To verify this, we resort to using a Software Composition Analysis (SCA) tool. SCA tools are used to identify open source components in software codebases to evaluate security, license compliance and overall code quality [10]. In our example, we use `npm audit`, a native tool of npm that reports vulnerabilities affecting software dependencies and maintains its own database of vulnerabilities. If a dependency is affected by one of more vulnerabilities, we refer to the dependency as a *vulnerable dependency*. In our example application, upon running `npm audit`, we receive the report that our simple application contains 6 moderate severity vulnerabilities, 13 high severity vulnerabilities, and 1 critical severity vulnerability. That is, without any further programming, our project already started with an alarming number of vulnerabilities of moderate, high, and critical severity. Examples of the reported high severity and critical severity vulnerabilities include Regular Expression Denial of Service, Template Injection, and Prototype Pollution.

Can reported vulnerabilities really affect our example application in production? Vulnerable dependencies are problematic and may affect the security of our project in multiple ways. However, the risk of vulnerable dependencies reaches its peak when the dependency is needed for the software to run in a production environment. To find which dependencies are part of our production software, i.e., *production dependencies*, we use a module bundler.

¹<https://zenodo.org/record/6518765>

Figure 2: A snippet of the source map generated by building our example project.

```

1  "version": 3,
2  "file": "main.js"
3  "mappings": "KAAK,CAACC,EAAOC,GAAL.."
4  "sources": ["node_modules/css-loader/dist/runtime/api.js",
5             "node_modules/object-assign/index.js",
6             "node_modules/react-dom/cjs/react-dom.production.min.js",
7             "node_modules/react/cjs/react.production.min.js",
8             "node_modules/scheduler/index.js",
9             "node_modules/style-loader/injectStylesIntoStyleTag.js"]

```

A *module bundler* is a tool that assists the building process of a software by resolving the software dependencies and pruning the dependencies that are not needed in the production software. The process of pruning dependencies is referred to as *tree shaking*. We use *webpack* [3], a popular JavaScript module bundler, to build our production software and export a list of production dependencies.

Upon building our project with *webpack*, the tool generates a *source map* file, which contains the list of production dependencies of our application. From the 1,764 dependencies in our example project, Figure 2 shows that only 6 are released to production: *react*, *object-assign*, *scheduler*, *react-dom*, *style-loader*, and *css-loader*. More so, none of our production dependencies contained any reported vulnerability, thus, our original report of 15 vulnerabilities affected dependencies that would not be present in the application in production.

The problem: security alert fatigue. Our example showcases an important problem in current software development. Even small applications may depend on thousands of dependencies and vulnerabilities are constantly being reported by the open source community. Developers face the difficult challenge to separate security alerts that are relevant to their application security from the long reports yielded by current SCA tools [32, 33]. In this paper, we evaluate this problem on a scale of 100 popular JavaScript projects.

3 STUDY DESIGN

The goal of the paper is to study how often project dependencies are shipped to production and its impact on the security of software projects. In this section, we describe how we select and curate the set of study projects (Sections 3.1 and 3.2), and how we identify production dependencies (Section 3.3). We provide an overview of our methodology in Figure 3.

3.1 Dataset of Candidate Projects

The focus of our study is to investigate how active software development JavaScript projects use their dependencies. To this aim, we start by collecting data of a large number of JavaScript repositories as candidate projects for our study. Many studies have used the number of GitHub stargazers as a way to select candidate projects [19, 23, 36]. Thus, we start with 11,860 popular JavaScript projects that were collected on July 27th, 2020 with at least 100 stargazers.

Finding what dependencies are shipped to production is a very challenging task making it impractical to apply this analysis on a large-scale [42]. In our study, we opt to select projects that already make use of tree shaking (see Section 2 for a more in-depth

explanation). Specifically, we select projects using either *webpack* or *rollup* [9] because they are two of the most popular module bundlers for JavaScript projects and they have integrated tree shaking support.

To find out which projects use *webpack* and *rollup*, we automatically parse the *package.json* files of the 11,860 projects to identify 1) if any of the bundlers are declared as a dependency and 2) the tree shaking algorithm is enabled for the project. Through this process, we find that 155 JavaScript projects make use of *webpack* or *rollup*, and have tree shaking enabled.

3.2 Building Candidate Projects

To assess whether a dependency is used in production, we have to successfully build each project in a production environment with a module bundler. During the build of a project, the module bundler first looks for all of the dependencies in the project and constructs a dependency graph (dependency resolution). The dependency graph is then converted along with source code into a single file (packing) called the bundle. Source maps are then generated after a successful build.

To build the candidate projects, we first clone each of the 155 JavaScript projects locally. We planned to build a framework to automate the build of all the 155 JavaScript projects. However, we soon realized that many projects require specific building commands and setup to be build successfully. In fact, the majority of the projects did not support the standard build command (`npm run-script build`). Furthermore, the environmental settings varied across projects, e.g., some projects require specific NodeJS versions and identifying this automatically is very challenging. We then proceed to semi-manually build each project using the following methodology:

- (1) **Read projects documentation.** We read the documentation of all the 100 studied projects to identify the specifics of each project build. The goal of this step is to identify all the steps of the building process: the build commands, the supported Node versions, the supported package manager (e.g. *YAML* or *npm*), and any other specificity of the project building configuration. At this point, we also confirmed that all selected projects are related to software development, i.e., are not personal toy-projects.
- (2) **Install dependencies.** We install all dependencies specified in the *package.json* file by using the `npm install` command. This generates a *node_modules* folder in every project's home directory which contains all installed dependencies.
- (3) **Build project.** Following each project's documentation, we build each project in the dataset. The first author manually followed the steps of the building process to ensure the build was

Table 1: Descriptive Statistics of the Selected Projects.

	Mean	Median	Min	Max
# stars	4827.6	1224	112	74201
# commits	1364.9	496	31	6188
# contributors	62.3	26	4	401
age (years)	5.2	5	1	12

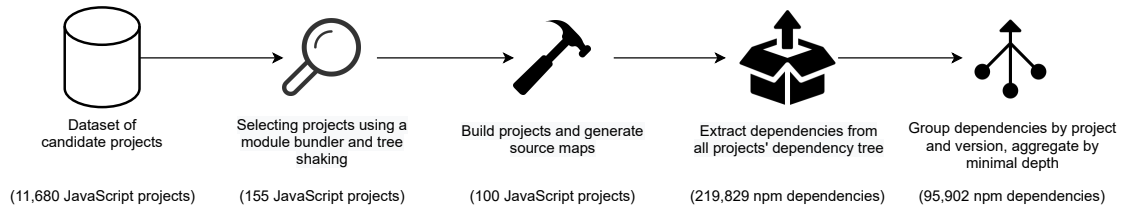


Figure 3: Overview of our approach for filtering projects and collecting dependencies.

successful, the source maps containing the production dependencies was generated, and the yielded artifacts targeted the production environment.

- (4) **Generate source maps.** Upon the successful completion of the building process, source maps are generated and saved in the project's temporary folder or home directory.

After our careful process, we successfully build and generate source maps for 100 JavaScript projects. From the 55 projects that failed in our process, the main culprit was the generation of the source maps file. In most of the failed cases, projects' configuration did not have the flexibility to generate the source maps file. For example, we found some projects created using the `create-react-app` package that do support module bundlers and tree shaking, but did not have the option to output source maps.

We present descriptive statistics of the 100 projects we successfully built and generate source maps in Table 1. The projects of our dataset are very popular (median 1,224 stargazers), tend to be mature projects (median of 5 years of development and 496 commits) and are developed by medium-sized team of developers (median of 26 developers).

3.3 Identifying Dependencies in Production

To identify production dependencies, we first collect all dependencies found in the source maps of each projects using a mix of source map parser [38] and regular expression (regex). Then, to obtain the version of each dependency found in the source maps, we locate its `package.json` file in the respective project's `node_modules` folder and parse it. This results in a dataset of production dependencies with their corresponding version.

In addition to identifying dependencies used in production, we also want to identify two very important characteristics of all dependencies, as they have an influence on the risk of vulnerabilities [25]: 1) the dependency scope, runtime or development and 2) whether the dependency is a direct or transitive dependency of the project. To classify a dependency into runtime or development, we analyse the `package.json` file of a project, classifying dependencies configured in the "dependency" section as runtime dependencies, and classifying dependencies declared in "devDependency" as development dependencies. Since transitive dependencies are not listed in the `package.json` file, we identify the type of the original dependency which determines the type of the transitive dependency.

To classify installed dependencies into direct or transitive dependencies, using the command `npm list` we generate the *dependency tree*, a hierarchical representation of relationship between dependencies. The `npm list` command lists all installed dependencies in json format, including the name, version, path, and depth of

each dependency. From the depth, we identify each dependency as direct or transitive, i.e., direct dependencies have depth = 1, while transitive dependencies have depth > 1.

Our methodology has one limitation. Peer dependencies are used to decouple dependencies between projects, to ensure a single version of the package is installed for all dependencies. For example, in applications with many npm packages depending on `react`, `react` can be declared as a peer dependency to prevent the installation of multiple (possibly conflicting) versions of `react`. Unlike runtime and development dependencies, peer dependencies are not automatically installed by npm. Instead, they must be included by the code that uses the package as a dependency. We find that 37 projects in our dataset have missing peer dependencies. Since it is not possible to automatically resolve missing peer dependencies for all 37 projects, we exclude the dependencies from our analysis.

4 RESULTS

In this section, we present the results of our three research questions. For each research question, we present its motivation, the approach to answer the question, and the results.

RQ1: How many installed dependencies are production dependencies?

Motivation: While reusing packages may reduce development time, developers have to constantly maintain their dependencies to fix bugs in the packages and mitigate the problems of vulnerable dependencies [14, 15, 26]. However, identifying dependencies used in production is not a trivial task which makes it difficult to prioritize dependency-related maintenance activities [32].

In this research question, we want to assess how often dependencies of the selected projects are actually production dependencies. Answering this question is the first step to understand how often a runtime and development dependency is used in production. It will also help us better understand how dependencies are used in practice and how they impact the security of software.

Approach: To approach this research question, we use the methodology described in Section 3.3. That is, we start by installing all dependencies from each project to retrieve the list of installed dependencies and their respective versions. To classify an installed dependency into direct or transitive, we generate the dependency tree of each studied project. Then, to identify production dependencies, we build all software projects with their respective module bundler (`webpack` or `rollup`). This building process was done manually by following the building steps specified in the project documentation, to ensure each project is built correctly and without errors. After

Table 2: Dependency profile of projects with and without production dependencies in absolute numbers and median of aggregated value per project. The percentages are always in relation to the # of Installed Dependencies.

Dependencies	Projects with Zero Production Deps		Projects with 1+ Production Deps	
	Total	Median	Total	Median
Installed	46,031 (100%)	851	53,421 (100%)	1,017
Runtime	1,005 (2.1%)	0	873 (1.6%)	5
Dev	45,025 (97.9%)	832	52,542 (98.4%)	1,017
Direct	1,539 (3.4%)	40	2,098 (3.9%)	29
Transitive	44,492 (96.6%)	809	51,307 (96.1%)	963
Production	-	-	497 (0.9%)	5

the building of the project, we analyze the yielded source maps to identify the production dependencies. Finally, we cross reference the installed dependencies and production dependencies to classify each project dependency into production/non-production, runtime/development, direct/transitive and report our findings.

Finding 1: Of the 100 projects, 51 projects contain no production dependencies. To make a better sense of our results, we split our dataset of 100 projects into two sets: projects with production dependencies (49 projects) and projects without production dependencies (51 projects). Table 2 shows the total number of dependencies and their characteristics in both sets of projects. The 51 projects with no production dependencies have installed a total of 46 thousand dependencies, including direct and transitive packages, however, none of the installed dependencies are used in production. More interestingly, among the installed dependencies, there were 1,005 packages that were declared to be runtime dependencies, which is supposedly required at the runtime of the the final software, but were not included in the final production artifact.

We also note that the set of 51 projects with no production dependencies have a median of runtime dependencies of zero. We confirm this finding through manual investigation and find that 39 projects in our dataset only declare development dependencies. Most of these projects are libraries meant to be used by other projects as development tools. Examples of such projects are Vuex, a state management pattern for Vue.js applications; three.js, a popular cross-browser 3D library; and polished, a lightweight toolset for writing styles in Javascript. All those library projects have the incentive to depend on little to no runtime dependencies, as the fewer dependencies they have, the less constrained their users may be to rely on their libraries [13, 18].

Finding 2: From the 49 projects with production dependencies, production dependencies represent less than 1% of the installed dependencies. The results show that projects with production dependencies have a total of 53,421 dependencies, of which only 497 dependencies (0.93%) are released to production (see Table 2). Analyzing the median number of dependencies per project (see Median column in Table 2), we find that projects have in median 5 production dependencies while depending in median over a thousand dependencies. However, we notice that not all runtime

Table 3: Characteristics of dependencies in projects with production dependencies.

		Direct	Transitive	Total
Production	Dev	62	77	139
	Runtime	175	178	353
Non-production	Dev	1,809	50,594	52,403
	Runtime	52	458	510
	Total	2,098	51,307	53,405

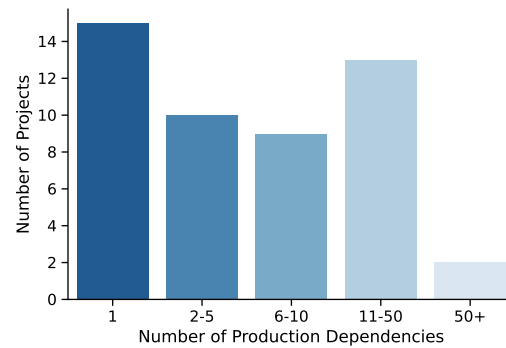


Figure 4: Number of production dependencies on the 49 project with one or more production dependencies.

dependencies are used in production. The total number of runtime dependencies installed (873) far exceeds the number of production dependencies (497), indicating that many runtime dependencies may be incorrectly configured or not used in the code.

Figure 4 shows the distribution of the number of production dependencies per project. We can observe that 15 projects contain a single production dependency and the vast majority of projects (65.3%) has less than 10 dependencies used in production. Still, we found some projects that depend heavily on packages in their production build, with ProjectMirador being the project with the most production dependencies in our dataset with 92.

Finding 3: More than half (59%) of the runtime dependencies are not used in production. As shown in the "Production" row of Table 3, we find that 510 out of 863 of the total runtime dependencies are not shipped to the production bundle. Runtime dependencies are dependencies (supposedly) required by the application to run. Our results, however, show that in the majority of the cases dependencies are declared as runtime but are not actually used in the code, thus, are excluded by the module bundler during the build. This finding suggests developers mistakenly maintain unused dependencies in their project configuration, which suggests that they lack the necessary information to determine whether a dependency is actually used by the software in production. This is corroborated by related work [26], where authors reported that unused dependencies occur in 80% of studied projects.

51 out of 100 projects do not use any dependencies in production. The 49 projects that ship dependency to production contain less than 1% of production dependencies. Contrary to common belief, 59% of runtime dependencies are not used in production.

RQ2: What are the characteristics of production dependencies?

Motivation: Production dependencies are the prime security liability in software systems since they can compromise a running software [42]. Current SCA tools may not distinguish dependency scope (i.e., production, non-production) [28, 32], which may lead to reporting unexploitable vulnerabilities (false positives). They may also only consider direct dependencies although vulnerabilities can be introduced transitively [20, 29, 32]. The problem is that assumptions about production dependencies are not always correct. In fact, RQ1 showed that runtime dependencies are not always in production. In this research question, we study the characteristics production dependencies to establish a practical understanding of how they are used and in what context they are used. Such findings help in improving current SCA tools as they provide insights on how dependencies are used in practice.

Approach: To identify the characteristics of dependencies used in production, we consider the production dependencies identified in RQ1 and classify them based on their scope (runtime, development), depth, and usage (e.g., package A is used in production X times, and not used in production Y times).

In theory, one can identify the scope of a dependency by looking at the nature of the functionality provided by a package. For instance, packages that provide development utilities should not become production dependencies. To investigate to what extent the nature of the package determines if it will be used as a production dependency, we analyze how packages are released to production across the 100 studied projects. We analyze a total of 1,269 unique packages. We then classify the packages in three categories: 1) packages that are always used in production, 2) packages that are never used in production, and 3) packages that are sometimes used in production.

Finding 4: 28.2% of production dependencies are development dependencies. The first section of Table 3 shows the characteristics of production dependencies. We find that 28.2% of production dependencies are development dependencies and the remaining 71.7% are declared as runtime dependencies. It is expected that all dependencies released to production consist of runtime dependencies since they provide the application with specific functionalities to be used by the client. It is then surprising to find that almost 30% of the dependencies released to production are development dependencies since such dependencies are, by default, not included in the production bundle.

While unusual, having a development dependency in production occurs in 37 of the projects in our dataset. This can occur for several reasons: First, the selected projects use module bundlers, which disregard the configuration of the package.json file and uses source code analysis to identify what should be a production dependency. Developers may not be as careful to specify their development dependencies as their building process does not depend

on a correct specification of development and runtime dependencies [4]. Second, it can be that the dependency is initially declared under the "dependencies" property of the package.json file, but is intentionally moved by the developer to "devDependencies" to get rid of security warnings, as it is explained in a create-react-app GitHub issue [8]. The author of the issue explains that `npm audit` reports vulnerabilities for code that never runs in production, but strictly at build time in development. They then suggest to move vulnerable dependencies to "devDependencies" to get rid of the security warning. We believe development production dependencies are unlikely to happen in projects that do not use module bundlers. By default, npm does not include development dependencies in a production build. This means that a project that requires a development dependency at runtime will not function because of the missing dependency.

Finding 5: The majority of production dependencies (51.8%) are transitive dependencies. Looking at the Transitive columns of Table 3, we notice that 51.8% of production dependencies are dependencies of their direct project dependencies. These results suggest that developers may not have control over the majority of production dependencies. Naturally, a transitive dependency can only be released to production if the original dependency is also released to production. Hence, developers should be extra careful when selecting production dependencies, preferably by selecting packages that have little to no production dependencies on their own, to reduce the attack surface through vulnerable dependencies.

Finding 6: The 237 production dependencies come from 183 unique npm packages. From these, 43 are sometimes not used in production in other projects. To put things in perspective, our dataset contains 1,269 unique npm packages. From this, we find that 1,086 (85.6%) are never used in production since they offer functionalities that are development-only. For example, `eslint` installed in 79 projects, is a static code analysis tool that is used to identify problematic patterns found in JavaScript code, `@babel/core` installed in 70 projects, is a command line interface tool that facilitates working with `babel`, and `rollup` installed in 65 projects, is a build tool for JavaScript projects.

For the rest of the packages, we find that 183 are used in production at least once. Taking a closer look at the production packages, we find that 140 (76.5%) are always used in production when installed in a project and that such packages do not occur frequently. In fact, they occur at most in 2 different projects and are installed as runtime dependencies. For example, `is-promise`, a library that tests whether an object is a `promises-a+ promise`, `query-string`, a library that parses and stringifies URL query strings, and `react-fast-compare`, a library that provides specific handling of fast deep equality comparison for React, are all installed in 2 projects, and used in production 100% of the time they are installed.

Interestingly, we find that 43 (23.5%) of the 183 production packages are not always shipped to production. We show in Table 4 10 examples of such packages, and how often they are in production versus how often they are installed. The results show that `react`, a library for building user interfaces, is the most frequently installed package appearing in 40 projects, but is only released to production in 4 projects. In contrast, `react-redux`, a React binding for Redux

Table 4: Frequently installed packages that are both used and not used in production.

Package	# Production Installations	Total # Installations	% in Production
react	4	40	10%
react-dom	3	37	8.1%
prop-types	13	23	56.5%
@babel/runtime	10	19	52.65%
lodash	4	14	28.6%
core-js	5	13	38.5%
classnames	5	8	62.5%
react-is	1	5	20%
react-redux	4	5	80%

allowing React components to read data from a Redux store, only appears in 5 projects, but is released to production in 4 out of 5 projects. In only one project (`redux-little-router`) is `react-redux` not released to production and declared as a development dependency. We further inspect the package `.json` of `redux-little-router` and find that `react-redux` is a peer dependency, thus, it is not included in the production bundle of the project. It is also worth noting that `redux-little-router` is a lightweight library that provides flexible React bindings and components. Thus, the project makes a conscious effort to mitigate dependency bloat, declaring most of its dependencies as development dependencies, and including `react`, `react-dom`, `react-redux`, and `redux` as peer dependencies.

The main takeaway from this finding is that we cannot identify production dependencies by looking at the nature and functionalities of a package alone. As we have shown, the scope of a dependency can vary based on the context and usage of a package, which means it may differ from project to project. Thus, it is important for SCA tools to include this scope analysis in their approach so that developers can more easily identify production dependencies based on their own usage and context.

Our findings indicate that 28.2% of production dependencies come from development dependencies and that 51.8% come from transitive dependencies. The functionality of the package alone does not determine if they will be shipped to production: 43 of 183 packages encountered in production in one project are not shipped to production in other projects.

RQ3: How often are npm security alerts emitted for production dependencies?

Motivation: The observations made in RQ1 suggest that the majority of the dependencies are not used in production. While vulnerabilities in non-production dependencies may affect the development environment (e.g., installing packages with malicious code), it is when a vulnerable dependency is released to production that the threat of exploitation reaches its peak [42]. Developers should constantly run scanners to identify security alerts in their project and prioritize fixes in production dependencies, to avoid having their software compromised. The problem is that tools such as `npm audit` often report many false alerts for deployed code, making

Table 5: Characteristics of vulnerable dependencies reported by npm vulnerability alerts.

	Direct	Transitive	Total
Development	9	410	419
Runtime	1	7	8
Total	10	417	427

vulnerability reports noisy and bloating audit resources [8, 34]. In this research question, we investigate how often security alerts are emitted for production dependencies compared to non-production dependencies and the characteristics of vulnerable dependencies.

Approach: To investigate how often vulnerabilities are encountered in production and non-production dependencies, we first generate the `npm` vulnerability report of each project by using the `npm audit` tool. Next, to obtain the `npm audit` reports in a parseable `csv` format, we adapt the `npm-deps-parser` [5], a tool that parses, summarizes, and prints `npm audit json` output to `markdown`. From this, each vulnerability report is identified with the project name, the vulnerable dependency and version, the severity, and a unique link to the GitHub Advisory Database (GAD) [11], a database of security advisories affecting the open source world. To obtain the scope and depth of each vulnerable dependency, we cross-reference the set of vulnerable dependencies with the set of production dependencies and installed dependencies for each project. Because of the limitations discussed in Section 3.3, we could not identify the scope and depth of 29 vulnerable dependencies and exclude them from further analysis.

Finding 7: A total of 608 security alerts are emitted for dependencies of 32 projects, yet none are related to production dependencies. In our dataset, no security alerts were emitted for 68 projects. The remaining 32 projects reported a total of 608 security alerts for 456 vulnerable dependencies, i.e., the same dependency may issue multiple security alerts. In median, these 32 projects reported 16 security alerts, none related to production dependencies.

There are a few reasons as to why security alerts may have been emitted only to non-production dependencies. First, as seen in RQ1, the vast majority of dependencies are not released to production (99%), the chances of vulnerabilities being encountered in non-production are 99x higher than in production dependencies. Second, developers of the selected projects are likely making the conscious effort of updating production dependencies to mitigate security vulnerabilities, since they may be aware of what dependencies may be used in production (e.g., developers open a PR in the project `InstantSearch` to update a vulnerable dependency [39]). The problem, however, is that tools such as `npm audit` make no distinction whether security alerts are referring to non-production dependencies. Developers have to know themselves which dependencies are released to production to filter out relevant security alerts that need urgent action, making it harder to prioritize management efforts.

Table 6: Count of vulnerability reports per severity level with the npm recommended action.

Severity	Vulnerability Severity Recommended action	Dependency	
		Runtime	Dev
low	Address at your discretion	3	33
moderate	Address as time allows	1	226
high	Address as quickly as possible	5	263
critical	Address immediately	0	45

Finding 8: 98.1% (419) of the vulnerable dependencies are development dependencies. In this analysis, we switch from security alert reports to vulnerable dependencies, as multiple reports may be issued for the same dependency under different vulnerabilities. The first row of Table 5 shows the number of development and runtime vulnerable dependencies reported by our experiment. The `npm audit` tool reports security alerts from a total of 419 vulnerable development dependencies, representing 98.1% of all vulnerable dependencies identified. From the 419 vulnerable development dependencies, 9 (2.1%) are direct dependencies, and 410 (97.9%) are transitive dependencies. Next, we analyze the severity of the vulnerability reports in relation to the characteristics of vulnerable dependencies as shown in Table 6. We find that all critical and most of the high-severity reports are emitted for development dependencies.

In some cases, tools such as `npm audit` allow developers to filter out development dependencies from the security reports, as they are supposedly not released to production [8]. It is dangerous, however, to completely ignore the security maintenance of development dependencies. Some development dependencies are used in production, as seen in RQ2, and hence, have the risk of being exploited in a production environment. Vulnerable transitive dependencies that are released to production are equally dangerous since even if they are reported by `npm`, developers are not in control of their update.

32 projects in our dataset reported a total of 608 security alerts, but none of the alerts referred to a production dependency. Projects have in median 16 security alerts, but the vast majority refer to development non-production dependencies which does not represent a threat for their running application.

5 DISCUSSION

In this section, we discuss the implications from our work and possible solutions for current source code based tools.

5.1 Implications

Tracking production dependencies is very challenging. While our study focuses on a selected number of 100 popular JavaScript projects, the results showcase the difficulties of mapping a project’s production dependencies. This difficulty arises primarily because assumptions commonly held by the development community regarding dependency management do not hold in practice:

- (1) Assumption 1: Runtime dependencies are always shipped to production. Our results showed that the majority of dependencies declared as runtime are not used in production (RQ1).

Developers may spend time ineffectively managing runtime dependencies due to security alerts, without confirming that such dependencies are bundled in their delivered software.

- (2) Assumption 2: Development dependencies are never shipped to production. In projects that use module bundler, dependencies declared as development may be shipped to production (RQ2). In fact, development dependencies represent a third of all production dependencies identified in our study. Developers may disregard all their development dependencies as being irrelevant for security upgrades, when in fact, some vulnerable development dependencies are shipped in their delivered software.
- (3) Assumption 3: The functionality provided by the package is sufficient to determine if it is a production dependency. Particularly in cases where packages provide runtime utilities, our results show that 43 out of 183 packages are released to production in some projects but not in others. Thus, the package’s functionality is not sufficient to determine whether a package is used in production (RQ2).

These assumptions have the potential to affect the security of the delivered software, as developers may wrongly assume what dependencies are sensitive to security exploits.

Not all vulnerabilities in dependencies are a security risk for the software in production. Prior research has shown that not all vulnerabilities are relevant for the software in production [33, 34]. In this paper, we expand on this by studying the relevance of dependencies for the security risk of a software in production. Our findings indicate that, given by the prominence of non-production dependencies (RQ1), the vast majority of security alerts will be emitted for dependencies that do not impact the security of a software in production (RQ3).

To put things in perspective, we analyze the types of vulnerabilities reported by `npm audit`. The most common type of vulnerability identified in the studied projects is Regular Expression Denial of Service (ReDoS), accounting for 25.3% of all reported vulnerabilities and for 27% of high severity vulnerabilities. While a diligent dependency management is of utmost importance to mitigate security risks, developers should be mindful of the types of alerts they should prioritize. In the case of a ReDoS attack, the performance of an application is compromised if there is a regular expression that, with malicious input, slows it down exponentially. However, we find that 97.3% of the dependencies affected by a ReDoS vulnerability are development-only, which tend not to be part of a production software. Previous research shows that developers tend to ignore security alerts when they receive a lot of them [28]. Our approach allows them to focus on the important ones first (i.e., security alerts for vulnerable dependencies in production).

Source maps and tree shaking can benefit developers beyond client-side applications. In this paper, we use module bundlers to accurately differentiate between installed dependencies and production dependencies. Module bundlers are most commonly used in client-side applications, which are generally defined as libraries or frameworks running in a Web browser (e.g., React, Vue, Angular) to support the development of Web applications. Module bundlers,

however, can benefit far beyond just client-side applications by helping developers:

- (1) Prioritize addressing security alerts on production dependencies. As security alerts are very commonly issued for projects that rely on open source code, developers should prioritize addressing security issues that have the potential to affect their production software, by identifying vulnerabilities affecting their production dependencies.
- (2) Prioritize maintenance tasks on production dependencies. As projects depend on increasingly high number of software dependencies, updating all dependencies in every release may become increasingly prohibitive. Updating dependencies always have the risk of breaking changes [17], leading to software bugs and mistrust between project maintainers [26]. Hence, developers should prioritize updating production dependencies to focus their maintenance tasks on packages that may affect their delivered software.

5.2 Towards Better Tool Support

SCA tools are constantly used by software projects to control the risk related to software dependencies, such as vulnerabilities, and compliance to open source licenses[10]. To understand the support current SCA tools provide to production dependencies, we investigate four popular tools: npm audit, Snyk, Dependabot, and OSWAP Dependency-Check. We analyze the documentation of the SCA tools, as well as apply them to some of our studied projects to assess their capabilities and limitations.

We present in Table 7 an overview of the features related to dependency scope and usage from four popular SCA tools. All SCA tools we assess cover all dependencies of a software project, including both direct and transitive dependencies. Given a project may have thousands of installed dependencies, we now dive into the filtering capabilities of the tools. We note that only npm audit and Snyk [12] provide ways of filtering security alerts based on whether the vulnerability affects runtime/development dependencies or direct/transitive dependencies. The filtering of runtime/development dependencies is based on project configuration (e.g., package.json file), thus, it is subject to limitations when it comes identifying production versus non-production dependencies. Neither Dependabot nor OSWAP Dependency-Check allow users to filter security alerts based on the scope or depth of their dependencies.

It is worth noting that none of the tools provide a way to differentiate between production versus non-production dependencies. There is no support, for instance, to input source map files in the tools, to help filter out vulnerabilities that concern non-production dependencies. We believe adding source map support to SCA tools would offer developers better insight on their production bundle without relying so much on the dependency configurations that have shown to be inconsistent across different projects.

Finally, we find that none of the tools provide a way to locate where the vulnerable dependency is used in code (column “Locate dep code”). Developers have to rely on their own set of static/dynamic analysis tools to know exactly where the vulnerable dependency is used in the codebase. We believe static analysis tools would benefit from using the features provided by module bundlers that scan the code for import statements to provide the path to the source file in which a dependency is imported and used.

6 RELATED WORK

6.1 Dependency Studies

Package ecosystems and the presence of vulnerable dependencies have been studied in the literature [13, 21, 26, 27]. Hejderup et al. report that one-third of the npm packages use vulnerable dependencies [24]. Similar to our study, the authors suggest context of usage of a package to be a possible reason for not fixing the vulnerable dependencies. Abdalkareem et al. conduct an empirical analysis on security vulnerabilities in Python packages [13]. They find that the number of vulnerabilities in the PyPi ecosystem increases over time and that it takes, on median, more than 3 years to get discovered, regardless of their severity. They emphasize on the need for more effective process to detect vulnerabilities in open source packages since both npm and PyPi allows to publish a package release to the registry with no security checks. Lauinger et al. conduct the first large scale of JavaScript open source projects and investigate the relationship between outdated dependencies and dependencies with known vulnerabilities [29]. They report that transitive dependencies are more likely to be vulnerable since developers may not be aware of them and have less control over them, which further corroborates with our findings of RQ3. Similarly, Williams et al. report that 26% of open source Maven packages have known vulnerabilities and refer to a lack of meaningful controls of the components used in the proprietary projects as a partial explanation to this high number of vulnerable dependencies [41].

These prior studies focus on the presence of known vulnerabilities in popular package ecosystems and the reason why the number of vulnerable dependencies is so high. However, their analysis does not consider the scope of dependencies (i.e., they do not distinguish production and non-production dependencies). As a result, the studied vulnerable dependencies may not be exploitable. Zapata et al. investigate vulnerable dependency migrations of npm packages and evaluate the impact of a vulnerability in the ws package on 60 JavaScript projects using the vulnerable version of the package [42]. The authors find that up to 73.3% of the dependent applications were safe from the vulnerability since they did not actually used the vulnerable code. The study also highlights that it is not trivial to map vulnerable code to client usage for JavaScript, which further corroborates with our findings in RQ1.

6.2 Detecting Vulnerable Dependencies

Alfadel et al. study the use of Dependabot security pull requests in 2,904 JavaScript open source GitHub projects [14]. Results show that the vast majority (65.42%) of the security-related pull requests are often merged within a day and that the severity of the vulnerable dependency or potential risk for breaking changes are not associated with the merge time. Ponta et al. propose a pragmatic approach to facilitate the assessment of vulnerable dependencies in open source libraries by mapping patch-based changes of vulnerabilities onto the affected components of the application [35]. Sejfia et al. present Amalfi, a machine-learning based approach for automatically detecting potentially malicious packages [37]. The authors evaluate their approach on 96,287 npm package versions published over the course of one week and identify 95 previously unknown vulnerabilities. Pashchenko et al. propose Vuln4Real, a methodology that addresses the over-inflation problem of academic and industrial

Table 7: Vulnerable dependency (VD) characteristics based metrics reported by current tools and support to locate vulnerable code (VC) in JavaScript projects.

Tool	Cover all dependencies	Filter Security Alerts by					Locate dep code
		Runtime	Development	Direct	Transitive	In production	
npm audit	Yes	Yes	Yes	Yes	Yes	No	No
Snyk	Yes	Yes	Yes	Yes	Yes	No	No
Dependabot	Yes	No	No	No	No	No	No
OSWAP Dependency-Check	Yes	No	No	No	No	No	No

approach for reporting vulnerable dependencies in free and open source software (FOSS) [33]. Vuln4Real extends state-of-the-art approaches to analyzing dependencies by filtering development-only dependencies, grouping dependencies by project, and assessing dead dependencies. Their evaluation of Vuln4Real shows that the methodology significantly reduces the number of false alerts for code in production (i.e., dependencies wrongly flagged as vulnerable). Pashchenko et al’s work is the closest to ours since it considers similar aspects in relationship to the relevance of vulnerable dependencies: exploitability and dependency scope. Our study touches on another aspect that is not discussed in Vuln4Real and that is the context in which a dependency is used versus how it is configured. Our paper shows that there is a discrepancy between the configuration of dependencies and its usage, and that this discrepancy may affect the exploitability of a vulnerability (i.e., its relevance to the application).

Imtiaz et al. [25] present an in-depth case study by comparing the analysis reports of 9 SCA tools on OpenMRS, a large web application composed of Maven and npm projects. The study shows that the tools vary in their vulnerability reporting and that the count of vulnerable dependencies reported for npm projects ranges from 32 to 239. From the 9 studied SCA tools, 4 freely available tools could be applied to npm projects: OWASP Dependency-Check, Snyk, Dependabot, and npm audit. The results show that all 4 tools detect vulnerable dependencies across all scopes and depths and that reported vulnerabilities are mostly introduced through transitive dependencies, except for Dependabot. While the authors of this paper report on the coverage capabilities of SCA tools, our study mainly focuses on the data that is shown to the user. For example, npm audit covers dependencies of all scope when reporting for vulnerabilities, but it is the user’s responsibility to filter the vulnerable dependencies by scope (production or development). That is, SCA tools don’t explicitly report on the scope of vulnerable dependencies, and when it is done manually by users, this analysis depends on the project’s dependency configurations rather than dependency usage.

7 THREATS TO VALIDITY

Threats to internal validity considers the experimenter’s bias and errors. Our method of analysis relies on building software projects with their configured module bundler to identify production dependencies, and errors in this process may introduce false positives/negatives in our analysis. We mitigate this threat by 1) only selecting projects that already use module bundlers to minimize any intervention that could introduce bugs in the process, 2) building each project manually by following the projects documentation, 3) manually inspecting the built artifacts (e.g., installed dependencies, source map files), and 4) removing 55 projects that

showed evidence of failed builds (e.g., errors, empty source map files). To further confirm the validity of this process, we also sampled 7 projects from our dataset and asked contributors to validate our results, by checking the accuracy of the yielded production dependencies. We received responses from 4 projects and contributors confirmed the yielded classifications, helping us validate the soundness of our methodology.

Threats to external validity considers the generalizability of the findings. We purposefully select projects that already use module bundlers which could limit the type of project our findings generalize. First, module bundlers tend to be used primarily by projects that want to minimize their production dependencies, such as client-side packages such as web-applications and libraries. In fact, our finding that development dependencies are shipped to production are unlikely to occur in projects that do not use module bundlers. Second, our dataset is strictly composed of open source JavaScript projects, thus, our results may differ if a study is performed on proprietary projects or projects written in other languages.

8 CONCLUSION AND FUTURE WORK

This research investigates projects dependencies that are released to production and their impact on security and dependency management. We conducted our study on 100 npm projects, one of the largest and fastest growing software ecosystems. Our results showed that production dependencies are rare among the installed dependencies of a project, but are difficult to identify. Commonly held assumptions of dependency management do not hold in practice and context is more important in determining the scope of a dependency as opposed to its configuration. Furthermore, we evaluate how often security alerts are reported for production dependencies, and found that none of the vulnerability reports are emitted for dependencies released to production. Rather, the majority of the alerts are emitted for development, transitive dependencies which has two main implications: 1) not every vulnerability is a threat to the software in production, and 2) vulnerabilities can be introduced transitively regardless of their scope, which further motivates the need for SCA tools to provide such an analysis.

Our paper outlines directions for future work. Using module bundlers as a way to identify production dependencies may augment current SCA tools to provide better insights on the scope of their dependencies within their project’s context and usage. Consequently, module bundlers or similar tools, may benefit far more than just client-side applications and should be part of the build process of projects that extensively rely on open source code.

REFERENCES

- [1] 2019. 2019 State of the Software Supply Chain. https://www.sonatype.com/hubfs/SSC/2019%20SSC/SO_N_SSSC-Report-2019_jun16-DRAFT.pdf
- [2] 2019. Eight Key Findings Illustrating How to Make Open Source Work Even Better for Developers. <https://cdn2.hubspot.net/hubfs/4008838/Resources/The-2019-Tidelift-managed-open-source-survey-results.pdf>
- [3] 2019. webpack. <https://webpack.js.org/>
- [4] 2020. Do "dependencies" and "devDependencies" matter when using Webpack? <https://jsramblings.com/do-dependencies-devdependencies-matter-when-using-webpack/>
- [5] 2020. npm-deps-parser. <https://github.com/nVisium/npm-deps-parser>
- [6] 2020. Securing the World's Software. <https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf>
- [7] 2021. Create react app. <https://create-react-app.dev/>
- [8] 2021. Help, 'npm audit' says I have a vulnerability in react-scripts! · Issue #11174 · facebook/create-react-app. <https://github.com/facebook/create-react-app/issues/11174>
- [9] 2021. rollup.js. <https://rollups.org/guide/en/>
- [10] 2022. The Complete Guide to Software Composition Analysis - FOSSA. <https://fossa.com/complete-guide-software-composition-analysis>
- [11] 2022. GitHub Advisory Database. <https://github.com/advisories>
- [12] 2022. Snyk | Developer security | Develop fast. Stay secure. <https://snyk.io/>
- [13] Rabe Abdalkareem, Vicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering* 25 (01 2020), 1168–1204. <https://doi.org/10.1007/s10664-019-09792-9>
- [14] Mahmoud Alfadel, Diego Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. <https://doi.org/10.1109/MSR52588.2021.00037>
- [15] Md Atique, Reza Chowdhury, Rabe Abdalkareem, and Emad Shihab. 2019. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *Journal of IEEE Transactions on Software Engineering* 01 (2019). http://das.ensc.concordia.ca/uploads/atique_tse2021.pdf
- [16] Victor R. Basili, Lionel C. Briand, and Walcécio L. Melo. 1996. How reuse influences productivity in object-oriented systems. *Commun. ACM* 39 (10 1996), 104–116. <https://doi.org/10.1145/236156.236184>
- [17] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes. *ACM Transactions on Software Engineering and Methodology* 30 (07 2021), 1–56. <https://doi.org/10.1145/3447245>
- [18] Xiaowei Chen, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Xin Xia. 2021. Helping or not Helping? Why and How Trivial Packages Impact the npm Ecosystem. *Empirical Software Engineering* 26 (03 2021). <https://doi.org/10.1007/s10664-020-09904-w>
- [19] Diego Elias Costa, Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2021. Breaking Type-Safety in Go: An Empirical Study on the Usage of the unsafe Package. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/TSE.2021.3057720>
- [20] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 109–118. <https://doi.org/10.1109/ICSE.2015.140>
- [21] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering* 24 (02 2019). <https://doi.org/10.1007/s10664-017-9589-y>
- [22] Josh Fruhlinger. 2020. Equifax data breach FAQ: What happened, who was affected, what was the impact? <https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-was-affected-what-was-the-impact.html>
- [23] Emitza Guzman, David Azócar, and Yang Li. 2014. Sentiment Analysis of Commit Comments in GitHub: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) (MSR 2014). Association for Computing Machinery, New York, NY, USA, 352–355. <https://doi.org/10.1145/2597073.2597118>
- [24] J. I. Hejderup. 2015. In Dependencies We Trust: How vulnerable are dependencies in software modules? *repository.tudelft.nl* (2015). <https://repository.tudelft.nl/islandora/object/uuid:3a15293b-16f6-4e9d-b6a2-f02cd52f1a9e?collection=education>
- [25] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (10 2021). <https://doi.org/10.1145/3475716.3475769>
- [26] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/tse.2021.3106247>
- [27] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) (MSR '17). IEEE Press, 102–112. <https://doi.org/10.1109/MSR.2017.55>
- [28] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (may 2017), 384–417. <https://doi.org/10.1007/s10664-017-9521-5>
- [29] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 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*. Internet Society. <https://doi.org/10.14722/ndss.2017.23414>
- [30] Emerson Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward Smith, and Matt Jorde. 2019. What Predicts Software Developers' Productivity? *IEEE Transactions on Software Engineering* (2019), 1–1. <https://doi.org/10.1109/tse.2019.2900308>
- [31] Stack Overflow. [n.d.]. Stack Overflow Developer Survey 2021. <https://insights.stackoverflow.com/survey/2021>
- [32] Ivan Pashchenko, Henrik Plate, Serena Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: counting those that matter. 1–10. <https://doi.org/10.1145/3239235.3268920>
- [33] Ivan Pashchenko, Henrik Plate, Serena Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* PP (09 2020), 1–1. <https://doi.org/10.1109/TSE.2020.3025443>
- [34] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. *A Qualitative Study of Dependency Management and Its Security Implications*. Association for Computing Machinery, New York, NY, USA, 1513–1531. <https://doi.org/10.1145/3372297.3417232>
- [35] Henrik Plate, Serena Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. 411–420. <https://doi.org/10.1109/ICSM.2015.7332492>
- [36] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 155–165. <https://doi.org/10.1145/2635868.2635922>
- [37] Adriana Sejfa and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. *arXiv preprint arXiv:2202.13953* (2022).
- [38] unisil. 2021. Source Map Parser. <https://github.com/unisil/source-map-parser>
- [39] Haroen Viaene. 2021. feat(dependencies): update algoliasearch-helper. <https://github.com/algolia/instantsearch.js/pull/4936>. (Accessed on 05/04/2022).
- [40] Stefan Wagner and Emerson Murphy-Hill. 2019. *Factors That Influence Productivity: A Checklist*. 69–84. https://doi.org/10.1007/978-1-4842-4221-6_8
- [41] Jeff Williams and Arshan Dabirsiaghi. 2012. The unfortunate reality of insecure libraries. *Asp. Secur. Inc* (2012), 1–26.
- [42] Rodrigo Zapata, Raula Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. 559–563. <https://doi.org/10.1109/ICSME.2018.00067>