On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks

**Ahmed Zerouali · Tom Mens ·
Alexandre Decan · Coen De Roover**

Abstract The increasing interest in open source software has led to the emergence of large language-specific package distributions of reusable software libraries, such as npm and RubyGems. These software packages can be subject to vulnerabilities that may expose dependent packages through explicitly declared dependencies. Using Snyk's vulnerability database, this article empirically studies vulnerabilities affecting npm and RubyGems packages. We analyse how and when these vulnerabilities are disclosed and fixed, and how their prevalence changes over time. We also analyse how vulnerable packages expose their direct and indirect dependents to vulnerabilities. We distinguish between two types of dependents: packages distributed via the package manager, and external GitHub projects depending on npm packages. We observe that the number of vulnerabilities in npm is increasing and being disclosed faster than vulnerabilities in RubyGems. For both package distributions, the time required to disclose vulnerabilities is increasing over time. Vulnerabilities in npm packages affect a median of 30 package releases, while this is 59 releases in RubyGems packages. A large proportion of external GitHub projects is exposed to vulnerabilities coming from direct or indirect dependencies. 33% and 40% of dependency vulnerabilities to which projects and packages are exposed, respectively, have their fixes in more recent releases within the same major release range of the used dependency. Our findings reveal that more effort is needed to better secure open source package distributions.

Keywords: security vulnerabilities, npm, RubyGems, vulnerable packages

A. Zerouali and C. De Roover
Vrije Universiteit Brussel, Brussels, Belgium
E-mail: ahmed.zerouali@vub.be, coen.de.roover@vub.be

T. Mens and A. Decan
Université de Mons, Mons, Belgium
E-mail: tom.mens@umons.ac.be, alexandre.decan@umons.ac.be

1 Introduction

Due to the increasing popularity and use of open source software (OSS), a large number of OSS ecosystems have emerged, containing huge collections of interdependent software packages. These ecosystems are usually supported by large communities of contributors, and can be considered as software supply chains formed by upstream transitive dependencies of packages and their downstream dependents. Typical examples of such ecosystems are package distributions for specific programming languages (e.g., npm for JavaScript, PyPI for Python, RubyGems for Ruby), totalling millions of interdependent reusable libraries maintained by hundreds of thousands of developers, and used by millions of software projects in a daily basis.

Given the sheer size of package distributions, combined with their open source nature, many packages are being affected by known or unknown vulnerabilities. Since package distributions are known to form huge and tightly interconnected dependency networks [1], a single vulnerable package may potentially expose a considerable fraction of the package dependency network to its vulnerabilities [2]. This exposure does not even stop at the boundaries of the package distribution, since dependent external software projects may also become exposed to these vulnerabilities [3]. According to a study carried out by Snyk [4], one of the leading companies in analysing and detecting software vulnerabilities in Node.Js and Ruby packages, 77% of the 430k websites run at least one front-end package with a known security vulnerability in place.

This exposure of software to vulnerabilities in its dependencies is considered as one of the OWASP top 10 application security risks [5]. There are many examples of such cases. For example, in November 2018 the widely used npm package `event-stream` was found to depend on a malicious package named `flatmap-stream` [1] containing a Bitcoin-siphoning malware. The `event-stream` package is very popular getting roughly two million downloads per week. Just because it was used as a dependency in `event-stream`, the malicious `flatmap-stream` package was downloaded millions of times since its inclusion in September 2018 and until its discovery and removal.

The main objective of this paper is therefore to empirically analyse and quantify the impact of vulnerabilities in open source packages, on transitively dependent packages that are shared through the same package distribution as well as on external projects distributed via GitHub that rely on such packages. We conduct our study on two different package distributions, npm and RubyGems. All along our research questions, we compare between the results found in these two package distributions. More specifically, we answer the following research questions:

– RQ_0: **How prevalent are disclosed vulnerabilities in npm and RubyGems packages?** This preliminary research question explores the dataset of vulnerabilities extracted from Snyk's database and provides insights about their characteristics and evolution over time.

---

[1] `https://github.com/dominictarr/event-stream/issues/116`

- RQ_1: **How much time elapses until a vulnerability is disclosed?** By answering this question for both npm and RubyGems, security researchers in these ecosystems will be able to assess how quick they are in finding and disclosing vulnerabilities. Users of these two ecosystems will be able to know which community has active security researchers which will eventually help them to assess their trust on the third-party packages they depend on.
- RQ_2: **For how long do packages remain affected by disclosed vulnerabilities?** Package dependents will gain insights about the number of dependency releases they should expect to be affected by a newly disclosed vulnerability and how many more releases are going to be affected by the same vulnerability even after its disclosure. The answer to this question will also help dependents to know to which type of package releases (i.e., major, minor or patch) they should update to have their dependency vulnerabilities fixed.
- RQ_3: **To what extent are dependents exposed to their vulnerable dependencies?** Vulnerable dependencies can expose their dependents to vulnerable code that might lead to security breaches. We will identify all direct and indirect vulnerable dependencies that are exposing packages and external GitHub projects, and characterize their vulnerabilities. In this and the next two research questions we study dependents as if they were deployed on 12 January 2020 (i.e., the snapshot of the vulnerability dataset).
- RQ_4: **How are vulnerabilities spread in the dependency tree?** The answer to this question will inform us how deep in the dependency tree we can find vulnerabilities to which packages and external GitHub projects are exposed. This helps us to quantify the transitive impact that vulnerable packages may have on their transitive dependents.
- RQ_5: **Do exposed dependents upgrade their vulnerable dependencies when a vulnerability fix is released?** We will quantify how much dependent packages and dependent external GitHub projects would benefit from updating their dependencies for which there is a known fix available. We will also report on the number of vulnerable dependencies that could be reduced by only doing backward compatible updates.
- RQ_6: **To what extent are dependents exposed to their vulnerable dependencies at their release time?** We will identify the dependencies that were only affected by vulnerabilities disclosed before each dependent's release time. Answering this research question will help us to assess whether developers are careful about incorporating dependencies with already disclosed vulnerabilities.

The remainder of this article is structured as follows: Section 2 discusses related work and highlights the differences to previous studies. Section 3 explains the research method and the data extraction process, and presents a preliminary analysis of the selected dataset. Section 4 empirically studies the research questions for the npm and RubyGems package distributions. Section 5

highlights the novel contributions, discusses our findings, and outlines possible directions for future work. Section 6 discusses the threats to the validity of this work and Section 7 concludes.

2 Related work

2.1 Terminology

This section introduces the terminology used throughout this article. All main terms are highlighted in **boldface**.

Package distributions (such as npm and RubyGems) are collections of (typically open source) software **packages**, distributed through some package registry. Each of these packages has one or more **releases**. New releases of a package are called **package updates**.

Each package release is denoted by a unique **version number**. The version number reflects the sequential order of all releases of a package. **Semantic versioning**, hereafter abbreviated as *semver*[2], proposes a multi-component version numbering scheme major.minor.patch[-tag] to specify the type of changes that have been made in a package update. Backward incompatible changes require an increment of the **major** version component, important backward compatible changes (e.g., adding new functionality that does not affect existing dependents) require an increment of the **minor** component, and backward compatible bug fixes require an increment of the **patch** component.

The main purpose of package distributions is to facilitate software reuse. To do so, a package release R can explicitly declare a **dependency relation** to another package P. R will be called a **(direct) dependent** of P, while P will be called a **(direct) dependency** of R. Dependency relations come with **dependency constraints** that specify which releases of the dependency P are allowed to be selected for installation when R is installed. Such constraints express a **version range**. For example, constraint `<2.0.0` defines the version range `[0.0.0, 2.0.0)`, signifying that *any* release below version 2.0.0 of the dependency is allowed to be installed. The highest available version within this range will be selected for installation by the package manager. Combining *semver* with dependency constraints enables maintainers of dependents to restrict the version range of dependencies to those releases that are expected to be backward compatible [6]. For exemple, a dependency relation in an npm release R could express a constraint `^1.2.3` to allow the version range `[1.2.3,2.0.0)` of backward compatible releases. In RubyGems, dependency constraint `~>1.2` would allow the version range `[1.2.0,2.0.0)` of backward compatible releases.

The collection of all package releases and their dependencies in a package distribution forms a **package dependency network**. A release R is an **indirect dependent** of another release D if there is a chain of length 2 or longer

---

[2] See `https://semver.org`

between them in the dependency network. Conversely D will be called an **indirect dependency** of R. We will refer to the union of direct and indirect dependents (respectively, dependencies) of a release as **transitive dependents** (respectively, dependencies) of that release.

In the context of this paper and more specifically in RQ_3 to RQ_6, we only study the vulnerabilities coming from dependencies used in the latest release of each package in npm and RubyGems. Because of this, we will occasionally use the term package to refer to the latest release of a dependent package. By abuse of terminology we declare a *package* to be a (direct/indirect) dependent of P if its latest available *release* depends (directly or indirectly) on P.

Not only *packages* can depend on other packages within a package dependency network, but the same is true for *external projects* that are developed and/or distributed outside of the package distribution (e.g., on GitHub). By extension of the term dependent package, we use the term **dependent project** to refer to a GitHub repository containing an external software project that (directly or indirectly) depends on one of the packages of the considered package distribution. For example, the project Atom [3] is a dependent of npm package mocha [4]. Similarly, Discourse [5] is a dependent project of RubyGems package json [6].

A **vulnerability** is a known reported security threat that affects some releases of some packages in the package distribution. The packages will be called **vulnerable packages** and their affected releases will be called **vulnerable releases**. The package's vulnerability is **fixed** as soon as a package update that is no longer affected by the vulnerability becomes available. A **vulnerable dependency** is a vulnerable release that is used as a dependency by another package or project (directly or indirectly). Vulnerable dependencies can expose their dependents to the vulnerability. We refer to those as **exposed dependents**. We can distinguish between *directly exposed* dependents (if a direct dependency is vulnerable) and *indirectly exposed* dependents (if an indirect dependency is vulnerable). To distinguish between dependent releases and dependent projects that may be exposed, we use the terms **exposed package** (release) and **exposed project**, respectively.

2.2 Package dependency networks

Software dependency management and package dependency networks have been subject to many research studies for different software ecosystems. Wittern et al. [7] examined the npm ecosystem in an extensive study that covers package descriptions, the dependencies among them, download metrics, and the use of npm packages in publicly available repositories. One of their findings is that the number of npm packages and their updates is growing superlinearly.

---

[3] `https://github.com/atom/atom/blob/master/package.json`

[4] `https://www.npmjs.com/package/mocha`

[5] `https://github.com/discourse/discourse/blob/master/Gemfile`

[6] `https://rubygems.org/gems/json`

They also observed that packages are increasingly connected through dependencies. More than 80% of npm packages have at least one direct dependency. Kikas et al. [8] analysed the dependency network structure and evolution of the JavaScript, Ruby, and Rust package distributions. One of their findings is that the number of transitive dependencies for JavaScript has grown by 60% in 2016. They also found that the negative consequences of removing a popular package (e.g., the left-pad incident [7]) are increasing. In a more extensive study, Decan et al. [9] empirically compared the impact of dependency issues in the npm, CRAN and RubyGems package distributions. A follow-up study [1] expanded this comparison with four more distributions, namely CPAN, Packagist, Cargo and NuGet. They observed important differences between the considered ecosystems that are related to ecosystem specific factors. Similarly, Bogart et al. [10] performed multiple case studies of three software ecosystems with different tooling and philosophies toward change (Eclipse, CRAN, and npm), to understand how developers make decisions about change and change-related costs and what practices, tooling, and policies are used. They found that the three ecosystems differ significantly in their practices, policies and community values. Gonzalez-Barahona et al. [11–14] introduced the notion of technical lag to quantify the degree of outdatedness of packages and dependencies, along different dimensions, including time lag, version lag and vulnerability lag.

2.3 Security vulnerabilities

Software vulnerabilities are discovered on a daily basis, and need to be identified and fixed as soon as possible. This explains the many studies conducted by software engineering researchers on the matter (e.g., [15–19]). Several researchers observed that outdated dependencies are a potential source of security vulnerabilities.

Cox et al. [20] analysed 75 Java projects that manage their dependencies through Maven. They observed that projects using outdated dependencies were four times more likely to have security issues and backward incompatibilities than systems that were up-to-date. Gkortzis et al. [21] studied the relationship between software reuse and security threats by empirically investigating 1,244 open-source Java projects to explore the distribution of vulnerabilities between the code created by developers and the code reused through dependencies. Based on a static analysis of the source code, they observed that large projects are associated with a higher number of potential vulnerabilities. Additionally, they found that the number of dependencies in a project is strongly correlated to its number of vulnerabilities. Massacci et al. [22] investigated whether leveraging on FOSS Maven-based Java libraries is a potential source of security vulnerabilities. They found that small and medium libraries have disproportionately more leverage on FOSS dependencies in comparison

---

[7] https://github.com/left-pad/left-pad/issues/4

to large libraries. They also found that libraries with higher leverage have 1.6 higher odds of being vulnerable in comparison to the libraries with lower leverage.

Ponta et al. [23] presented a code-centric and usage-based approach to detecting and assessing OSS vulnerabilities, and to determining their reachability through direct and transitive dependencies of Java applications. Their combination of static and dynamic analysis improves upon the state of the art which commonly considers dependency meta-data only without verifying to which dependents the vulnerability actually propagates. The Eclipse Steady tool instantiates the approach, and has been shown to report fewer false positives (i.e., vulnerabilities that do not really propagate to dependencies) than earlier tools. In a similar vein, Zapata et al. [24] carried out an empirical study that analysed vulnerable dependency migrations at the function level for 60 JavaScript packages. They provided evidence that many outdated projects are free of vulnerabilities as they do not really rely on the functionality affected by the vulnerability. Because of this, the authors claim that security vulnerability analysis at package dependency level is likely to be an overestimation.

Decan et al. [2] conducted an empirical analysis of 399 vulnerabilities reported in the npm package dependency network containing over 610k JavaScript packages in 2017. They analysed how and when these vulnerabilities are disclosed and to which extent this affects directly dependent packages. They did not consider the effect of vulnerabilities on transitive dependents, nor did they study the impact on external GitHub projects depending on npm packages. They observed that it often takes a long time before an introduced vulnerability is disclosed. A non-negligible proportion of vulnerabilities (15%) are considered to be more risky because they are fixed only after public announcement of the vulnerability, or not fixed at all. They also found that the presence of package dependency constraints plays an important role in vulnerabilities not being fixed, mainly because the imposed constraints prevent fixes from being installed.

Zimmermann et al. [25] studied 609 vulnerabilities in npm packages, providing evidence that this ecosystem suffers from single points of failure, i.e., a very small number of maintainer accounts could be used to inject malicious code into the majority of all packages (e.g., the event-stream incident [8]). This problem increases with time, and unmaintained packages threaten large code bases, as the lack of maintenance causes many packages to depend on vulnerable code, even years after a vulnerability has become public. They studied the transitive impact of vulnerable dependencies, as well as the problems related to lack of maintenance of vulnerable packages.

Alfadel et al. [26] carried out a study on 550 vulnerability reports affecting 252 Python packages from PyPI. They found that the number of vulnerabilities disclosed in Python packages increases over time, and some take more than 3 years to be disclosed. Meneely et al. [27] inspected 68 vulnerabilities in the Apache HTTP server and traced them back to the first commits that

---

[8] https://www.theregister.com/2018/11/26/npm_repo_bitcoin_stealer/

contributed to the vulnerable code. They manually found 124 Vulnerability-Contributing Commits (VCCs). After analyzing these VCCs, they found that VCCs have more than twice as much code churn on average than non-VCCs. They also observed that commits authored by new developers have more chances to be VCCs than other commits.

Prana et al. [28] analysed vulnerabilities in open-source libraries used by 450 software projects written in Java, Python, and Ruby. Using an industrial software composition analysis tool, they scanned versions of the sample projects after each commit. They found that project activity level, popularity, and developer experience do not translate into better or worse handling of dependency vulnerabilities. As a recommendation to software developers, they highlighted the importance of managing the number of dependencies and of performing timely updates.

Pashchenko et al. [17] studied the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in OSS software. After inspecting 200 Java libraries they found that 20% of their dependencies affected by a known vulnerability are not deployed, and therefore, they do not represent a danger to the analyzed library because they cannot be exploited in practice. They also found that 81% of vulnerable direct dependencies may be fixed by simply updating to a new version. In our article, we follow the procedure recommended by Pashchenko et al. [17] by only focusing on runtime dependencies which are essential for deployment.

2.4 Novelty of our contribution

The empirical study proposed in this article expands upon previous work in different ways. We conduct a quantitative comparison of vulnerabilities in both the npm and RubyGems package dependency networks based on a more recent dataset of packages and their vulnerabilities (2020). We study the impact of a large set of 2,786 vulnerabilities, of which 2,118 for npm and 668 for RubyGems while grouping them by their severity levels. We also consider dependencies of external GitHub projects on vulnerable npm and RubyGems packages. For the latter, we are the first to study the prevalence, disclosure and fixing time of their vulnerabilities. When studying the impact of vulnerable packages on their dependents, we do not only focus on direct dependencies, but also consider the indirect ones. For those indirect dependencies we study the evolution and spread of vulnerable indirect dependencies at different levels in the dependency tree. Finally, we are the first to compare the vulnerability of packages distributed via package distributions with the vulnerability of GitHub projects that use these packages. Such a comparison is important since packages are supposed to be reused as libraries, their maintainers are supposed to be more careful than developers of external projects that just depend on these reusable libraries and that are much less likely to have other projects depending on them. We are also the first to report results on how vulnerability disclosure time duration is evolving over time.

3 Dataset

This paper analyzes the Common Vulnerabilities and Exposures (CVE[9]) affecting the npm and RubyGems distributions of reusable software packages. Both package distributions are well-established and mature (RubyGems was created in 2004 and npm in 2010) and both recommend *semver* for package releases [6]. Due to an important difference in popularity of the targeted programming language (JavaScript and Ruby, respectively), the number of packages distributed through npm is an order of magnitude higher than the number of packages distributed through RubyGems. We focus on these package distributions because they have a community that is actively looking for and reporting vulnerabilities. The vulnerabilities in both distributions are reported and tracked by well-known Central Numbering Authorities (CNA).

3.1 Vulnerability dataset

To detect vulnerabilities in npm and RubyGems packages we rely on a database of vulnerability reports of third-party package vulnerabilities collected by the continuous security monitoring service Snyk [10]. We received a snapshot of this vulnerability database on 17 April 2020. This snapshot contained 2,874 vulnerability reports for the considered package distributions, of which 2,188 for npm and 686 for RubyGems. The higher number of reported vulnerabilities for npm can be explained by the higher number of packages contained in it.

Each vulnerability report contains information about the affected package, the range of affected releases, the severity of the vulnerability as reported by Snyk's security team, its origin (i.e., the package distribution), the date of disclosure, the date when it was published in the database, the first fixed version (if available), and the unique CVE identifier. Figure 1 shows an example of a vulnerability report of the popular RubyGems package `rest-client` [11]. Vulnerability reports for npm packages contain similar information.

Vulnerability name:	Malicious Package
Severity:	critical
Affected package:	rest-client
Affected versions:	>=1.6.10, <1.7.0.rc1
Package manager:	RubyGems
Disclosure date:	2019-08-19
Publication date:	2019-08-20
Version with the first fix:	1.7.0.rc1
CVE identifier:	CVE-2019-15224

Fig. 1 Excerpt of vulnerability report for RubyGems package rest-client.

---

[9] `cve.mitre.org`

[10] `https://snyk.io/vuln`

[11] `https://snyk.io/vuln/SNYK-RUBY-RESTCLIENT-459900`
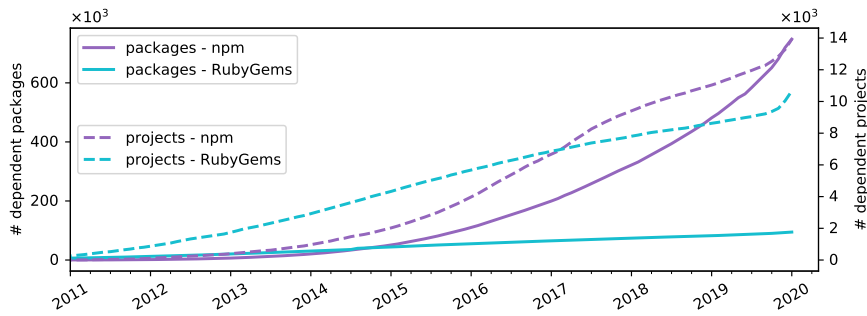
3.2 Dependency dataset of packages and external projects

Using version 1.6.0 of the *libraries.io* Open Source Repository and Dependency
Dataset [29] that was released on 12 January 2020, we identified all *package
releases* in the npm and RubyGems package distributions. As this dataset
was released three months before the snapshot date of the Snyk vulnerability
database (17 April 2020), it does not contain releases that are affected by vul-
nerabilities disclosed after 12 January 2020. We therefore ignore these releases
in our analysis.

   We also identified all *external projects* hosted on GitHub and referenced in
libraries.io dataset as depending on npm or RubyGems packages. Repositories
of these projects do not correspond to the development history of any of the
considered packages, and are not forked from any other repository. This way
we ensure that packages and external projects are mutually exclusive, and we
avoid considering the same projects multiple times in the analysis.

   A manual analysis revealed that *libraries.io* is not always accurate about
the dependencies used by external projects. More specifically, we occasionally
observed a mix between run-time and development dependencies for projects
that use npm, i.e., dependencies that are declared as development dependencies
in the project's repository in GitHub are reported as run-time dependencies in
libraries.io. Because of this inaccuracy, we only relied on *libraries.io* to identify
the names of the most starred projects, i.e., those who received 90% of all stars
within the same ecosystem (npm or RubyGems). Overall, the selected GitHub
projects received 5.37M stars out of 5.96M. The minimum number of stars
found in the resulting dataset was 62. Afterwards, to determine the npm and
RubyGems package dependencies for these projects, we extracted and analysed
their *package.json* and *Gemfile* from GitHub, which are the files in which the
dependency metadata is stored for the respective package distributions.

   As older package releases are more likely to be exposed to vulnerable pack-
ages than recent versions [20, 30], this might bias the analysis results. We
therefore decided to focus only on the latest available version of each consid-
ered package, and on the snapshot of the last commit before 12 January 2020
of each considered external GitHub project. We also decided to focus only on
packages and external projects with *run-time dependencies*, thereby ignoring
development and optional dependencies. Run-time dependencies are needed
to deploy and run the dependent in production, while development dependen-
cies are only needed while the dependent is being developed (e.g., for testing
dependencies). We ignore the latter in our study because they are unlikely to
affect the production environment [17].

   Based on all of the above, we selected the latest available releases in *li-
braries.io* of 842,697 packages (748,026 for npm and 94,671 for RubyGems)
and 24,593 external projects hosted on GitHub (13,930 using npm packages
and 10,663 using RubyGems packages) with run-time dependencies. Figure 2
shows the evolution over time of the cumulative number of packages (left y-
axis) and external projects (right y-axis) considered in this study, grouped by
package distribution.

Fig. 2 Evolution of the cumulative number of packages (straight lines, using the scale on the left y-axis) and external GitHub projects (dotted lines, using the scale on the right y-axis) for npm and RubyGems

Using the dependency constraint resolver proposed in [6], which supports several package distributions, we determined the appropriate version of packages to be installed for each dependent according to the constraints for its run-time dependencies. As some constraints may resolve to different versions at different points in time, we use the *libraries.io* snapshot date as the resolution date. This implies that we study vulnerabilities in packages and external projects as if they were installed or deployed on 12 January 2020.

Having determined the versions of all direct dependencies, we turn to the indirect ones. All considered npm packages have a total of 68,597,413 dependencies of which 3,638,361 are direct (i.e., 5.3%), while all RubyGems packages have 1,258,829 dependencies of which 224,959 are direct (i.e., 17.9%). The considered external projects have 2,814,544 npm dependencies of which 147,622 are direct (i.e., 5.2%), and 544,506 RubyGems dependencies of which 101,079 are direct (i.e., 18.6%). We observe that RubyGems packages and external projects have more than thrice as many direct dependencies as npm packages and external projects. This is in line with the observations made by Decan et al. [6]. Figure 3 shows the evolution over time of the cumulative number of direct and indirect dependencies for package latest releases and external projects considered in this study, grouped by package distribution.

4 Empirical Analysis

Using the datasets of Section 3, this section answers the research questions introduced in Section 1. For RQ_0 to RQ_2, we present statistical analyses based on the affected npm and RubyGems package releases according to the vulnerability dataset. For RQ_3 to RQ_6 we study the impact of affected releases on exposed packages and exposed projects that directly or indirectly depend on them.

As part of the statistical analyses, we use the non-parametric Mann-Whitney U test to compare various types of distributions without assuming them to
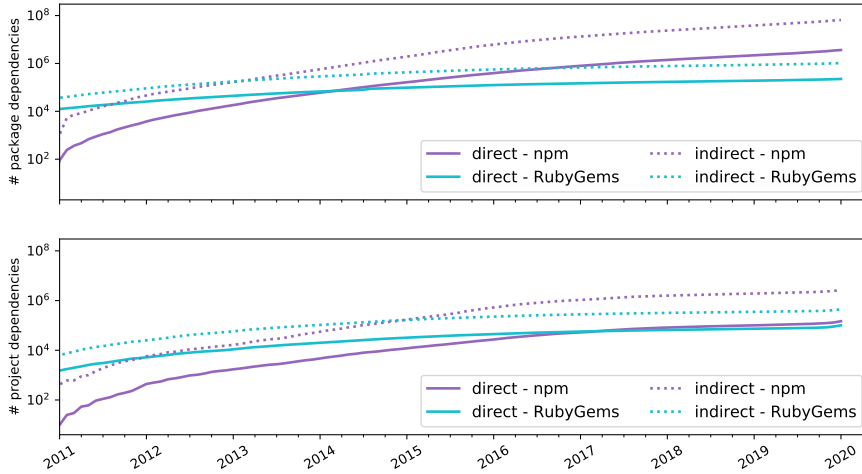
Fig. 3 Monthly evolution of the cumulative number of direct and indirect dependencies for packages (top figure) and external projects (bottom figure) for npm and RubyGems.

follow a normal distribution. The null hypothesis H_0 states that there is no difference between two distributions. We set a global confidence level of 95%, corresponding to a significance level of $\alpha = 0.05$. To achieve this overall confidence, the p-value of each individual test is compared against a lower α value, following a Bonferroni correction[12]. If the null hypothesis can be rejected, we report the effect size with Cliff's delta d, a non-parametric measure that quantifies the difference between two populations beyond the interpretation of p-values. Following the guidelines of Romano et al. [31], we interpret the effect size to be *negligible* if $|d| \in [0, 0.147[$, *small* if $|d| \in [0.147, 0.33[$, *moderate* if $|d| \in [0.33, 0.474[$ and *large* if $|d| \in [0.474, 1]$.

We use the technique of survival analysis [32] to estimate the probability that an event of interest will happen. Survival analysis creates a model estimating the survival rate of a population over time until the occurrence of an event, considering the fact that some subjects may leave the study, while for others the event of interest might not occur during the observation period. We rely on the non-parametric Kaplan-Meier statistic estimator commonly used to estimate survival functions.

All code used to carry out this analysis is available in a replication package [13].

---

[12] If n different tests are carried out over the same dataset, for each individual test one can only reject H_0 if $p < \frac{0.05}{n}$. In our case $n = 48$, i.e., $p < 0.001$.

[13] https://github.com/AhmedZerouali/vulnerability_analysis

4.1 RQ_0: How prevalent are disclosed vulnerabilities in npm and RubyGems packages?

This research question aims to characterise the vulnerability dataset of Section 3.1, its evolution over time as well as the number of package releases affected by these vulnerabilities. After the identification of package releases and dependencies in Section 3.2, we found that 88 vulnerabilities do not affect any package release that is used as a dependency. Therefore, our final dataset contains 2,786 vulnerabilities, of which 2,118 directly affecting npm packages and 668 directly affecting RubyGems packages.

Figure 4 depicts the number of considered vulnerabilities in this study, and the proportion of *low, medium, high* or *critical* severities for each package distribution. We observe that most of the vulnerabilities are of *medium* or *high* severity (76% for npm and 89% for RubyGems). We also observe that npm has nearly thrice as many *critical* vulnerabilities as RubyGems. Another difference is that npm has more *high* vulnerabilities than *medium* ones, while the inverse is true for RubyGems. The collected vulnerabilities affect 1,672 npm and 321 RubyGems packages. The oldest vulnerability in npm was disclosed in June 2011, whereas the oldest one in RubyGems was disclosed in August 2006. We also found that 1,175 (42%) of the vulnerabilities did not have any known fix, of which 1,058 from npm and 117 from RubyGems.

Finding the reasons behind the observed differences between both ecosystems in terms of number, severity, and types of vulnerabilities (see Table 2) can be hard. Each package distribution has its own tools, practices and policies [33] as well as differences in topological structure of the dependency network and size and growth of the ecosystem [1]. Both package distributions also focus on different programming languages. Figure 4 therefore does not normalize against the size of each ecosystem. The proportion of vulnerable packages is 0.22% for npm while it is 0.4% for RubyGems, even though npm is 8 times larger than RubyGems.
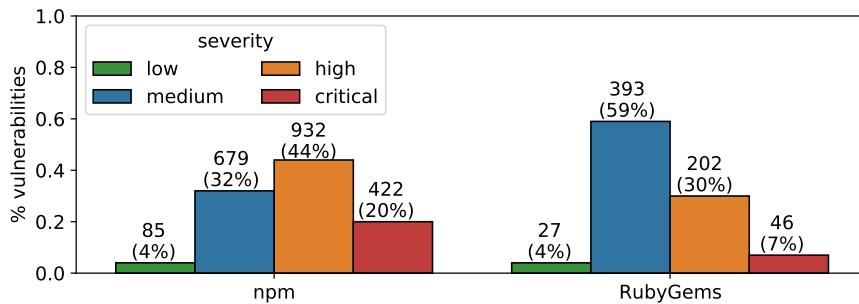
Fig. 4 Proportion and number of considered vulnerabilities affecting npm and RubyGems packages, grouped by severity.

Figure 5 depicts the evolution over time of the cumulative number of vulnerabilities (straight lines) grouped by severity, and their corresponding affected packages (dotted lines). The y-axis scale for npm is different from the one for RubyGems because more vulnerabilities have been reported for npm. The number of reported vulnerabilities and thereby affected packages increases for both npm and RubyGems over time. We also see that before 2017, the number of medium vulnerabilities was higher than the number of high vulnerabilities in npm. Since 2016 and until 2018, the number of high vulnerabilities started increasing following a different trend.

A similar trend can be observed for the critical vulnerabilities in 2019. This means that proportionally speaking, the severity of npm vulnerabilities tends to increase over time, while such an observation cannot be made for RubyGems.
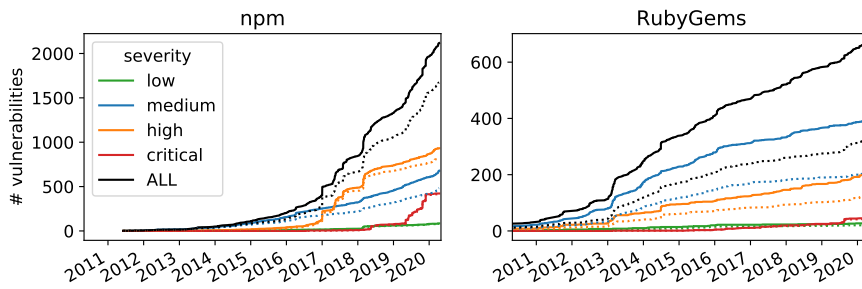
Fig. 5 Temporal evolution of the cumulative number of disclosed vulnerabilities (straight lines) and the corresponding number of affected packages (dotted lines) per severity.

Looking at *ALL* vulnerabilities we can see an exponential growth in the number of npm vulnerabilities, while RubyGems shows a linear growth rate. This is statistically confirmed by a regression analysis using linear and exponential growth models. The R^2 values reflecting the goodness of fit are summarized in Table 1. [14] For both vulnerabilities and affected packages, npm follows an exponential growth while RubyGems follows a linear one. The exponential growth of npm is in line with the exponential growth of its total number of packages [1]. Given that npm is by far the largest package distribution available[15], it is considerably more likely to contain reported vulnerabilities. npm's popularity may attract more malicious developers on the one hand, that want to exploit vulnerabilities contained in some of its packages, and on the other hand it may attract more security researchers that aim to find and report vulnerabilities before they can be exploited.

---

[14] $R^2 \in [0, 1]$ and the closer to 1 the better the model fits the data.
[15] According to *libraries.io*, in May 2021, npm contained 1.79M packages compared to "only" 173K packages in RubyGems.

Table 1 R^2-values of regression analysis on the evolution of the number of vulnerabilities and affected packages.

	npm		RubyGems	
	# vulns	# packages	# vulns	# packages
linear	0.85	0.84	<u>0.94</u>	<u>0.93</u>
exponential	<u>0.96</u>	<u>0.97</u>	0.90	0.90

For each vulnerability report in the vulnerability dataset, we identified the affected range of releases of the specified package in our package dataset. Here too we relied on the constraint parser proposed in [6]. While the vulnerability dataset corresponds to a relatively low number of vulnerable packages (see above), the majority of their releases is in fact affected: 67.4% (i.e., $43,330$ out of $64,236$) for npm and 63.8% (i.e., $11,488$ out of $17,987$) for RubyGems. 89.9% of these affected releases (comprising both npm and RubyGems) concern vulnerabilities of either *medium* severity (52.9%) or *high* severity (37% high). Regardless of their severity, 57.3% and 27.8% of all npm and RubyGems vulnerabilities are affecting more than 90% of the releases of their corresponding packages. Most of these vulnerabilities (85.6% and 62.9%, respectively) are open ones that do not have any fix. Focusing only on fixed vulnerabilities, this proportion decreases to 16.2% and 12.5% for npm and RubyGems vulnerabilities affecting more than 90% of their package releases, respectively. This means, for the clients of these packages, that they should be using the latest available releases, especially the clients of npm packages.

Table 2 shows the top ten vulnerability types affecting npm and RubyGems packages, with the number of vulnerabilities of each type grouped by severity. We observe that the most prevalent vulnerability type is *Malicious Package* and *Cross-site Scripting (XSS)* found in 19.3% of all npm vulnerability reports, and 17.4% of all RubyGems vulnerability reports. We observe that npm and RubyGems packages are exposed to similar vulnerabilities but with different occurrences. Some vulnerability types seem to affect JavaScript packages more than Ruby packages, e.g., *Malicious Package* and *Directory Traversal*.

> The number of reported vulnerabilities is increasing exponentially for npm and linearly for RubyGems. The relative proportion of high and critical vulnerabilities seems to increase over time. Two out of three releases of vulnerable packages are affected by at least one vulnerability for both ecosystems. The types of most prevalent vulnerabilities differ between npm and RubyGems.

4.2 RQ_1: How much time elapses until a vulnerability is disclosed?

Delayed fixing of security vulnerabilities puts software packages and their dependents at risk as it lengthens the window that hackers have to discover and exploit the vulnerability. Unknown vulnerabilities may linger and remain to be present in more recent releases of a vulnerable package, exposing the depen-

Table 2 Top ten vulnerability types affecting npm and RubyGems packages, with the number of vulnerabilities of each type grouped by severity (C = critical, H = high, M = medium, L = low).

npm vulnerability types	#vulns	C	H	M	L
Malicious Package	410	345	64	1	0
Directory Traversal	331	5	297	28	1
Cross-site Scripting	322	15	55	245	7
Resource Downloaded over Insecure Protocol	154	0	145	8	1
Regular Expression Denial of Service	138	0	57	50	31
Denial of Service	91	5	52	32	2
Prototype Pollution	77	1	38	36	2
Command Injection	57	0	20	34	3
Arbitrary Code Execution	44	11	24	7	2
Arbitrary Code Injection	36	7	9	20	0

RubyGems vulnerability types	#vulns	C	H	M	L
Cross-site Scripting	116	0	6	108	2
Denial of Service	54	1	18	34	1
Arbitrary Command Execution	47	2	29	16	0
Information Exposure	40	1	6	27	6
Arbitrary Code Execution	27	2	16	9	0
Man-in-the-Middle	26	0	3	22	1
Malicious Package	19	18	1	0	0
Cross-site Request Forgery	18	0	4	14	0
SQL Injection	18	1	11	6	0
Directory Traversal	16	0	5	10	1

dents of an entire release range for a substantial amount of time. For example, in January 2021, a vulnerability named "Baron Samedit" was discovered in the popular Linux package sudo [16]. This vulnerability allowed local users to gain root-level access and was introduced in the sudo code back in July 2011, effectively exposing all releases during the past decade and therefore millions of deployments.

RQ_1 aims to study how much time it takes until a vulnerability is discovered. Since it is not possible to accurately know when a vulnerability was actually discovered, we rely instead on the vulnerability *disclosure* date as a proxy for the *discovery* date and we study the time needed before a lingering vulnerability gets disclosed.

Usually, when a vulnerability is discovered, a CVE identifier will be reserved for it so it can be uniquely identified later. These CVE IDs can be reserved by a Central Numbering Authorities (CNAs) that have the priority to reserve CVE IDs from MITRE [17]. Snyk, GitHub and HackerOne are examples of such CNAs. To study the disclosure time in more detail, we inspected MITRE using the CVE identifiers we obtained for the vulnerabilities in our vulnerability dataset. We extracted the reservation date and CNA of each CVE and found that only 1,487 vulnerabilities (55%) have a CVE linked to them. 820 of them were disclosed before the CVE reservation date, 639

---

[16] https://snyk.io/vuln/SNYK-DEBIAN9-SUDO-1065095

[17] https://cve.mitre.org/cve/request_id.html

were disclosed after that date, and 100 vulnerabilities had a disclosure date that coincided with the CVE reservation date. We found many organizations responsible for reserving the CVEs of vulnerabilities in npm and RubyGems packages, including Red Hat, Microsoft and GitHub. The top three of CNAs with the most CVEs was HackerOne, Mitre and Snyk for npm, and Mitre, Red Hat and HackerOne for RubyGems.

For all vulnerabilities, we computed the *disclosure lag*, i.e., the number of days between the first affected release and the vulnerability disclosure date. Figure 6 shows the evolution over time of the disclosure lag distribution, grouped by CNA. A gap can be observed between 2018 and 2020 for *RubyGems - NO CVE* because during this period we did not find any disclosed vulnerability for RubyGems with the NO CVE id. The disclosure lag tends to increase in both package distributions, implying that recently reported vulnerabilities take longer to disclose than older ones. This trend can be observed for all CNAs. We also observe a longer disclosure lag for RubyGems vulnerabilities than for npm ones. Comparing between CNAs, Figure 6 also reveals that some CNAs tend to disclose and report vulnerabilities faster than others.

To make a fair comparison, we decided to focus on recent vulnerabilities that have been disclosed in the last three years, i.e., after 2017-04-17. In addition, we focus only on vulnerabilities that have been disclosed in packages that received at least one update in the last two years. Indeed, since inactive packages have less maintenance activity, it would be unfair to compare vulnerabilities disclosed in these packages with vulnerabilities of packages receiving continuous maintenance, including vulnerability inspection. This led us to consider only 1,276 vulnerabilities (45.8%) for RQ_1 and RQ_2. Figure 7 shows the proportion and number of vulnerabilities kept after this filtering. Compared to the unfiltered set in Figure 5 there is a considerably higher proportion of critical vulnerabilities.

Figure 8 shows the cumulative proportion of disclosed vulnerabilities and their disclosure lag, grouped by severity level. For npm, we observe that critical vulnerabilities are the fastest to disclose. It only takes 3.1 months to disclose 50% of all *critical* vulnerabilities, while it takes respectively 49.3, 49.3 and 44.5 months to disclose 50% of all *low, medium* and *high* vulnerabilities. For RubyGems, the disclosure lag seems to depend much less on its severity level. Using log-rank tests for both package distributions, we could only confirm a statistically significant difference for the comparisons with *critical* vulnerabilities in npm (e.g., *critical* vs *high*). For all other comparisons in RubyGems, the null hypothesis stating that there is no difference in disclosure lag could not be rejected.

Disregarding the severity status, vulnerabilities have a lower disclosure lag for npm than for RubyGems (in npm it takes 31.5 months to disclose 50% of all vulnerabilities compared to 84.4 months for RubyGems). To statistically confirm this observed difference in disclosure lag, we carried out a Mann-Whitney U test. The null hypothesis could be rejected with a *large* effect size ($|d| = 0.53$), statistically confirming that vulnerabilities take longer to disclose for RubyGems than for npm. This could also be observed when grouping the
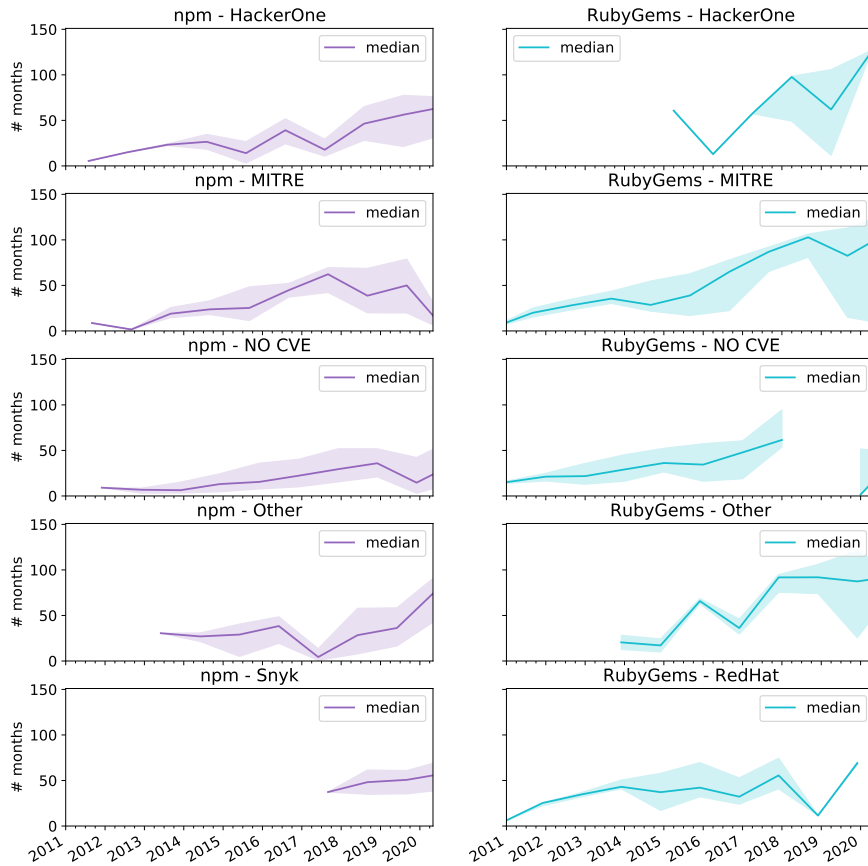
Fig. 6 Evolution of the vulnerability disclosure lag distribution in npm and RubyGems, grouped by CVE Central Numbering Authority. The shaded areas represent the interval between the 25th and 75th percentile.

analysis by CNA. Possible explanations for this finding may be that npm has better security detection tools and more security researchers than RubyGems. We found that vulnerabilities of npm packages were disclosed by 635 security researchers and communicated to MITRE via 21 CNAs, while RubyGems vulnerabilities were disclosed by 236 security researchers and communicated to MITRE via 17 CNAs.

Redoing the same analysis for *inactive* libraries, we found that it took 24.8 and 34.3 months to disclose 50% of the vulnerabilities in inactive npm and RubyGems packages, respectively. To compare between inactive and active packages, we carried out Mann-Whitney U tests. We could only find a statistically significant difference with a *large* effect size ($|d| = 0.48$) in favor of active packages when doing the comparison for RubyGems. This means that, for
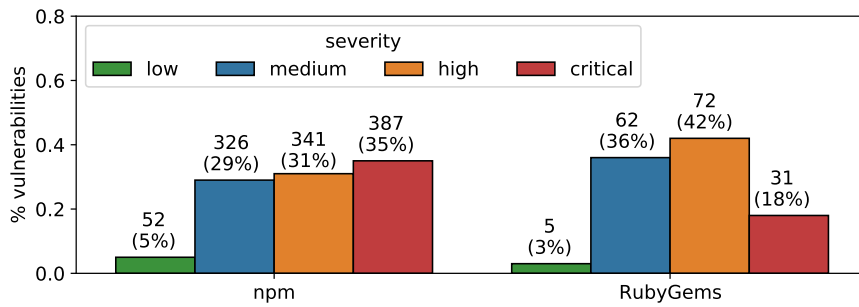
Fig. 7 Proportion and number of vulnerabilities after filtering out old vulnerabilities and inactive packages.

RubyGems, vulnerabilities of inactive packages took less time to be disclosed than those of active packages.

Since *Malicious Package* vulnerabilities are intentionally injected in the form of new updates or packages, we compared them to other vulnerabilities and expected them to be disclosed faster. Indeed, we found that 50% of the *Malicious Package* vulnerabilities in npm are disclosed within 12 days, while those of RubyGems are disclosed within 35 days. Carrying out Mann-Whitney U tests between *Malicious Package* vulnerabilities and other vulnerabilities, we could only find a statistically significant difference with a *small* effect size ($|d| = 0.25$) in favor of non-*Malicious Package* vulnerabilities when doing the comparison for npm. This means that, for npm, *Malicious Package* vulnerabilities are disclosed faster than other vulnerabilities.
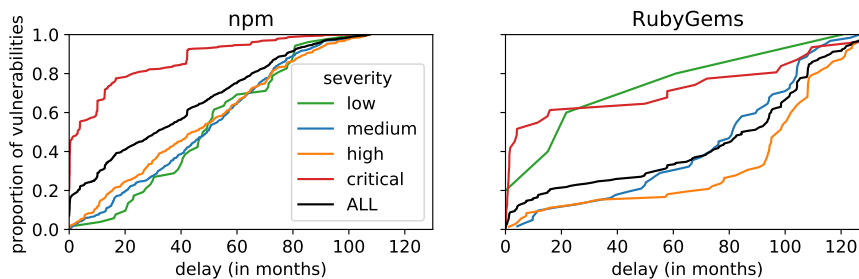
Fig. 8 Cumulative proportion of disclosed vulnerabilities in function of the time elapsed since the first affected package release, grouped by severity level.

> In npm, critical vulnerabilities are disclosed faster. Vulnerabilities in npm are
> disclosed faster than in RubyGems. It takes 2.3 and 7 years to disclose half
> of the vulnerabilities lingering in npm and RubyGems packages, respectively.
> The vulnerability disclosure lag has been increasing over time in both package
> distributions. In npm, Malicious Package vulnerabilities are disclosed faster
> than other vulnerability types.

4.3 RQ_2: For how long do packages remain affected by disclosed vulnerabilities?

RQ_1 studied the *disclosure lag* between the first package release affected by a vulnerability and the moment this vulnerability was disclosed. RQ_2 investigates the time that a vulnerability remains in a package until its fix (1) since the first affected release and (2) since the disclosure time. Case (1) refers to the number of days between the release date of the first affected release and the release date of the first package update that is no longer affected by the vulnerability, while case (2) refers to the number of days between the disclosure date of the vulnerability and the release date of the first package update that is no longer affected by the vulnerability. This analysis is relevant since the longer a package remains affected, the longer it will remain a source of vulnerabilities to potential users and dependents. The latter will be obliged to rely on vulnerable package releases as long as the package maintainers did not release a package update that fixes the vulnerability. This also harms the package dependency network since if a vulnerability fix is being delayed, more and more dependents may potentially make use of the vulnerable package or may become potentially exposed through transitive dependencies. The later a fix becomes available, the more difficult it becomes for all dependents to update their transitive dependencies.

RQ_2 uses the same filtered dataset as RQ_1, ignoring inactive packages and focusing only on recent vulnerabilities. In addition, we exclude *Malicious Package* vulnerabilities since they are known to be fixed in a different way, most frequently by simply removing the malicious package (release) from the registry.

The analysis of RQ_2 will be subdivided into three parts: RQ_2^a a characterisation of the type of versions in which vulnerabilities are fixed; RQ_2^b the time between the first affected release and the fix; and RQ_2^c the time between the vulnerability disclosure and the fix.

RQ_2^a *Version types in which vulnerabilities are fixed.*

Relying on the *semver* specification [6, 34], we studied the package release version (i.e., patch, minor or major) that fixes a vulnerability. The purpose is to determine whether there is a relation between the severity of the vulnerability

and the version type of the first release that included a fix. According to *semver*, patch releases are the most likely candidates for vulnerability fixes.

As both npm and RubyGems promote and encourage the use of *semver*, we expect to find most of their vulnerable packages to be fixed in a patch release. Figure 9 presents stacked bar plots showing the proportion of fixed vulnerabilities per severity, grouped by the version type of the first unaffected release.[18] Our expectations are confirmed, since the majority of vulnerabilities are fixed in patch releases. Disregarding the severity level, 65% of the vulnerabilities were fixed in patch releases, 22.2% in minor releases and only 12.8% in major releases. The severity of a vulnerability does not seem to play a major role in the version type of the release that contains its fix. An exception are the *low* vulnerabilities in RubyGems: in contrast to npm, they do not seem to be considered as important, as they are fixed more often in minor and major releases that incorporate other types of changes as well.
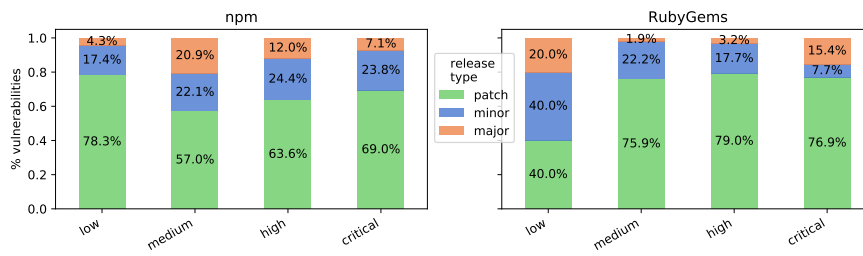
Fig. 9 Proportion of fixed vulnerabilities per severity, grouped by version type of the first unaffected release.

65% of all disclosed vulnerabilities are fixed in patch releases. *Low* severity vulnerabilities in RubyGems are fixed less often in patch releases. The severity of a vulnerability does not seem to have an impact on the first release type in which the vulnerability is fixed.
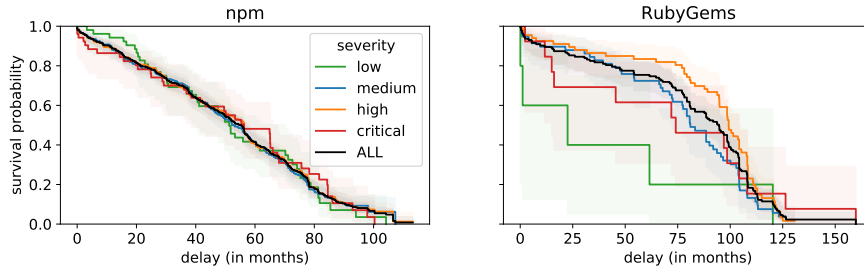
RQ_2^b *Time between the first affected release and the fix.*

Since many vulnerabilities in our dataset did not yet receive a fix (128 out of 693 for npm, and 9 out of 143 for RubyGems), we used a survival analysis [32] to estimate the probability over time for the event "vulnerability is fixed" with respect to the date of the first affected release, and grouped by severity type. Figure 10 shows the Kaplan-Meier survival curves for this analysis. For npm, the confidence intervals of all survival curves overlap, suggesting that there is no difference in the time to fix a vulnerability since its first appearance

---

[18] We implicitly assume here that the first unaffected release is the one containing the fix.

depending on the severity of the vulnerability. A similar observation can be made for RubyGems, with the exception of *low* severity vulnerabilities that seem to be fixed considerably faster than any of the other severity types. It takes only 22 months to fix 50% of all *low* vulnerabilities in RubyGems, while this is 81, 99 and 74 months for, *medium*, *high* and *critical* vulnerabilities. However, log-rank tests did not allow us to confirm any statistical differences between severity levels for any package distribution.

When ignoring the severity level, however, a log-rank test could confirm a statistically significant difference in fixing time (since the first affected release) between npm and RubyGems. Vulnerabilities in npm have a considerably smaller fixing time. For example, half of all vulnerabilities in npm are fixed after 55 months, while it takes 94 months in RubyGems.

Fig. 10 Survival probability for event "vulnerability is fixed" since the first affected release. The shaded colored areas represent the confidence intervals ($\alpha = 0.05$) of the survival curves.

The fairly long observed time between the first affected release and the fix suggests that vulnerabilities affect many releases before they are fixed. Indeed, focusing only on vulnerabilities that have received a fix, an npm vulnerability affects a median of 30 package releases before it is fixed. While for RubyGems, it affects a median of 59 releases. The boxen plots of Figure 11 show how many releases of a vulnerable package are affected by each severity level. We observe that the number of affected releases per vulnerable package is quite high. We also observe that for RubyGems, *low* severity vulnerabilities affect less releases than other vulnerabilities. This is in line with the observations made in Figure 10.

Disclosed vulnerabilities in npm take a considerably shorter time to fix since the first affected release. Half of all disclosed npm vulnerabilities take 55 months to fix since their introduction, compared to 94 months for disclosed RubyGems vulnerabilities. As a consequence, the impact of vulnerabilities is higher for RubyGems, affecting a median of 59 package releases compared to 30 package releases for npm.
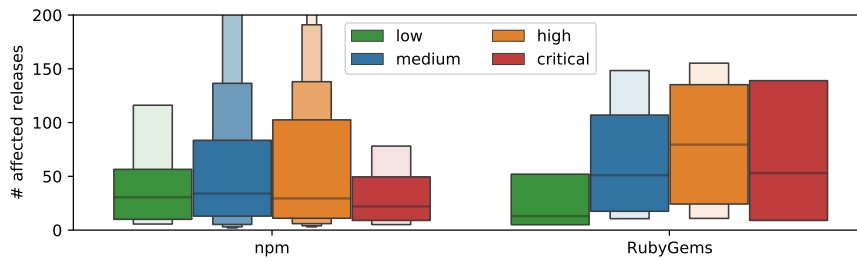
Fig. 11 Boxen plots of the distribution of the number of affected releases of vulnerable packages in npm and RubyGems, grouped by severity.

RQ_2^c *Time between the vulnerability disclosure and the fix.*

It is important to fix vulnerabilities rapidly after their discovery, and especially after they have been publicly disclosed. If a vulnerability is publicly disclosed before a fix is available, more attackers will know about it and will be able to exploit it. Hence, when an open source vulnerability is reported to a security monitoring service, it is usually first disclosed privately in order to give the maintainers time to fix it before it is made public. For example, when Snyk receives a report about a vulnerable package, it informs the package maintainers and gives them 90 days to issue a remediation of the vulnerability. An extension can be granted at the maintainers' request, depending on the severity of the discovered vulnerability. To investigate whether maintainers fix their vulnerabilities within 90 days, we computed the time difference between the vulnerability fix date and the date when it was disclosed [19]. We found a significant proportion of vulnerabilities that exceeded this 90 day period: 17.8% for npm and 10% for RubyGems.

According to a survey carried out by Snyk in 2017, *"34% of maintainers said they could respond to a security issue within a day of learning about it, and 60% said they could respond within a week"* [4]. Our own quantitative observations align with this claim: For npm, 36.9% of all vulnerabilities with a known fix were fixed within a day and 54.4% were fixed within a week, while for RubyGems 46.2% were fixed within a day and 63% were fixed within a week. Overall, 38.9% of all vulnerabilities with a known fix were fixed within a day, and 56.3% were fixed within a week after their disclosure.

> For npm, 17.8% of the fixed vulnerabilities needed more than 90 days after their disclosure to be fixed, while this proportion is 10% for RubyGems. 38.9% of all fixed vulnerabilities were fixed within a day, and 56.3% were fixed within a week after their disclosure.

---

[19] This analysis included Malicious Package vulnerabilities

4.4 RQ_3: To what extent are dependents exposed to their vulnerable dependencies?

We have so far only studied vulnerable package releases. RQ_3 focuses on packages as well as external projects that may be *exposed* to a vulnerability within their direct or indirect dependencies. This exposure may lead to a security breach if the affected functionality of the vulnerable dependency is being used. RQ_3^a will study the exposure of *the latest package releases* to direct or indirect vulnerable dependencies, whereas RQ_3^b will study the exposure of *external projects* to such vulnerable dependencies.

RQ_3^a: To what extent are packages exposed to their vulnerable dependencies?

For all 842,697 of the latest package releases available in the npm and RubyGems snapshots, we determined the direct and indirect dependencies by resolving their dependency constraints. We narrowed down the analysis to those dependencies that are referenced in the vulnerability dataset. 42.1% of all considered npm packages (315,315 out of 748,026) and 39% of all considered RubyGems packages (36,957 out of 94,671) were found to have at least one vulnerable direct or indirect dependency in their latest release. More specifically, 15.7% of npm latest releases and 17.8% of RubyGems latest releases are directly exposed, while 36.5% of npm releases and 27.1% of RubyGems releases are indirectly exposed [20]. This is in line with the findings of Zimmerman et al. [25] reporting that up to 40% of all packages depend on code with at least one publicly known vulnerability.

> More than 15% of the (latest) dependent package releases are exposed to vulnerable **direct** dependencies. 36.5% of npm and 27.1% of RubyGems latest package releases are exposed to vulnerabilities coming from vulnerable **indirect** dependencies.

Vulnerable direct dependencies. We found only a small minority of direct dependencies (i.e., package releases on which at least one other package directly depends) to be vulnerable (because they contain at least one vulnerability). Of all 3,638,361 dependencies considered for npm, only 154,455 (4.2%) are vulnerable; and of all 224,959 dependencies considered for RubyGems, only 20,336 (9%) are vulnerable. Figure 12 shows the distribution, per severity category, of the number of vulnerable package releases found among the direct dependencies in npm and RubyGems. We observe that medium and high severity vulnerabilities are the most common among vulnerable dependencies. We also observe that vulnerable dependencies in RubyGems tend to be more often of medium severity than in npm (blue boxen-plot), and less often of high severity (orange boxen-plot). To statistically confirm these observed differences between npm and RubyGems, we carried out Mann-Whitney U tests

---

[20] The two categories of directly and indirectly exposed package releases are non-exclusive.

to compare the number of vulnerabilities per severity type. The null hypothesis could be rejected for all comparisons. However, the effect size (shown in Table 3) was *negligible* for all comparisons, except for the category of high severity vulnerabilities where a *moderate* effect size was reported. Disregarding the severity category, the effect size was *small* in favour of npm, suggesting that direct dependencies in npm tend to have more vulnerabilities.
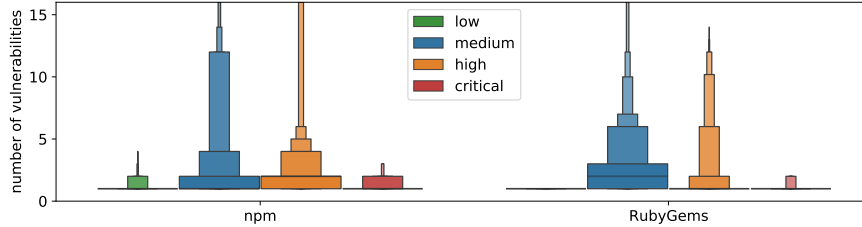
Fig. 12 Boxen plots showing the distribution of the number of vulnerabilities in vulnerable **direct** dependencies of the npm and RubyGems snapshots, grouped by severity.

Table 3 Mean and median number of vulnerabilities found in direct dependencies, in addition to effect sizes and their directions when comparing npm and RubyGems dependency vulnerabilities.

|	npm		RubyGems		direction	|d|	effect size
	mean	median	mean	median			
low	1.07	1	1.02	1	>	0.05	negligible
medium	2.55	1	2.73	2	<	0.06	negligible
high	2.32	2	1.81	1	>	0.39	**moderate**
critical	1.17	1	1.04	1	>	0.11	negligible
all	2.22	1	1.98	1	>	0.16	small

> RubyGems has more than twice the proportion of vulnerable direct dependencies than npm (9% compared to 4.2%). On the other hand, direct dependencies in npm tend to have more vulnerabilities than direct dependencies in RubyGems.

Vulnerable indirect dependencies. Releases may also be exposed *indirectly* to vulnerable dependencies. 1,225,724 out of the 64,959,052 indirect dependencies in npm (1.9%) are vulnerable; whereas a much higher proportion of 65,090 out of 1,033,870 indirect dependencies in RubyGems (6.3%) are vulnerable. Figure 13 shows the distribution, per severity category, of the number of vulnerable package releases found among the indirect dependencies for npm and RubyGems. Similar to Figure 12, medium and high severity vulnerabilities are the most common. We also observe a higher number of vulnerabilities for each

severity level for npm. Mann-Whitney U tests comparing the distributions be-
tween npm and RubyGems confirm a statistically significant difference. The
effect sizes for low, medium, high and critical vulnerabilities are *small* to *mod-
erate* in favor of npm (see Table 4). Disregarding the severity category, the
effect size is small (i.e., $|d| = 0.32$).

Table 4 Mean and median number of vulnerabilities found in indirect dependencies, in
addition to effect sizes and their directions when comparing npm and RubyGems dependency
vulnerabilities.

|	npm		RubyGems		direction	$\|d\|$	effect size
	mean	median	mean	median			
low	1.95	1	1.08	1	>	0.43	**moderate**
medium	3.53	2	2.77	1	>	0.32	small
high	3.41	2	2	2	>	0.27	small
critical	1.33	1	1.12	1	>	0.15	small
all	3.08	2	1.94	1	>	0.32	small
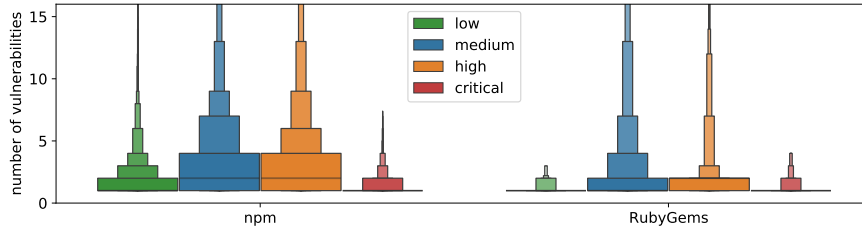
Fig. 13 Boxen plots showing the distribution of the number of vulnerabilities in vulnerable
indirect dependencies of the npm and RubyGems snapshots, grouped by severity.

The proportion of vulnerable indirect dependencies for RubyGems (6.3%) is
more than three times higher than for npm (1.9%). On the other hand, for
each severity, vulnerable indirect dependencies in npm have more vulnera-
bilities than in RubyGems.

Vulnerabilities of all transitive dependencies. Considering both direct
and indirect dependencies, we investigate whether the number of dependency
vulnerabilities is related to the date when the studied package was released.
Figure 14 visualises the monthly evolution of the distribution of the number
of dependency vulnerabilities for all packages released during that month. In
general, we observe that for both ecosystems, the number of vulnerabilities
decreased over time, in the sense that more recent packages are exposed to less
vulnerabilities coming from their dependencies than older packages. However,
we notice that the number of vulnerabilities for npm was increasing over time

until 2014 (i.e., dashed line in red), after which it started decreasing. This coincides with the time when npm introduced the permissive constraint *caret* (^) as default constraint for npm dependencies instead of the more restrictive constraint *tilde* (∼). Caret accepts new patch and minor releases to be installed while tilde only accepts new patches. This means that packages with permissive dependency constraints are exposed to less dependency vulnerabilities than those with restrictive constraints. Intuitively, permissive constraints accept wider ranges of releases than restrictive ones and thus they provide more opportunities to install dependencies with fixed vulnerabilities.
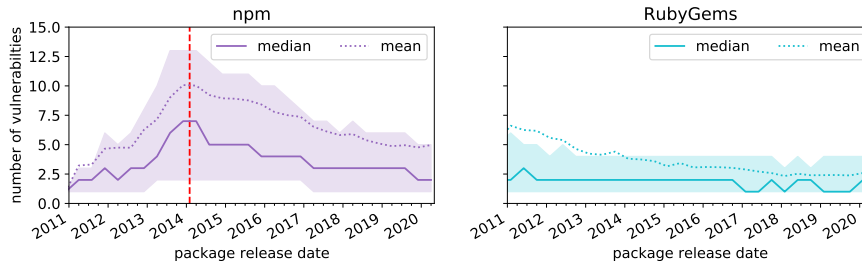

Fig. 14 Monthly evolution of the distribution of the number of vulnerabilities coming from transitive dependencies of all studied packages. The shaded areas correspond to the interval between the 25th and 75th percentile.

> Older packages are exposed to more vulnerabilities coming from their dependencies than recent ones. The introduction of the permissive dependency constraint caret in npm led to have packages with less vulnerable transitive dependencies.

Exposed packages. Only 849 vulnerable packages used as dependencies (667 for npm and 182 for RubyGems) are responsible for all exposed packages. This means that 60.1% and 43.3% of vulnerable packages in npm and RubyGems is never used as a dependency, as the number of vulnerable packages we originally found is 1,672 for npm and 321 for RubyGems. Moreover, only a small subset of the used vulnerable packages is responsible for most of the vulnerabilities found among direct and indirect dependencies. In fact, 90% of the vulnerabilities found in npm dependencies come from 50 packages only. For RubyGems this subset is even smaller with only 20 packages responsible for 90% of the vulnerabilities.

Starting from a unique vulnerable package that is used as a dependency, we quantify the number of dependent packages that are directly or indirectly exposed to a vulnerability because of it. Figure 15 shows the distribution, revealing that considerably more packages are indirectly exposed to vulnerabilities. The median number of packages that is directly exposed to one vulnerable package is 11 for npm and 12 for RubyGems, while it is about twice as high

for indirect exposures (26 for npm and 21.5 for RubyGems). We carried out Mann-Whitney U tests to confirm that the distribution for indirectly exposed packages is higher than for directly exposed packages between the distributions. The null hypothesis could only be rejected for npm with a *small* effect size ($|d| = 0.19$). Without distinguishing between direct or indirect dependencies, one single vulnerable package is responsible for exposing a median of 21 and a maximum of 213,851 (67.8%) npm packages, and a median of 19 and a maximum of 22,233 (60.2%) RubyGems packages, respectively.
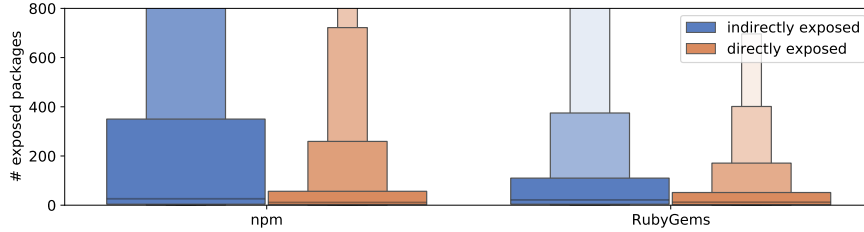
Fig. 15 Distribution of the number of exposed npm and RubyGems packages affected by a given vulnerable package.

A limited set of vulnerable packages is responsible for most of the vulnerabilities exposed through dependencies. One single vulnerable package can be responsible for exposing two thirds of all dependent latest package releases.

RQ_3^b: **To what extent are external projects exposed to their vulnerable dependencies?**

The main purpose of package managers such as npm and RubyGems is to facilitate depending on packages. Therefore, not only packages but also external projects can be exposed to the vulnerabilities of their dependencies. We intuitively expect external projects to be more exposed to vulnerabilities through their dependents than packages distributed through the package manager, as the maintainers of the latter intend their packages to be depended on.

For the 24,593 collected external projects (see Section 3.2) we searched for all dependencies referenced in the vulnerability dataset. We found 79% external projects for npm (11,003 out of 13,930) and 74.1% for RubyGems (7,901 out of 10,663) with at least one direct or indirect vulnerable dependency. More specifically, for npm 47% of all external projects are directly exposed and 70.4% are indirectly exposed; whereas for RubyGems 54% of all external projects are directly exposed and 66.4% are indirectly exposed [21].

---

[21] The two categories of directly and indirectly exposed projects are non-exclusive.

About half of all external projects (47% for npm and 54% for RubyGems) are exposed to vulnerabilities coming from vulnerable **direct** dependencies. About two thirds of all external projects (70.4% for npm and 66.4% for RubyGems) are exposed to vulnerabilities coming from vulnerable **indirect** dependencies.

Vulnerable direct dependencies of external projects. Out of 147,622 of the direct dependencies of external projects on npm packages, 11,969 (8.1%) are vulnerable. Out of 101,079 of the direct dependencies of external projects on RubyGems packages, 11,034 (10.9%) are vulnerable. Figure 16 shows the distribution of the number of vulnerabilities affecting direct dependencies of external projects. Similar to what we observed for RQ_3^a, medium and high severity vulnerabilities are the most common among direct dependencies, and dependencies on npm packages tend to have higher numbers of vulnerabilities than dependencies on RubyGems packages. Table 5 shows the mean and median number of vulnerabilities caused by direct dependencies, grouped by severity. We performed Mann-Whitney U tests to compare the number of vulnerabilities in project dependencies between npm and RubyGems. We found statistically significant differences for all compared distributions but the effect size was *negligible*, with one exception: a *moderate* effect size was found for highly vulnerable dependencies.
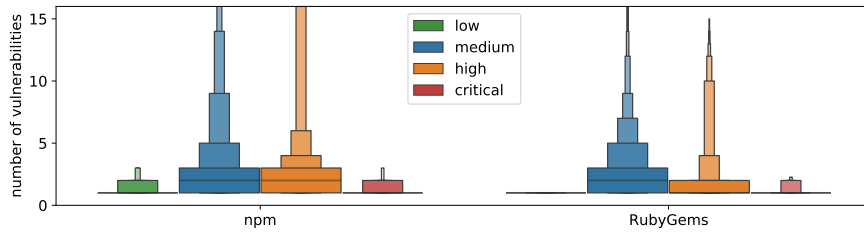
Fig. 16 Boxen plots showing the distribution of the number of vulnerabilities found in vulnerable **direct** npm and RubyGems dependencies of GitHub projects, grouped by severity.

Table 5 Mean and median number of vulnerable dependencies, in addition to effect sizes and their directions, for direct dependencies of GitHub projects on vulnerable npm and RubyGems packages.

|	npm		RubyGems		direction	|d|	effect size
	mean	median	mean	median			
low	1.16	1	1.03	1	>	0.11	negligible
medium	2.96	2	2.80	2	<	0.05	negligible
high	2.80	2	1.78	1	>	0.33	**moderate**
critical	1.17	1	1.07	1	>	0.09	negligible
all	2.60	2	2.27	1	>	0.06	negligible

8.1% of the direct dependencies of external projects on npm are vulnerable, while this is 10.9% for RubyGems. npm-dependent projects have more highly vulnerable direct dependencies than RubyGems-dependent projects.

Vulnerable indirect dependencies of external projects. Out of 2,666,922 indirect dependencies of external projects on npm packages, 87,062 (3.2%) are vulnerable. Out of 443,427 indirect dependencies of external projects on RubyGems packages, 46,682 (10.5%) are vulnerable. Figure 17 shows the distribution of the number of vulnerabilities affecting indirect dependencies of external projects. Similar to Figure 16, medium and high severity vulnerabilities are the most common. We observe that indirect dependencies on RubyGems tend to be more often of medium severity than indirect dependencies on npm packages, while the latter tend be more often of low severity. Mann-Whitney U tests comparing the distributions between npm and RubyGems confirmed a statistically significant difference. Regardless of the severity level we found a *small* effect size ($|d| = 0.15$) in favour of npm dependency vulnerabilities (see Table 6). Per severity levels, we only found a non-negligible effect size for *critical* severity vulnerabilities in favour of RubyGems ($|d| = 0.25$) (i.e., RubyGems projects have more *critical* vulnerabilities coming form indirect dependencies than npm projects), and for *low* severity vulnerabilities in favour of npm ($|d| = 0.71$). The *negligible* effect size for *medium* severity vulnerabilities is in favour of RubyGems ($|d| = 0.1$), while for *high* severity vulnerabilities it is in favour of npm ($|d| = 0.09$).

Table 6 Mean and median number of vulnerable dependencies, in addition to effect sizes and their directions, for indirect dependencies of GitHub projects on vulnerable npm and RubyGems packages.

	npm		RubyGems		direction	|d|	effect size
	mean	median	mean	median			
low	2.88	2	1.05	1	>	0.71	**large**
medium	6.56	4	12.36	5	<	0.1	negligible
high	6.2	4	4.55	3	>	0.09	negligible
critical	1.4	1	1.67	2	<	0.25	small
all	5.16	3	5.74	2	>	0.15	small

Only 3.2% of the indirect npm dependencies of external projects are vulnerable, while this is more than three times higher (10.5%) for RubyGems. Disregarding severities, external projects for npm have more vulnerabilities coming from vulnerable indirect dependencies than for RubyGems. RubyGems external projects have more critical vulnerabilities coming from vulnerable indirect dependencies than npm external projects, while the latter have more low severity vulnerabilities.
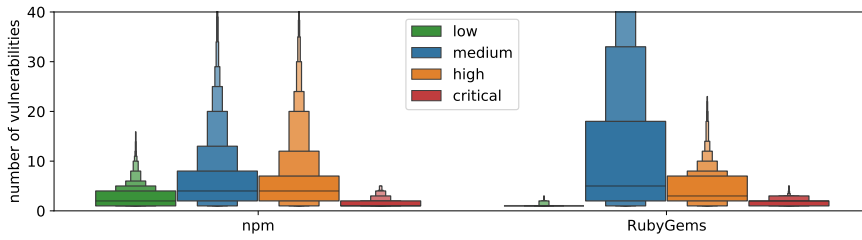
Fig. 17 Distribution of the number of vulnerabilities found in vulnerable **indirect** npm and RubyGems dependencies of GitHub projects, grouped by severity.

Vulnerabilities of all transitive dependencies. Figure 18 visualises the monthly evolution of the distribution of the number of dependency vulnerabilities for all external projects that have their last commit during that month. Similar to Figure 14, we observe that more recently active projects are exposed to less vulnerabilities coming from their dependencies. Also, we can again see the impact of the introduction of the caret (^) dependency constraint in npm in 2014 (left figure).
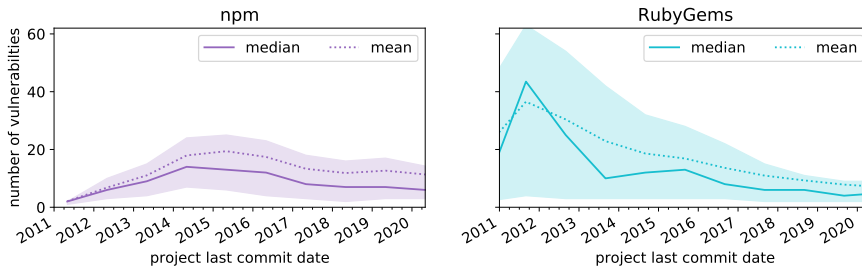
Fig. 18 Monthly evolution of the distribution of the number of vulnerabilities coming from transitive dependencies of all studied external projects. Time points refer to the project's last commit date, i.e., each project is considered only once. The shaded areas correspond to the interval between the 25th and 75th percentile.

> More recently active external projects are exposed to fewer vulnerabilities coming from their dependencies than older ones. The introduction of the permissive dependency constraint caret in npm seems to have led to less vulnerable dependencies in external projects.

Exposed external projects. Only 560 (28%) vulnerable packages (400 for npm and 160 for RubyGems) are responsible for all exposed external projects. This is less than the number we found for exposed packages in RQ_3^a, most likely because we studied fewer external projects than (internal) packages.

Similar to Figure 15, we analysed how many external projects are exposed to vulnerabilities because of a single vulnerable package. Figure 19 shows the distribution of the number of projects that are directly or indirectly exposed to one unique vulnerable package. We observe that there are considerably more indirectly exposed projects than direct ones. The median number of projects that one vulnerable npm package directly exposes is 4, while for indirect exposure it is 12. The median number of projects that one vulnerable RubyGems package directly exposes is 8, while for indirect exposure it is 12.

We carried out Mann-Whitney U tests between the distributions of direct and indirect dependency vulnerabilities exposing external projects. The null hypothesis could only be rejected in the case of npm comparison with a *small* effect size ($|d| = 0.27$) in favor of indirectly exposed projects. Without distinguishing between direct or indirect dependencies, we found that one single vulnerable package is responsible for exposing a median of 8 and a maximum of 7,506 (68.2%) projects on vulnerable npm dependencies; while a median of 13.5 and a maximum of 5,270 (49.4%) projects is exposed to vulnerable RubyGems dependencies.
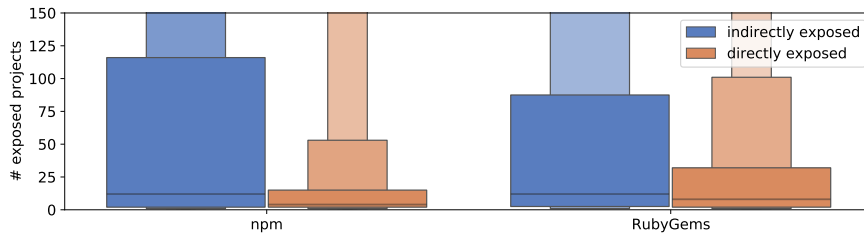
Fig. 19 Distribution of the number of exposed npm and RubyGems projects that one single vulnerable package is affecting.

Only 28% of all vulnerable packages are responsible for all vulnerabilities of exposed external projects. One single vulnerable package can be responsible for exposing 68.2% of all exposed projects that use npm, and 49.4% of all exposed projects that use RubyGems.

4.5 RQ_4: How are vulnerabilities spread in the dependency tree?

RQ_3 revealed that dependents are considerably more often indirectly exposed to vulnerabilities than directly. With RQ_4, we want to know how deep in the dependency tree we can find vulnerabilities to which packages and external projects are exposed. This allows us to quantify the transitive impact that vulnerable packages may have on their (transitive) dependents. In an earlier study focusing on the package dependency networks of 7 different package distributions (including npm and RubyGems), Decan et al. [1] analysed the

prevalence of indirect dependencies. They observed that more than 50% of the top-level packages[22] in npm have a dependency tree depth of at least 5, whereas for the large majority of top-level packages in RubyGems this was 3 or less. As a result, npm appears to be potentially much more subject to deep vulnerable dependencies. RQ_4 aims to quantify this claim, focusing on the latest package releases in RQ_4^a and external projects in RQ_4^b.

RQ_4^a: How are vulnerabilities spread in the dependency trees of packages?

We computed the number of vulnerabilities at each depth for all vulnerable (direct or indirect) dependencies. Figure 20 shows the distribution of the number of vulnerabilities in dependencies, grouped by dependency depth. The first level (corresponding to direct dependencies) and the second level (dependencies of dependencies) have higher numbers of vulnerabilities. The number of vulnerabilities decreases at deeper levels, as a consequence of the fact that there are fewer releases with deep dependency trees. Statistical comparisons confirmed, with non-negligible effect sizes, that more shallow levels correspond to higher numbers of vulnerabilities. Nevertheless, vulnerabilities do remain present at the deepest levels. For example, we could find at least one vulnerable dependency at depth 16 for npm package formcore that is indirectly exposed to a vulnerability in package kind-of, and at depth 10 for RubyGems package erp_inventory that is indirectly exposed to a vulnerability in package rack.

The vulnerable packages used as dependencies in npm with the highest number of vulnerabilities are node-sass, lodash and minimist, while for RubyGems they are nokogirl, activerecord and actionpack. Unsurprisingly, we found the same set of vulnerable dependencies and vulnerability types reoccurring at deeper dependency levels. The most prevalent vulnerability type for npm was *Prototype Pollution (PP)*, while for RubyGems dependencies it was *Denial of Service (DoS)* (see Table 2 for an overview of the most common vulnerability types).

> The number of dependency vulnerabilities for the latest package releases decreases at deeper levels of the dependency tree. Yet, vulnerable dependencies continue to be found at the deepest levels. The same vulnerability types can be found at all dependency tree levels.

We also studied how deep in the dependency tree a vulnerability can reach by identifying for each exposed package the maximum dependency depth where a vulnerable dependency could be found (Figure 21). For npm, the number of exposed packages increases from the dependency depth 1 (direct dependencies) to depth 4 and then starts to decrease. For RubyGems, the numbers start to decrease starting from depth 1. This implies that npm packages are more susceptible to indirect vulnerable dependencies at deeper dependency levels.

---

[22] Top-level packages are packages that do not have any dependent packages themselves.
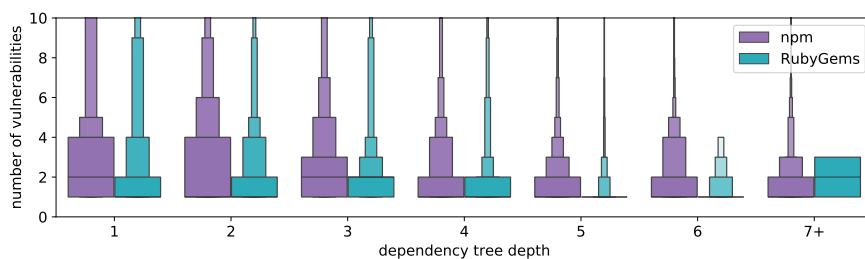
Fig. 20 Distribution of the number of vulnerabilities found in all (direct and indirect) dependencies of npm and RubyGems latest package releases, grouped by dependency tree depth.

This seems to be in line with the findings of Decan et al. [1] mentioned at the beginning of this research question.
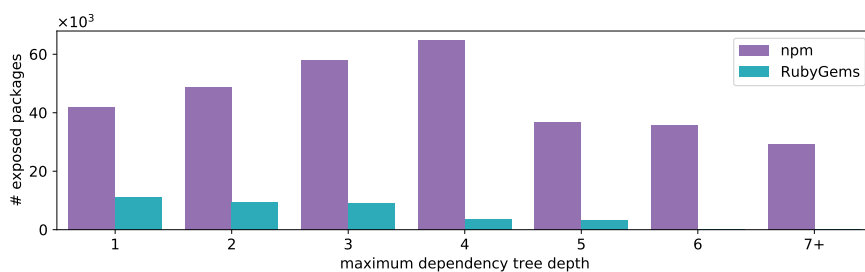
Fig. 21 Number of exposed npm and RubyGems latest package releases and their maximum dependency depth.

npm packages are more likely than RubyGems packages to be exposed to vulnerabilities deep in their dependency tree.

RQ_4^b: *How are vulnerabilities spread in the dependency trees of external projects?*

To gain more insights about the prevalence of vulnerabilities at different depths in the dependency tree of external projects depending on npm or RubyGems packages, we computed the number of vulnerabilities found at each dependency depth. Figure 22 shows the distribution of the number of vulnerabilities in dependencies, grouped by dependency depth. From the second level onwards, the number of vulnerabilities starts to decrease with increasing levels of depth. As for RQ_4^a, statistical comparisons confirmed, with non-negligible effect sizes, that more shallow levels correspond to higher numbers of vulnerabilities. Yet, vulnerabilities continue to remain present at the deepest levels.

For example, we could find at least one external project with a vulnerable dependency at depth 14 on npm package kind-of, and one external project with a vulnerable dependency at depth 7 on RubyGems package nokogiri. We also observe that external projects for RubyGems and npm have the same median number of dependency vulnerabilities at depths 2 and 3, while starting from level 4 we observe the same trend as in Figure 20. The reason for this difference at shallow depths (compared to what we saw in RQ_4^a) is because external projects depending on RubyGems tend to include more direct dependencies than normal RubyGems packages.
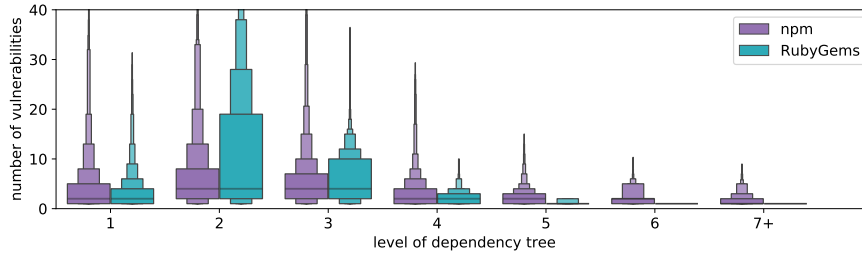
Fig. 22 Distribution of the number of vulnerabilities found in all (direct and indirect) dependencies of npm and RubyGemsGitHub projects, grouped by dependency depth.

> Starting from depth 2, the number of vulnerable dependencies for external projects decreases as the depth within the dependency tree increases. Yet, vulnerable dependencies are still found at the deepest levels.

We also analysed how far in the dependency tree a vulnerability can reach. Figure 23 shows the number of exposed external projects with the maximum depth at which we found at least one vulnerable dependency. We observe that for both npm and RubyGems, the number of exposed external projects increases from the first level (direct dependencies) until the fourth and third levels, respectively, and then starts decreasing. Comparing this finding with Figure 21 in RQ_4^a, we observe that RubyGems projects are more susceptible to having vulnerable dependencies at deeper levels than RubyGems packages.

> External projects dependent on RubyGems are more likely to be exposed to vulnerabilities deep in their dependency tree than external projects for npm.

4.6 RQ_5: Do exposed dependents upgrade their vulnerable dependencies when a vulnerability fix is released?

57.8% of the vulnerabilities in our dataset (i.e., 1,611 to be precise) have a known fix. With RQ_5 we aim to quantify how much dependent packages and dependent external projects would benefit from updating their dependencies for which there is a known fix available. Upgrading their dependencies to more
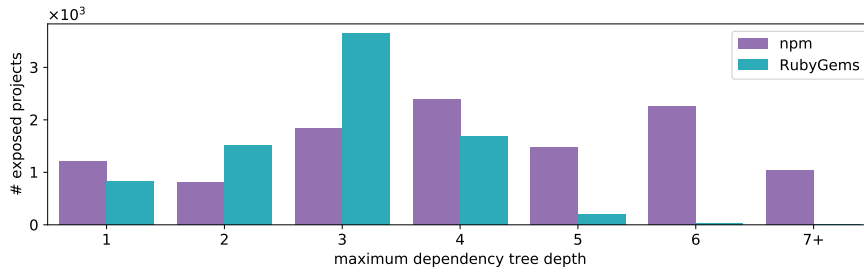
Fig. 23 Number of exposed external projects for npm and RubyGems and their maximum dependency depth, grouped by dependency tree depth.

recent releases will reduce their exposure to vulnerabilities. First, we start by exploring how many vulnerable dependencies have known vulnerability fixes. Table 7 shows the proportion of direct and indirect dependencies that are only affected by vulnerabilities that have a known fix. We observe that for the large majority of the vulnerable dependencies, fixes are available (more than 90% for npm and more than 60% for RubyGems). npm indirect dependencies have more fixed vulnerabilities than direct ones, while for RubyGems we observe the inverse. We also observe that fewer affected RubyGems dependencies have fixes available than npm dependencies. These results show that most of the vulnerable dependencies could be made safe if the maintainers of the dependent packages or projects would choose the appropriate non-vulnerable version of their dependencies.

Table 7 Proportion of vulnerable dependencies (for packages and external projects) having a known fix.

	packages		projects	
	direct	indirect	direct	indirect
npm	90.0	96.9	92.5	97.1
RubyGems	85.2	66.7	90.0	76.0

Updating a dependency is not always easy, especially when maintainers of dependent projects or packages would be confronted with breaking changes. For example, if a package release or external project uses a dependency that resolves to version 1.1.0 while a vulnerability fix exists in 1.2.0 then the exposure to the vulnerability can be removed by only doing a minor version update of the dependency. On the other hand, if a package release or external project uses a dependency that resolves to version 1.1.0 while a vulnerability fix exists in 2.0.0, then one would need to update to a new major version. If the dependency is adhering to *semver*, then the second but not the first kind of dependency update would require updating the dependent's implementation according to backwards incompatible changes. We therefore verified whether it would be possible for dependent latest package releases and external projects

to avoid vulnerabilities by only updating their vulnerable dependencies to a higher version within the same major version range that is currently in use. Focusing only on direct dependencies, 32.8% of the vulnerabilities affecting direct dependencies of latest package releases and 40.9% of the vulnerabilities affecting direct dependencies of external projects, could be avoided by making backward compatible dependency updates. For indirect dependencies, 22.1% of the vulnerabilities affecting indirect dependencies of latest package releases and 50.3% of the vulnerabilities affecting indirect dependencies of external projects, have a fix within the major version range that is currently in use. We also found that 5.4% of the exposed latest package releases and 5% of the exposed external projects could be made completely vulnerability-free by only making backward compatible dependency updates to their vulnerable direct dependencies.

> Vulnerability fixes are available for the large majority of vulnerable dependencies. Around one out of three dependency vulnerabilities to which the latest package releases or external projects are exposed, could be avoided if software developers would update their direct dependencies to more recent releases within the same major release range. Performing backward compatible updates to vulnerable direct dependencies could make 5.4% of the exposed packages and 5% of the exposed external projects completely vulnerability-free.

4.7 RQ_6: To what extent are dependents exposed to their vulnerable dependencies at their release time?

RQ_3, RQ_4 and RQ_5 studied vulnerabilities of dependencies used in dependent packages and external projects as if they were deployed on 12 January 2020. This led us to consider all vulnerabilities already disclosed and reported in Snyk's dataset. RQ_6 investigates if dependents were already incorporating dependencies with disclosed vulnerabilities at their release time. To do so, we first need to resolve dependency constraints used in package releases and external projects at the time of their release. Then, we identify dependencies affected by disclosed vulnerabilities only. For example, the latest version of the package `node-sql` [23] was released on August 2017 while depending on the package `lodash` [24]. If we resolve the used version of `lodash` on August 2017, we find that it was `4.1.0` which is affected by the vulnerability `CVE-2019-10744` [25]. However, at the version release date in 2017, this vulnerability was not disclosed yet and thus the developers of `sql-node` could not do anything about it. Answering this research question will help us to assess how careful developers are when incorporating dependencies with already disclosed vulnerabilities. Therefore, we will only focus on direct dependencies.

---

[23] `https://www.npmjs.com/package/sql`
[24] `https://www.npmjs.com/package/lodash`
[25] `https://nvd.nist.gov/vuln/detail/cve-2019-10744`

RQ_6^a: *To what extent are packages exposed to their vulnerable dependencies at their release time?*

For all 842,697 of the latest package releases available in the npm and RubyGems snapshots, we determined their direct dependencies by resolving their dependency constraints at their release time. We narrowed down the analysis to those dependencies that are referenced in the vulnerability dataset. 6.8% of all considered npm latest package releases (50,720 out of 748,026) and 6.2% of all considered RubyGems latest package releases (5,896 out of 94,671) were found to have, at their release dates, at least one direct dependency affected by at least one vulnerability that is already disclosed.

Moreover, of all 3,638,361 direct dependencies considered for npm packages, only 58,184 (1.6%) were affected by vulnerabilities disclosed before the latest package release in which the dependencies are incorporated. For RubyGems, of all 224,959 direct dependencies, only 6,410 (2.8%) were affected. Table 8 shows more details about the number of disclosed vulnerabilities found in direct dependencies.

Comparing these results to RQ_3^a, we can clearly see that at their creation dates, the latest package releases were exposed to fewer disclosed vulnerabilities than on 12 January 2020 (the dataset snapshot date). Moreover, more than half of the package releases that are exposed to vulnerabilities via their dependencies at the snapshot date, were not exposed to any disclosed vulnerabilities when they were created.

Table 8 Mean and median number of disclosed vulnerabilities found in direct dependencies at the package release creation date, in addition to effect sizes and their directions when comparing npm and RubyGems dependency vulnerabilities.

| | npm | | RubyGems | | direction | \|d\| | effect size |
	mean	median	mean	median			
low	1.04	1	1.03	1	>	0.01	negligible
medium	1.96	1	1.49	1	>	0.1	negligible
high	1.72	1	1.27	1	>	0.2	small
critical	1.07	1	1	1	>	0.07	negligible
all	1.76	1	1.36	1	>	0.1	negligible

At their release dates, RubyGems latest package releases had proportionally more vulnerable direct dependencies than npm (2.8% compared to 1.6%). More than half of the latest package releases that are exposed to vulnerabilities via their dependencies at the observation date, were not exposed to any disclosed vulnerabilities when they were first created.

RQ_6^b: To what extent are external projects exposed to their vulnerable dependencies at the date of their last commit?

For all 24,593 external projects that make use of npm and RubyGems packages, we determined their direct dependencies by resolving their dependency constraints at their release time. 22.1% of all considered external projects for npm (3,077 out of 13,930) and 33.9% of all considered external projects for RubyGems (3,619 out of 10,663) were found to have, at their last commit date, at least one direct dependency affected by at least one vulnerability that is already disclosed.

Out of 147,622 of the direct dependencies of external projects on npm packages, only 4,600 (3.1%) were affected by vulnerabilities disclosed before the date of the last commit. For RubyGems, of all 101,079 direct dependencies of external projects, only 5,264 (5.2%) were affected. Table 9 shows more details about the number of disclosed vulnerabilities found in direct dependencies.

Table 9 Mean and median number of disclosed vulnerabilities found in direct dependencies of GitHub external projects at their last commit dates, in addition to effect sizes and their directions.

| | npm | | RubyGems | | direction | \|d\| | effect size |
	mean	median	mean	median			
low	1.16	1	1.02	1	>	0.12	negligible
medium	2.29	1	1.65	1	>	0.14	negligible
high	1.88	1	1.38	1	>	0.17	small
critical	1.13	1	1.07	1	>	0.04	negligible
all	2.01	1	1.55	1	>	0.11	negligible

Comparing these results to RQ_3^b, we observe that, at the time of their last commit, GitHub external projects were exposed to a lesser number of disclosed vulnerabilities than at the dataset snapshot date (i.e., 12 January 2020). Moreover, more than half of the GitHub projects that make use of npm packages and are exposed to vulnerabilities via their dependencies at the snapshot date, were not exposed to any disclosed vulnerabilities when they were first created. This is different for GitHub projects that make use of RubyGems packages, since only one third of the projects that are exposed to vulnerable dependencies at the snapshot date were not exposed to any vulnerability at the time of their last commit.

> At the time of their last commit, GitHub external projects that make use of RubyGems packages had proportionally more vulnerable direct dependencies than projects with npm dependencies (33.9% compared to 22.1%). Half of the external projects with npm dependencies that are exposed to vulnerabilities at the observation date, were not exposed to any disclosed vulnerability at the time of their last commit, while this is only one third for projects with RubyGems dependencies.

5 Discussion

This section discusses our findings and their implications for developers, security researchers and package managers. It provides insights specific to npm and RubyGems, as well as some challenges that need to be overcome to better secure open source package distributions.

We start our exposition with the vulnerable packages in each package distribution (Section 5.1), continue with the ramifications of how direct and indirect dependents are exposed to these vulnerabilities (Section 5.2) and end with a discussion about comparing different package distributions (Section 5.3).

5.1 Vulnerable packages

In a comparative study of package distributions (for the observation period 2012–2017), Decan et al. [1] observed that both npm and RubyGems have an exponential increase in their number of packages. The monthly number of package updates remained more or less stable for RubyGems, while a clear growth could be observed for npm. RQ_0 builds further upon that work by analysing and comparing how vulnerable packages are in each package distribution using Snyk's dataset of vulnerability reports.

A first observation was that npm has more vulnerabilities affecting more packages than RubyGems. We posit that this is due to the popularity of npm that exposes the package distribution considerably more to attackers and attracts more security researchers. At the time of writing this article, the number of packages distributed through npm is an order of magnitude higher than those distributed through RubyGems. In fact, we found that npm has more security researchers that disclose vulnerabilities than RubyGems. Based on the adage "Given enough eyeballs, all bugs are shallow" [35], the community of package repositories should invest in organizing bug bounty programs to attract more security researchers [36] that may help to discover hidden vulnerabilities. This will also help to reduce the vulnerability disclosure gap (see RQ_1).

Challenge: *How to incite more security researchers to inspect open source packages for vulnerabilities?*

Let us now focus on vulnerability reports with the vulnerability type *Malicious Package*. It is the most prevalent vulnerability type for npm packages. Even though most packages suffering from this vulnerability will eventually be removed from the package distribution, previous studies [37] and experiments [38] have shown that this type of vulnerability can be very dangerous. npm has 410 packages corresponding to the *Malicious Package* vulnerability, while RubyGems has only a mere 19 vulnerabilities of this type.

The second most common vulnerability in npm is *Directory Traversal*, a vulnerability allowing an attacker to access arbitrary files and directories on the application server. This vulnerability could also be classified under two

other types of attacks, namely *Information Exposure* and *Writing Arbitrary Files*. For RubyGems, we found *Information Exposure* as the 4th ranked vulnerability type, probably because its contributors prefer to report *Directory Traversal* under the more general category of *Information Exposure* vulnerabilities.

> **Recommendation:** *Malicious packages are more prevalent in npm than RubyGems. npm users should be aware of the common strategies of malicious packages like Typosquatting [37].*

42% (1,175) of the reported vulnerabilities in the dataset did not have any known fix. We can exclude from them 406 unfixed *Malicious Package* vulnerabilities (389 for npm and 17 for RubyGems) that can ultimately be fixed by simply removing the package or the affected releases from the distribution. This leaves us with 669 vulnerabilities different from the *Malicious Package* type that affect all recent releases of npm packages in which they were discovered, and 100 such vulnerabilities for RubyGems packages. Users of package distributions should be aware of whether package managers incorporate and mark the vulnerable packages and their releases in their registries. Having such information in package managers will help developers and users to decide which releases they should depend upon. It will also simplify the work of tools like Dependabot [26] and Up2Dep [39] that try to find and fix insecure code of used dependencies. Such tools should also be aware that not all vulnerabilities are in MITRE or NVD [27] since we found that 45% of all vulnerabilities do not have a CVE reserved for them. Many of the vulnerability threats and their information are shared through social media channels (e.g., Reddit, Twitter, Stack Overflow, etc) [40]. Therefore, other vulnerability databases, security advisories and issue trackers might help these tools to extend their reach to vulnerabilities that are not in NVD.

> **Recommendation:** *Package managers can help developers by marking vulnerable package releases in their registries.*

To offer more secure packages, all known vulnerabilities should be fixed, discovered and disclosed rapidly. Unfortunately, many vulnerabilities remain undisclosed for months to years, providing attackers plenty of time to search for flaws and develop exploits that can harm millions of users of these packages. For npm, we observed that *critical* vulnerabilities were disclosed more rapidly, while for RubyGems we did not observe a relation to the severity of the vulnerability. Software package maintainers should invest more effort in keeping their packages secure, especially those packages that are frequently used as dependents of other packages or external projects, in order to reduce the transitive exposure to vulnerabilities. They should be supported in this task by additional static and dynamic application security testing tools, perhaps tailored to the characteristics of packages and libraries. For instance,

---

[26] https://dependabot.com/
[27] https://nvd.nist.gov/

there is typically no single method that can serve as the entry point for a whole-program analysis.

Moreover, the longer a vulnerability lasts, and the more releases are affected by it, the more likely it becomes that someone will unknowingly use a vulnerable release. Registries of snapshot-based deployments such as *Docker Hub* that are attracting more and more popularity exacerbate this problem [13, 41] as users might unknowingly reuse images that wrap vulnerable package releases. Unfortunately, the majority of the vulnerabilities in npm and RubyGems packages are fixed within several years after their first introduction leaving most of the package releases affected. However, after their disclosure, vulnerabilities seem to be fixed after a short time. Only about one out of five vulnerabilities takes more than 3 months to be fixed, which is the deadline given by Snyk to maintainers before publishing a reported vulnerability publicly. This means that most of the maintainers are willing to fix vulnerabilities as soon as possible after disclosure.

> **Challenge:** *A vulnerability can stay hidden in a package for years and it affects most of its previous releases before it is finally fixed. Effective static and dynamic analysis tools should emerge to support security researchers in finding vulnerabilities.*

5.2 Dependents exposed to vulnerabilities

RQ_3^a investigated how packages are exposed to vulnerable dependencies. To this end, we studied the latest release of each package available in the considered snapshot of each package distribution. Since previous studies [13, 42] showed that less recent releases of a package tend to be more vulnerable, we expect to find more vulnerable dependencies if we would study all package releases rather than only the latest one. The results of RQ_3^a revealed that only a small set of vulnerable packages is responsible for a large number of vulnerable dependencies. Moreover, RQ_4^a showed that these vulnerable dependencies can often be found deep in the dependency tree, making it difficult for dependents to cope with them. Several very popular packages that are used by many dependents have been affected by vulnerabilities, leading to a vulnerability exposure in thousands of transitively dependent packages.

> **Recommendation:** *Package maintainers should inspect not only direct, but also indirect dependencies, since vulnerabilities are often found deep in the dependency chain.*

> **Challenge:** *How can the community help to secure popular packages, that are present in many dependency chains?*

Another factor behind this vulnerability exposure is that most of the reported vulnerabilities do not have a lower bound on affected releases. They are found to affect all package releases before the version in which the vulnerability was fixed, yielding a wide range of vulnerable package releases. For

example, according to Snyk, the critical vulnerability CVE-2019-10744 affects all releases before version 4.17.12 of the popular package lodash [28]. However, maintainers of `lodash` showed vigilance since this vulnerability was fixed exactly 13 days after its disclosure. Users of package distributions therefore have an important responsibility in keeping the ecosystem in a healthy shape by keeping their own dependencies up to date. However, managers of package distributions can share this responsibility by raising awareness of the importance of keeping dependencies to popular packages up to date. Maintainers of those popular packages can participate as well by informing their dependents whenever vulnerabilities are discovered and fixes become available in newer versions, and maintainers of dependents that should actually update their dependencies to those fixed releases.

> **Recommendation:** *Package developers should be aware that disclosed vulnerabilities frequently affect all previous releases of a package. This kind of vulnerabilities can be prioritized since all dependents of the package will be impacted.*

> **Challenge:** *How often should one inspect previous releases for a vulnerability that is disclosed in a recent release, and how accurate is this information in vulnerability report databases? [27, 43, 44]*

The analysis of RQ_3^b revealed that external projects hosted on GitHub, containing software that is not distributed via the package distributions, are exposed to high numbers of vulnerabilities coming from transitive dependencies caused by a small set of vulnerable packages. Package dependents should rely on tools like `npm audit` to run security checks and be warned about such vulnerable dependencies. We also noticed that many npm vulnerabilities are coming from duplicated dependencies across the dependency tree. Dependents should reduce the number of duplicated npm dependencies – especially vulnerable ones – by sharing the common dependencies using commands like `npm dedupe` [29] available for npm. In a similar vein, dependents should check for bloated vulnerabilities which are not necessary to build or run the dependent software and remove them. This may reduce the number of vulnerable dependencies and their vulnerabilities. In fact, previous studies about dependencies in Maven showed that 2.7% and 57% of direct and transitive dependencies of Maven libraries are bloated [45].

> **Recommendation:** *npm dependents should reduce the relatively large number of vulnerable indirect dependencies by eliminating duplicate package releases.*

We found that about 40% of the packages and 70% of the external projects have at least one vulnerable transitive dependency. This does not necessarily mean that the dependents are at risk and affected by the vulnerabilities of their dependencies. Many of the vulnerabilities will only affect the functionalities

---

[28] https://nvd.nist.gov/vuln/detail/cve-2019-10744
[29] https://docs.npmjs.com/cli/v7/commands/npm-dedupe

that are not actually used by the dependent [46]. When fixing vulnerabilities coming from dependencies, developers should prioritize vulnerabilities by identifying the ones that are actually exposing their software by distinguishing between effective and ineffective functionalities used from dependencies. However, there are many cases where it is not necessary to use the vulnerable dependency to be affected by its vulnerability, e.g., Prototype Pollution [30] and Malicious Package vulnerabilities [38]. For this reason, it is important to know about all vulnerabilities coming from both direct and indirect dependencies and then decide whether they are exposing the dependent software or not.

> **Recommendation:** *Indirect dependencies come with a high number of vulnerabilities, especially in npm. Dependents should reduce the number and depth of their indirect dependencies or monitor them alongside the direct ones.*

Verifying direct dependencies for effective functionalities could be done by the developers of software dependents, while this could be difficult to do with transitive dependencies. Developers of package and project dependents may rely on Software Composition Analysis (SCA) tools to check transitive dependencies for vulnerabilities affecting their used dependency functionalities. They can also combine SCA tools to identify and mitigate the maximum number of vulnerabilities. In fact, previous studies have shown that SCA tools vary in their vulnerability reporting [47].

> **Recommendation:** *Package dependents should rely on available tools to run security checks and be warned about vulnerable dependencies and their fixes.*

> **Challenge:** *Should developers be warned about all disclosed vulnerabilities of their dependencies or only about vulnerabilities affecting the functionalities they use? [17, 23, 46].*

RQ_5 highlighted that most of the vulnerabilities exposing dependent packages and projects have a known fix, implying that by updating their vulnerable dependencies, dependents might avoid those vulnerabilities. To keep informed about new releases that may include vulnerability fixes, dependents can rely on tools like Dependabot, which creates pull requests to update outdated and vulnerable direct dependencies to newer and patched releases. It is the responsibility of package maintainers to keep their own dependencies up to date so dependent packages and projects can have secure transitive dependencies. To do so, package dependents can rely on permissive *semver* constraints [6] when specifying the dependencies to rely on in manifests like *package.json* and *Gemfile*. This will ensure that dependencies get updated automatically. For project maintainers, it is possible to use commands like npm update PACKAGE --depth= DEPTH to update their transitive dependencies and then lock their

---

[30] https://www.whitesourcesoftware.com/resources/blog/
prototype-pollution-vulnerabilities/

dependencies using lockfiles like *package-lock.json* [31] and *Gemfile.lock* [32]. An other option could be to update frequently. There is a big difference between updating a dependency with a security patch with a change of few lines of code versus several years worth of code [48]. The analysis of RQ_5 showed that around one in three vulnerabilities might be avoided if every dependent updated its dependencies to a newer minor or patch increment of the major release it is already using. This means that many vulnerable dependencies can be fixed just by modifying the dependency constraints to accept new minor and patch releases. On the other hand, for vulnerable dependencies for which the fix requires a major update, maintainers of dependent packages and projects should be careful as major dependency updates might introduce breaking changes. Therefore, maintainers of required packages should not only provide fixes in new major releases but should also attempt to provide these fixes in older major releases via backports [49]. This way, even dependents that need to stick to older major releases could still benefit from the backported vulnerability fix. Perhaps popular packages with a high package centrality [50] can benefit from community support in bringing such backports to older major releases.

> **Recommendation:** *Packages and external GitHub projects should invest more efforts in updating their dependencies.*

In addition, in RQ_2 we found a considerable proportion of vulnerabilities that have been fixed in minor and major releases. We think that package maintainers should try as much as possible to fix their vulnerabilities in patch updates or at least in backward compatible releases and follow semantic versioning. If a new package release incorporates breaking changes, then the major version number should be incremented. This will help developers to know whether they can update their vulnerable dependencies or not. Previous studies showed that developers are hesitant to update their vulnerable dependencies because they are afraid that the new releases not only include security fixes but also bundle them with functional changes. This hinders adoption due to lack of resources to fix functional breaking changes [39, 51].

> **Recommendation:** *Vulnerability fixing updates should not include breaking changes. If package maintainers can rely on semantic versioning, they will be able to characterize their package updates, while their dependents will be able to decide whether they want to update their dependency or not.*

> **Challenge:** *Is it always possible to incorporate vulnerability fixes within patch updates?*

5.3 Comparing package distributions

---

[31] https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json
[32] https://bundler.io/rationale.html

Our analysis revealed many differences between npm and RubyGems. npm has more reported vulnerabilities than RubyGems, and npm projects exposed to vulnerabilities have more dependencies. On the other hand, RubyGems has higher proportions of vulnerable dependencies than npm. It is likely that the security efforts undertaken by npm –such as its integrated dependency auditing tools– are gradually making npm more secure. Still, tooling could be improved further, as most of the existing tooling relies on dependency metadata alone, i.e., available security monitoring tools only rely on dependency information extracted from manifests like *package.json* and *Gemfile* to detect vulnerabilities. Usage-based vulnerability detection tools [23, 25] should emerge to help developers in identifying which dependency vulnerabilities can actually be exploited.

> **Lesson learned:** *More effort is needed to better secure open source package distributions. All parties can help including package managers, security experts, communities and developers.*

However, presence or absence of tooling is by no means the only factor that influences the proportion of vulnerable packages and vulnerable dependencies in a packaging ecosystem. Decan et al. [1] have shown important differences in the topological structure and evolution of package dependency networks, whereas Bogart et al. [33] have shown that each ecosystem uses different practices and policies of their communities. All these factors are likely to play a role in vulnerability management.

While this paper focused on npm and RubyGems, Alfadel et al. [26] studied the PyPI package distribution. Through an analysis of 550 vulnerability reports affecting 252 Python packages they studied the time to disclose and fix vulnerabilities in the PyPI package distribution. There are some similarities between their results and our own observations for npm and RubyGems. For example, the number of vulnerabilities found in PyPI increases over time and the majority of those vulnerabilities are of medium or high severity. The most prevalent vulnerabilities in PyPI are *Cross-Site-Scripting (XSS)* and *Denial of Service (DoS)*, which is similar to what we found for RubyGems. Vulnerabilities in PyPI are disclosed after a median of 37 months, which is similar to what we found for npm. On the other hand, vulnerabilities in PyPI seem to take longer to be fixed than those in npm and RubyGems. Since Alfadel et al. [26] did not study the exposure of dependent packages to vulnerabilities, we cannot compare PyPI to npm and RubyGems on this aspect.

> **Challenge:** *What are the main factors in a packaging ecosystem that play a role in its security management?*

6 Threats to validity

The empirical nature of our research exposes it to several threats to validity. We present them here, following the classification and recommendations of [52].

The main threat to *construct validity* comes from imprecision or incompleteness of the data sources we used to identify vulnerabilities and their affected and exposed packages. We assumed that the Snyk vulnerability database represents a sound and complete list of vulnerability reports for third-party packages. This may have led to an underestimation since some vulnerabilities may not have been disclosed yet and are therefore missing from the database. Another data source that we relied on is *libraries.io*. While there is no guarantee that this dataset is complete (e.g., there may be missing package releases), we did not observe any missing data during a manual inspection of the dataset. Considering the full set of packages ever released also constitutes a threat to validity since some of them could have been removed from the package registry yet are still referenced in *libraries.io*. To mitigate this threat we sanitised the dataset by excluding a number of npm packages that were removed from npm. Examples are the wowdude-x [33] and neat-x packages [34]. The only purpose of these packages was to bundle a large set of run-time dependencies.

As explained in Section 3.1, the vulnerability severity labels are extracted from Snyk. This data source has its own way of computing the severity of vulnerabilities. Relying on another data source might have led to different vulnerability severity results. To understand how different severity labels in Snyk compare to other sources, we extracted the severity labels for all vulnerabilities with a CVE from NVD [35]. For all 1,487 vulnerabilities (55% of the entire dataset) that we found with a CVE ID in RQ_1, only 1,227 have a severity label in NVD. 792 (64.55%) of them have the same severity in both Snyk and NVD. The heatmap of Figure 24 shows the proportion of vulnerabilities with different severity labels in Snyk and NVD. We observe that 21.68% of the vulnerabilities have higher severity in NVD than in Snyk, while 13.77% of the vulnerabilities are more severe in Snyk than in NVD. This confirms that the findings could differ if another vulnerability data source were to be used. For example, the observed vulnerabilities exposing package and project dependents could be more severe than what is reported in this paper. Nevertheless, since many vulnerabilities are not even reported by NVD, we chose for the pragmatic option to rely on the more complete dataset of Snyk.

We also tried to verify severity labels given by CNAs (i.e., vulnerability reporters in NVD), but only 42 vulnerabilities contained such labels.

Another threat to construct validity stems from the fact that we found some packages whose first vulnerable releases were never distributed via the package manager. For example, the cross-site scripting (XSS) vulnerability type [36] affects all releases before version 0.4.0 of the npm package wysihtml. Those releases, including version 0.4.0, were never actually distributed through npm [37] even though they are present on the package's GitHub repository [38].

---

[33] https://libraries.io/search?q=wowdude

[34] https://libraries.io/npm/neat-106

[35] https://nvd.nist.gov/

[36] https://snyk.io/vuln/npm:wysihtml:20121229

[37] https://www.npmjs.com/package/wysihtml
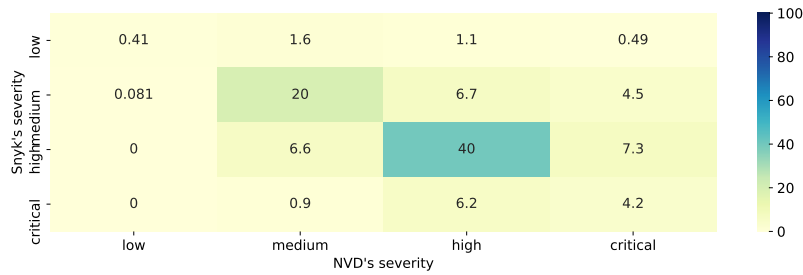
[38] https://github.com/Voog/wysihtml/tags?after=0.4.0

Fig. 24 Heatmap of the proportion of vulnerabilities that have the same or different severity labels in Snyk and NVD.

Since our study only focused on vulnerabilities affecting package releases contained in the *libraries.io* dataset, our results might underestimate the exact time needed to disclose a vulnerability.

We noticed that many of the reported vulnerabilities affect *all* package releases before the version in which the vulnerability was fixed, leading to a large number of vulnerable package releases. To ascertain that indeed all package releases before the fix are affected, we contacted the Snyk security team. They confirmed that they carry out manual inspections of the affected releases before declaring the set of vulnerable releases in their vulnerability report. However, since many vulnerabilities were not analyzed by Snyk but only copied from other security trackers, we decided to manually inspect 50 additional vulnerability reports (25 from npm and 25 from RubyGems), randomly chosen from the whole set of vulnerability reports that do not have a lower bound on the range of affected package releases. For each vulnerability, we manually checked its fixing commit and then verified whether the vulnerability was present in the first initial release [53]. Among these 50 cases, we found 8 false positives (4 for each ecosystem), i.e., vulnerability reports in which the first initial version is claimed to be vulnerable, while it is not. This corresponds to a confidence interval of 0.81574 ± 0.10356 using Agresti-Coull's method [54] with a confidence level of 95%. This manual verification implies that our results overestimate the actual number of packages and package releases affected by reported vulnerabilities.

To construct our dataset of vulnerabilities to study in $RQ_{1,2}$, we have removed vulnerabilities affecting inactive packages and vulnerabilities that have been disclosed before 2017-04-17. Considering all vulnerabilities of all packages (i.e., active and inactive) in the analysis might produce different outcomes.

As a threat to *internal validity*, when studying external GitHub projects that are not distributed via the package managers, we only focused on those GitHub projects explicitly mentioned in the *libraries.io* dataset. These projects were the most popular ones in terms of number of stars. While one may argue that this sample is not representative of all external projects depending on npm or RubyGems packages, the selected projects do have 90% of the

total number of stars attributed to all possible project candidates available in the *libraries.io* dataset. We therefore consider the chosen set of GitHub projects to be representative for most GitHub projects that depend on npm and RubyGems packages.

As a threat to *conclusion validity*, we used metadata to identify vulnerable dependencies. This identification approach assumes that the metadata associated with the used dependencies (e.g., package, version) and vulnerability descriptions (e.g., affected package, list of affected versions) are always accurate. These metadata are used to map each package onto a list of known vulnerabilities that affect it. However, dependents that rely on a vulnerable package might only access functionality of the dependent package that is not affected by the vulnerability. Therefore, our results present an overestimation of the actual risk. Still, we believe in the importance of signalling maintainers of exposed dependents that they are relying on vulnerable dependencies. It will be the responsibility of those maintainers to decide whether they are actually accessing vulnerable code, and to get rid of the vulnerable dependency if this happens to be the case.

As a threat to *external validity*, our findings do not generalise to other package distributions (e.g., Maven Central, CRAN, Cargo, PyPI). However, the design of our study can easily be replicated for other package distributions that are known to recommend *semver* practices.

Another threat to *external validity* stems from the fact that we relied on the manifests *Gemfile* and *packages.json* instead of the lockfiles *Gemfile.lock* and *package-lock.json* when extracting dependencies used in GitHub projects. The former are the default manifests that are always present in an npm or RubyGems project and in which dependencies with their permissive or restrictive constraints are declared, while lockfiles list the specific releases of all dependencies that should be selected to replicate the deployment of the project. We do not know whether our results would remain the same when considering the dependencies specified in lockfiles rather than the ones specified in the manifest of each package. A manifest expresses (through dependency constraints) the set of releases that could be selected when the package is installed through the package manager. The latter always select the highest available release satisfying the constraints. As such, it may be the case that the specific releases pinned in a lockfile do not correspond to the ones that will be selected by the package manager, potentially leading to a different exposure to vulnerabilities. Although the use of a lockfile allows maintainers to explicitly select a non-vulnerable release of a package that they know to be vulnerable, the lockfile prevents from automatically benefiting from a fix in case the selected, pinned release is vulnerable. On the other hand, nothing prevents the maintainer from excluding vulnerable releases through appropriate dependency constraints in the package manifest, while still allowing future security patches to be selected as soon as they become available.

7 Conclusion

This paper quantitatively analysed and compared how security vulnerabilities are treated in npm and RubyGems, two popular package distributions known to recommend the practice of semantic versioning. Relying on the Snyk vulnerability database, we studied 2,786 vulnerabilities which affect 1,993 packages directly. We observed that the number of reported vulnerabilities is increasing exponentially in npm, and linearly in RubyGems. Most of the reported vulnerabilities were of medium or high severity. We observed that malicious packages occur much more frequently in npm.

We analyzed the time needed to disclose and fix vulnerabilities and we found that half of the studied vulnerabilities needed more than two years to be disclosed, and more than four years to be fixed. Maintainers of reusable packages should therefore invest more effort in inspecting their packages for undiscovered vulnerabilities, especially if those packages have a lot of dependents that may get exposed to these vulnerabilities directly or indirectly. Better tooling should be developed to help developers in looking for undiscovered vulnerabilities and fixing them.

By analysing the impact of vulnerable packages on dependent packages and dependent external projects (hosted on GitHub), we observed that vulnerabilities can be found deep in the packages' and projects' dependency trees. Around one out of three packages and two out of three external projects are exposed to vulnerabilities coming from indirect dependencies. The most prevalent vulnerability type that affects npm dependencies is *Prototype Pollution*, while for RubyGems dependencies it is *Denial of Service*.

An important observation is that most of the vulnerabilities affecting dependencies of packages and external projects have known fixes (in the form of more recent package releases that are no longer vulnerable). Maintainers of dependents could therefore invest more effort in checking their exposure to vulnerable dependencies, and in updating their outdated dependencies in order to reduce the number of dependency vulnerabilities. In contrast, the majority of the outdated and vulnerable dependencies need to be updated to a new major release to avoid the vulnerabilities. We found that around one out of three vulnerabilities affecting direct dependencies of packages and external GitHub projects, might be avoided by only making backward compatible dependency updates.

References

1. Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, Feb 2019.
2. Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories*, 2018.
3. Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS Symposium*, 2017.
4. Snyk. The state of open source security. https://snyk.io/wp-content/uploads/The-State-of-Open-Source-2017.pdf, 2017. accessed: 10/06/2021.
5. OWASP. Owasp top ten web application security risks. https://owasp.org/www-project-top-ten/, 2017. accessed: 24/04/2021.
6. Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019.
7. Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 351–361. IEEE, 2016.
8. Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.
9. Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 2–12. IEEE, 2017.
10. Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Int'l Symp. Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016.
11. Jesus M Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. Technical lag in software compilations: Measuring how outdated a software deployment is. In *IFIP International Conference on Open Source Systems*, pages 182–192. Springer, 2017.
12. Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.
13. Ahmed Zerouali, Tom Mens, Alexandre Decan, Jesus Gonzalez-Barahona, and Gregorio Robles. A multi-dimensional analysis of technical lag in Debian-based Docker images. *Empirical Software Engineering*, 26(2):1–45, 2021.
14. Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *Int'l Conf. Software Maintenance and Evolution*, pages 404–414. IEEE, September 2018.
15. N. H Pham, T. T Nguyen, H. A Nguyen, X. Wang, A. T Nguyen, and T. N Nguyen. Detecting recurring and similar software vulnerabilities. In *Int'l Conf. Software Engineering*, pages 227–230, 2010.
16. Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2010.
17. Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018.
18. Jukka Ruohonen. An empirical analysis of vulnerabilities in Python packages for web applications. In *International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 25–30. IEEE, 2018.
19. Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in Node. js applications: How far are we? In *International Conference on Automated Software Engineering (ASE)*, pages 1199–1203. IEEE, 2020.

20. Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *International Conference on Software Engineering*, pages 109–118. IEEE Press, 2015.

21. Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 2020.

22. Fabio Massacci and Ivan Pashchenko. Technical leverage in a software ecosystem: Development opportunities and security risks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1386–1397. IEEE, 2021.

23. Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

24. Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *International Conference on Software Maintenance and Evolution*, pages 559–563. IEEE, 2018.

25. Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, pages 995–1010, 2019.

26. Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in Python packages. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021.

27. Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

28. Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26, 2021.

29. Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020.

30. Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 491–501. IEEE, Feb 2019.

31. Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices? In *Annual Meeting of the Southern Association for Institutional Research*, 2006.

32. John P. Klein and Melvin L. Moeschberger. *Survival Analysis: Techniques for Censored and Truncated Data*. Springer, 2013.

33. Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Trans. Softw. Eng. Methodol.*, 30(4), July 2021.

34. Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org/, 2013.

35. Thomas Maillart, Mingyi Zhao, Jens Grossklags, and John Chuang. Given enough eyeballs, all bugs are shallow? revisiting eric raymond with bug bounty programs. *Journal of Cybersecurity*, 3(2):81–90, 2017.

36. Nikolaos Alexopoulos, Andrew Meneely, Dorian Arnouts, and Max Mühlhäuser. Who are vulnerability reporters? a large-scale empirical study on floss. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021.

37. Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020.

38. Alex Birsan. Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610. accessed: 07/05/2021.

39. Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. Up2dep: Android tool support to fix insecure code dependencies. In *Annual Computer Security Applications Conference*, pages 263–276, 2020.

40. Raúl Aranovich, Muting Wu, Dian Yu, Katya Katsy, Benyamin Ahmadnia, Matthew Bishop, Vladimir Filkov, and Kenji Sagae. Beyond nvd: Cybersecurity meets the semantic web. 2021.

41. Ahmed Zerouali, Tom Mens, and Coen De Roover. On the usage of JavaScript, Python and Ruby packages in Docker Hub images. *Science of Computer Programming*, page 102653, 2021.

42. J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *International Conference on Software Engineering*, pages 109–118, 2015.

43. Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. A screening test for disclosed vulnerabilities in foss components. *IEEE Transactions on Software Engineering*, 45(10):945–966, 2018.

44. Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering*, 21(6):2268–2297, 2016.

45. César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):1–44, 2021.

46. Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 2020.

47. Nasif Imtiaz, Seaver Thorne, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. *arXiv preprint arXiv:2108.12078*, 2021.

48. Ahmed Zerouali. *A Measurement Framework for Analyzing Technical Lag in Open-Source Software Ecosystems*. PhD thesis, University of Mons, September 2019.

49. Alexandre Decan, Tom Mens, Ahmed Zerouali, and Coen De Roover. Back to the past–analysing backporting practices in package dependency networks. *IEEE Transactions on Software Engineering*, 2021.

50. Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Ayme Saied, and Bram Adams. Towards using package centrality trend to identify packages in decline. *arXiv preprint arXiv:2107.10168*, 2021.

51. Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531, 2020.

52. C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

53. Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 6, pages 10–5555, 2006.

54. Alan Agresti and Brent A Coull. Approximate is better than exact for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.