Delft University of Technology

Structure and Evolution of Package Dependency Networks

Kikas, Riivo; Gousios, Georgios; Dumas, Marlon; Pfahl, Dietmar

DOI
10.1109/MSR.2017.55

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017

Citation (APA)
Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017). Structure and Evolution of Package Dependency Networks. In R. Bilof (Ed.), *Proceedings - 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, MSR 2017* (pp. 102-112). IEEE. https://doi.org/10.1109/MSR.2017.55

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Structure and Evolution of Package Dependency Networks

Riivo Kikas
University of Tartu
Tartu, Estonia
riivokik@ut.ee

Georgios Gousios
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

Marlon Dumas
University of Tartu
Tartu, Estonia
marlon.dumas@ut.ee

Dietmar Pfahl
University of Tartu
Tartu, Estonia
dietmar.pfahl@ut.ee

***Abstract*—Software developers often include available open-source software packages into their projects to minimize redundant effort. However, adding a package to a project can also introduce risks, which can propagate through multiple levels of dependencies. Currently, not much is known about the structure of open-source package ecosystems of popular programming languages and the extent to which transitive bug propagation is possible. This paper analyzes the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems. The reported results reveal significant differences across language ecosystems. The results indicate that the number of transitive dependencies for JavaScript has grown 60% over the last year, suggesting that developers should look more carefully into their dependencies to understand what exactly is included. The study also reveals that vulnerability to a removal of the most popular package is increasing, yet most other packages have a decreasing impact on vulnerability. The findings of this study can inform the development of dependency management tools.**

I. Introduction

Open-source software development has resulted in an abundance of freely available software packages (libraries) that can be used as building blocks for new projects. Usage of existing libraries can increase velocity and reduce the cost of a software project [1]. However, introducing third-party libraries makes a project dependent on them. Dependencies need to be kept up-do-date to prevent exposure to vulnerabilities and bugs [2]. At the same time, bugs can also originate through transitive dependencies [3]. Developers might not have an overview of all the transitive dependencies as they did not include them themselves. Updating dependencies also entails risks, as new versions may break existing functionality or API correctness [4].

In March 2016, a single JavaScript package, *left-pad* was removed from the central JavaScript package repository `npm`. The removal caused issues also for projects that depended on it indirectly through transitive dependencies [5]. The *left-pad* incident illustrates the hidden risks of relying on publicly available packages. A problem with a single package can propagate through multiple levels of dependencies.

Over the years, a number of studies have addressed the question of how to develop maintainable software and how to cope with software evolution challenges [6], [7]. On the other hand, dependency management practices have received little attention, despite being a crucial part of almost all software projects. A recent study of the JavaScript package ecosystem [8] revealed that dependency requirement specifications using semantic versioning with flexible version constraints (e.g. the latest version) are widely used. This practice often leads to a new version of dependency being used implicitly every time a project is built. Another study of Maven packages [4] revealed that the semantic versioning scheme is not always used properly and breaking changes are also introduced in minor version releases. Implicit updates combined with non-conforming API changes can introduce unexpected behavior or software defects. Considering the *left-pad* incident and the lack of studies on dependency management, we seek to enhance the understanding of the state of dependency update practices and the structure of dependency networks.

More recently, data has become available from package repositories and GitHub repositories that enable us to study the package ecosystems of different programming languages. Having access both to packages that are published in a central repository and applications using these can give us an idea how often dependencies are updated and what is the state of the dependency ecosystem.

In this work, we take a novel network-based approach for studying dependency networks of JavaScript, Ruby, and Rust. We use data from package repositories and a subset of GitHub projects. We compose a network of projects based on dependency relations to understand how the dependency network evolves and how susceptible it is to a removal of a random project. We show that dependency networks of popular languages such as JavaScript and Ruby are growing and have at least one single package whose removal can affect more than 30% of projects in the ecosystem.

The goal of this work is to study the state of current dependency networks, to understand their characteristics, and to reason about their future evolution. We have formulated the following research questions to guide our research:

RQ1: *What are the static characteristics of package dependency networks?*

RQ2: *How do package dependency networks evolve?*

RQ3: *How vulnerable are package dependency networks to a removal of a random project?*

Answers to these questions can help to quantify the state of the ecosystems, give an overview of the trends in dependency management, and can inform the development of improved dependency management tools.

IEEE: © 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

II. Background and Related work

In this section, we explain the terminology and give an overview of the related work.

A. Terminology

Current work analyses dependencies among software projects. We distinguish between two types of software projects: packages and applications. We define *packages* as a reusable code or set of components that can be included in other applications by using dependency management tools. Packages are published in *repositories* and are available to everyone. *Applications* are projects that make use of packages, are not published as a package and thus can not be used in other projects as a dependency. Packages and applications can have multiple versions distinguished by version numbers.

A package can depend on another package. If package A depends on package B we say that A has a *dependency* (A is a dependent of B) and B has a *reverse dependency* (B has a *dependent*). Applications can have dependencies but since they are not published as a reusable package they cannot have reverse dependencies. A project has a *direct dependency* if a package on which the project depends, and which it needs to be built, is directly included in the project. A project can have a *transitive dependencies* on packages that are not needed for the project itself but needed for the direct dependencies included in the project to work. Transitive dependencies can be included through multiple levels of dependencies.

A *dependency network* is composed of packages, applications, and dependency relations between them. An *ecosystem* is the set of packages and applications involved in the dependency network.

B. Related work

The related work deals with analyzing dependency networks, analyzing risks associated with dependency usage, and API stability in libraries.

Dependency networks. Network-based analysis of programming language dependency networks has emerged recently. A first large-scale analysis of the `npm` ecosystems was carried out by Wittern et al. [8]. Their analysis concludes that JavaScript is a striving ecosystem because of frequent releases of new and existing packages. They use GitHub applications only to study version numbering practices and state that there is a prevalence of flexible (not exact) version number specifications. They conclude that usage of flexible version constraints should result in the immediate adoption of a new release.

Decan et al. [9] analyze topologies of `npm`, `PyPI`, and `CRAN` and find that there are differences across ecosystems, e.g., the `PyPI` is less interconnected than `npm`. They state that analysis results are not generalizable from one ecosystem to another. Their follow-up work [10] focusing on dependency version specification usage analysis, points out that current tools and versioning schemes can introduce resiliency issues to the ecosystem.

German et al. [11] study packages in the R ecosystem. They find that most packages do not have any dependencies, but popular ones are more likely to have. They also find that growth of the ecosystem comes from user-submitted packages, and it takes a longer time to build a community around user-submitted packages than around core contributed packages. Another analysis of the R ecosystem [12] studies dependency resolution in R packages finds that lack of dependency constraints in package descriptions and backward incompatible changes often break dependencies. As community contributed packages are hosted on GitHub, there is no way to resolve dependencies among GitHub packages, and therefore a small amount of GitHub packages cannot be automatically installed.

Bogart et al. [13] interview seven maintainers of R and `npm` packages to understand how dependencies are maintained. They find that developers are not aware of the stability of packages in the ecosystems and make changes on ad-hoc principles. In a follow-up work [14], they found that `npm`, `CRAN` and Eclipse ecosystems differ substantially in their practices about resolving API breaking conflicts and expectations toward change.

Dependency management. A study of dependency management process in Apache projects [15] found that if the number of projects in the ecosystem grows linearly, the dependencies among them grow exponentially. Bavota et al. [16] also find that new releases often do not contain updates to their dependencies. Dependencies are updated only if major new features or bug fixes are released for the dependencies. Kula et al. [17] measure latency to adopt new versions among a sample of Java projects that use Maven. They conclude that over time, the maintainers become more trusting and update faster, although no reason is known for this behavior. Cox et al. [2] measure dependency freshness in 75 different closed source projects of 30 different vendors. Their findings indicate that projects with low dependency freshness are more than four times likely to include a security vulnerability.

Besides programming language ecosystems, previous research studied the Debian package ecosystem, how to resolve strong dependencies in it, and how to improve the planning of dependency changes [18], [19], [20], [21].

Vulnerabilities. Hejderup [3] studies vulnerability spreading across `npm` packages. He uses information about known vulnerabilities, tracks how long it takes for projects to update from a vulnerable version and shows that vulnerabilities can affect projects through dependencies. He also observes that some of the projects have a discussion in the issue tracker about vulnerable dependencies that need updating. Through qualitative analysis, he finds that developers were not aware of the vulnerabilities and the risk of breaking functionality is what holds back blindly updating vulnerabilities.

Cadariu et al. [22] propose a tool to track known vulnerabilities in Java projects. They conduct a case study on private Dutch enterprise projects and find that 54 out of 75 projects use at least 1 (and up to 7) vulnerable dependency.

Synthesis of related work. The three research questions proposed in this paper have received attention in the context of

existing research. There are similarities with existing research, but none of them fully covers the scope and problem of this paper. Wittern et al. [8] and Decan et al. look at the network topologies for `npm`, `PyPI` (Python) and `CRAN` (R). Compared to [8], our work considers the network analysis in more detail and includes applications in the network analysis step. Compared to [9], [10] we also focus on the network evolution and outline more accurate dependency network model. Hejdreup [3] studies vulnerability spreading among `npm` projects. Our work analyses the whole ecosystem and includes evolution analysis to study if over time such vulnerabilities will become less or more likely.

III. Research questions

We have formulated three research questions to guide our research. The overall motivation is to analyze structure and evolution of dependency networks to get insight into current dependency usage and possible issues. Next, we explain the motivation behind each research questions in more detail.

RQ1 (Structure). Currently, not much is known about the static properties and topologies of programming language package ecosystems. For example, we know to what extent dependencies are used in packages only [8], [9]. However, we do not know if there are differences in dependency usages across published packages and applications? Modern package managers allow different conventions for specifying dependency version numbers such as exact version or version range. However, we do not know what the most popular way of specifying dependencies is. Answers to these questions enable us to understand the current state of dependency ecosystem and would be the starting point for analyzing ecosystem evolution.

RQ2 (Evolution). Software projects can add new dependencies and update existing dependencies. Changes in dependencies in a new release of a single package will also be reflected in the overall dependency network. Studying the dependency network evolution since its creation can explain the current state and also provide knowledge to reason and make predictions about its future evolution. Need for such analysis was outlined by respondents to a recent survey on software ecosystems challenges [23]. One of the answers given by respondents stated: *if an ecosystem is not able to evolve quickly it is going to die* [23]. Similarly, our goal is to understand the current evolution state of the studied ecosystems and analyze if they are growing or stabilizing.

RQ3 (Vulnerability). When selecting a package use, several factors are important besides the functionality it provides. Developers ideally would like to be sure that the package quality is good, it is maintained, and is trustworthy. As these properties are not explicitly visible, developers might end up using packages of varying quality. For example, if an attacker publishes packages with names very similar to the names of popular packages, developers making a typo could end up using them unwillingly [24]. The *left-pad* incident happened because the developer decided to remove the package. How vulnerable are ecosystems to such scenarios? We define vulnerability as the number of projects that are affected if we remove a package or a specific version of it. This scenario also helps to estimate what fraction of the dependency network is impacted if a package contains a bug. Such information could be incorporated in measuring package importance with regards to vulnerability in an ecosystem.

IV. Method

In the following section, we describe the data collection method, preprocessing steps and our approach for modeling dependency networks using graphs.

A. Context

In this work, we study three package ecosystems for the programming languages, i.e., JavaScript, Ruby, and Rust. We chose these three languages as the majority of their packages and applications are hosted on GitHub. These languages have central repositories for hosting packages, namely `npm`, `RubyGems`, and `Crates`. Developers specify required packages in their project's dependency files (`package.json`, `Gemfile`, `Cargo.toml`) and packages are retrieved by the dependency manager (*npm*, *Bundler*, *Cargo*). The packages contain source code and developers can use functionality from packages in their project. In addition to packages, we study applications download from GitHub. By adding applications, we can analyze package usage from the end-user viewpoint.

We chose to study JavaScript and Ruby, both dynamically typed languages popular choices for among web application development. Rust, on the other hand, is a multi-paradigm language that supports static typing primarily meant for systems programming. JavaScript and Ruby have been used since the 1990s and their corresponding central package managers appeared in 2010 and 2004. Rust first appeared in 2010 and its central package management in 2014. Our analysis of JavaScript revolves around the packages used in the *node.js* environment and managed through `npm` tool, but also includes packages only needed for web development, such as front-end frameworks. JavaScript differs from the other languages used in the study as it supports multiple versions of a project in its dependency chains. For example, if package A depends on package B version 1.0 and package C version 2.0, and package B depends again on package C version 3.0, then `npm` downloads both versions of the package C. Rust and Ruby do not allow such scenario and a single version of package C is required. In practice, JavaScript developers can have more freedom in including dependencies, but Rust and Ruby developers need to make sure their dependencies do not conflict.

B. Data collection

We used multiple sources for composing the dataset. For JavaScript and Ruby, we downloaded the full list of packages, release dates, dependencies, and other relevant meta-data from their central repositories, `npm` and `RubyGems` respectively. For getting data from `npm`, we used the public API [25].

For RubyGems, we used a copy of their meta-data database available on-line [26].

Central repositories such as npm and RubyGems host only projects that are typically libraries, frameworks, command line applications or resource bundles for web development. We also include end user applications from GitHub in our study to understand the package usage in practice. We used the GHTorrent [27] database of March 2016 to select projects whose repository language identified by GitHub was either Rust, JavaScript or Ruby, were not forks, and the project GitHub repository did not appear in the npm or RubyGems hosted project list. After composing the initial list of projects, we made an HTTP request to every repository to check if it had a dependency file in the root folder of the latest revision. We only cloned repositories that had a dependency file present in the latest revision. For Rust, we cloned all projects listed in GHTorrent, but for JavaScript and Ruby, we only cloned those that either had at least one fork or at least one star, to minimize the number of projects to collect. We acknowledge that we were not trying to collect all the projects from GitHub.

Rust has a central repository called *Crates.io*, but the meta-data is not available from there in a structured machine-readable format. Therefore, for Rust, we only rely on the packages from GitHub by first selecting all Rust language projects from the GHTorrent database and then filtering out those that do not have a dependency file named *Cargo.toml*. The Rust data can be considered as a sample of the whole package universe of Cargo and additional applications written in Rust.

Data collection took place during April and May 2016. We collected the package repository data after collecting applications from GitHub. We excluded all updates and changes after April 2016, to get a comparable time scale for all ecosystems.

C. Parsing GitHub projects

The projects obtained from GitHub have their dependency information recorded in dependency files. To extract dependencies, we consider all revisions of the dependency files to recover the dependency history. We used the *git log* command to extract all changes to the dependency file. For accurate modeling, we had to know when each version of a project was released. JavaScript's *package.json* and Rust's *cargo.toml* provide explicit version information of the project. Ruby's dependency files (.gemspec and Gemfile) are written in Ruby code and sometimes the version number is expressed as a variable or read in from a file. This makes reading the exact version numbers hard, as there is no general pattern. Extracting this is therefore not feasible, as it would require manual inspection or executing the code. In cases we could not extract explicit version numbers, we used the time of the last modification of the dependency file. This only affects applications and does not impact the dependency network structure as they do not have dependents. The limitation of this approach is that there might be many more revisions than actual releases. If multiple revisions of a dependency file exist with the same version number, we use the latest revisions for the version. Developers might change contents of the file during development with the new version number already entered but after the release the contents will not change.

D. Resolving dependencies

When parsing dependency files, we encountered situations where some of the dependencies were not available. A dependency might not be available in a case where a single revision of a dependency file committed to the repository contained typos or incorrect version constraints, thus the dependency does not exist. We only kept those dependencies that we could match in the central repositories for JavaScript and Ruby. For Rust, we kept all dependencies we could match among the projects as we did not use official package repository data. If a dependency is specified as a reference to a git source code repository, we only kept this in the case of Rust projects and the repository was in the list of collected projects.

Dependency version constraints can be specified in different ways, for example as exact version, latest version or pattern based matching using the semantic versioning notation. A version number is typically written in the format of MAJOR.MINOR.PATCH. An increase in the MAJOR number denotes incompatible API changes, an increase in the MINOR number indicates an addition of backward compatible changes, and an increase in the PATCH number indicates a bug fix. A version requirement specification has specific notations for describing valid version. JavaScript and Rust support similar notation formats. To obtain *any* version or the *latest* version, the requirement should be specified as the wild-card (*) or with an explicit condition (≥ 0). The *tilde* operator (˜) matches the most recent MINOR version. For example ˜3.0.3 matches the highest version in the range $[3.0.3, 3.1)$, but will not match 3.1. The *caret* (ˆ) will select the most recent MAJOR version (the first number). For example, ˆ1.2.3 matches highest version in the range $[1.2.3, 2.0)$. Ruby does not support the *tilde* and the *caret* directly, but has something similar called the pessimistic operator, expressed by $\sim>$. For example, $\sim>$ 3.0.3 is equivalent to ˜3.0.3. Requirement $\sim>$ 1.1 is equivalent to ˆ1.2, i.e., matches the highest version in the range $[1.2.3, 2.0)$.

For network construction, we must be able to represent the state of dependencies as they were at the time a package was released or an application was committed to the repository. With inexact version requirements the actual version that might be included in the project might differ every time the project is built, as a more up-to-date version of a dependency that satisfies the requirements might have become available. We resolved all dependency version requirements to the version that would have been used when the package was released or a GitHub commit was made. Therefore, we knew when the release was made and also could trace back which packages and versions were available at that time. For JavaScript projects, we used the package *semver* to find for each dependency the highest version candidate available. For Ruby projects, we used *Gem* library code for finding the latest revision among

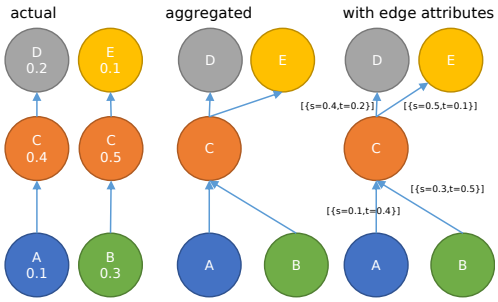


Fig. 1. Dependency network construction approaches

all matching candidates. For Rust, we implemented our own dependency resolution.

Dependency version resolution did not take into account transitive dependencies and possible version conflicts. We are aware that in practice, some other version might have been chosen. To resolve all dependencies we would have needed to re-implement the corresponding language dependency resolution algorithm because dependency management tools do not support resolving dependencies as they would have been resolved at any arbitrary time in the past.

E. Network construction

When modeling a system with a network, we need to define what nodes and edges represent. A straightforward approach to represent dependency relations in networks is to model projects as nodes, and directed edges between them denote dependencies between projects. The limitation of this solution is the lack of differentiation between project versions and thus this modeling approach could give misleading information about the network. Figure 1 illustrates three different approaches for network modeling. Packages A and B depend on different versions of C, but only C version 0.4 depends on D. The aggregated network model would indicate that package B is dependent on package D, which is not true. The number of different packages dependent on D is two (A and C) in the actual network, but aggregated version would give us three projects (C, A, and B). We also studied an approach where we annotate network edges with attributes. We have a list of pairs (source version, target version) for which this edge is valid. When traversing the network, we have to make sure that the target version on the edge that was used to access the node has a corresponding source node for taking the next step. For evolution analysis, both aggregated network and aggregated network with attributes are unsuitable. If we want to answer questions such as what is the number of transitive dependencies, we have to consider all project versions. A new release of a project can update its dependencies, thus increasing the connectivity in the aggregated graph. For example, all versions of the aggregated graphs (Figure 1) would give us that project C has two dependencies, however, at any time it only has one. Considering this, it might affect all the projects and we would get a more connected graph than the actual and the number of dependencies would not reflect the actual value.

We chose an approach where a node represents a specific project version. The edges denote dependency relations between specific versions (Figure 1, *actual*). With this modeling approach, we can have correct answers to queries such as how many different versions depend on a project and how many of these are unique projects.

In our analysis, we sometimes used the aggregated modeling version with edge attributes for some calculations. Whenever we did so, we mention it explicitly in the following. By analyzing the top 10 projects for JavaScript based on the number of dependencies, we confirmed that the aggregated network without edge attributes overestimates the dependency counts. Therefore we decided to use edge attribute information when analyzing dependency chains.

Or choice of dependency network model makes it hard to compare our results with existing research, which uses the aggregated network without attributes [8], [9]. Only Hejderup [3] uses a similar approach to our actual network. The difference is that Hejderup also keeps meta-nodes in the network to represent a project. Each meta node has links to the corresponding project's version node.

We only use projects that have at least one dependency or one reverse dependency. If a project does not have dependencies nor is a dependency for others, it does not appear in the network. As soon as a project adds a dependency, it will appear in the network. Due to this filtering, single isolated nodes can not exist in the network, while isolated clusters of connected nodes can.

We kept snapshots of the network for each month. A snapshot records how the ecosystem looked at the end of the corresponding month. Snapshots are cumulative, adding new projects and dependency links. Neither projects nor links are ever removed. All analyses involving the temporal evolution is also cumulative, i.e., if we calculate some property at a specific time, we calculate the property for all the projects published until that point.

We manually removed three projects from our dataset that appeared to be outliers. Two JavaScript applications and one Ruby package had been engineered so that they would contain all possible packages in their dependency file.

V. Results

A. Description of dependency networks (RQ1)

In this subsection, we describe the data sets and basic properties of the dependency networks.

1) Static properties: Table I lists basic properties of the language ecosystems used in our study, the number of projects initially collected, and different releases in the network.

We initially collected 11037 Rust, 339453 JavaScript, and 184919 Ruby projects. However, not all packages have dependencies or are used as a dependent, and therefore we exclude those projects in the network based analysis. The exclusion was based on the latest snapshot and included projects that never had any dependencies. The final dataset comprises 7978, 246670 and 147449 projects for Rust, JavaScript, and Ruby correspondingly.

TABLE I
SUMMARY OF DATASETS

	Projects in the network						Initially collected	
	Projects	Dependencies	Applications	Packages	Version	Version dependencies	Applications	Packages
Rust	7,978	25,144	0	7,978	22,105	66,055	0	11,037
JS	246,670	1,182,114	78,657	168,013	1,319,919	7,260,426	84,987	254,466
Ruby	147,449	776,061	69,544	77,905	1,231,480	10,747,737	62,133	122,786

TABLE II
MEAN (MEDIAN) NUMBER OF DEPENDENCIES AND DEPENDENTS

	Transitive		Direct	
	Dependencies	Dependents	Dependencies	Dependents
JS	54.6 (17)	15.5 (0)	5.5 (3)	1.3 (0)
Ruby	34.1 (22)	6.4 (0)	8.7 (4)	1.2 (0)
Rust	9.3 (5)	7.4 (0)	3.0 (2)	1.6 (0)

TABLE III
DISTRIBUTION OF VERSION UPDATE COUNTS

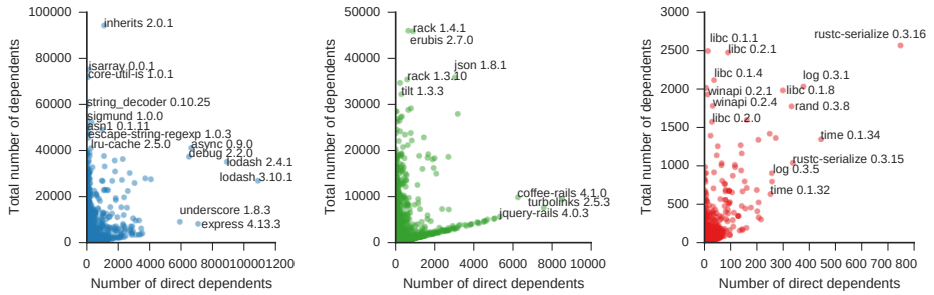
		Type	5p	median	mean	95p	max
explicit	JS	Package	1.0	1.0	1.06	1.0	69.0
		Application	1.0	1.0	1.37	3.0	253.0
	Ruby	Package	1.0	1.0	1.19	2.0	96.0
		Application	1.0	1.0	1.53	3.0	343.0
	Rust		1.0	1.0	1.19	2.0	62.0
implicit	JS	Package	1.0	1.0	1.11	2.0	66.0
		Application	1.0	1.0	1.70	4.0	280.0
	Ruby	Package	1.0	1.0	1.91	6.0	95.0
		Application	1.0	1.0	2.17	6.0	344.0
	Rust		1.0	1.0	1.44	4.0	63.0

Table II lists the number of dependencies and dependents (reverse dependencies) per release. Comparing languages, we see that Ruby projects have more direct dependencies on average (8.8) than JavaScript (5.5) and Rust (3.0). The differences in the number of direct dependents are smaller, i.e., 1.2, 1.3, and 1.6, respectively. However, we again see larger differences across transitive dependencies and transitive dependents (the average number of projects that depend on a project). JavaScript has the largest amount of transitive dependencies and dependents, 54.6 and 15.5, respectively. Ruby has 34.1 and 6.4, and Rust 9.3 and 7.4, respectively. The number of transitive dependents for JavaScript is almost two times larger than for other languages. Ruby has the highest average number of direct dependencies and Rust has the highest number of direct dependents. Differences in the amount dependencies across ecosystems reveal that the internal structures of dependency networks are different across ecosystems. JavaScript's large dependency count could possibly be attributed to tool support for different versions of a single package in dependencies.

2) Direct and transitive dependents: The *left-pad* incident had a high impact not because it was directly used in many projects but indirectly, through transitive dependencies. Figure 2 shows the relationship between the total number of dependents (direct and transitive dependents) and direct dependents for all projects at the beginning of April 2016. For all ecosystems, we can see that there exist projects that have a small amount of direct dependents (less than 100) and a large number of transitive dependents. We can see that this pattern is stronger in JavaScript (Figure 2b) and Ruby(Figure 2b) than for Rust. Ruby also exhibits a clear pattern with a package having an equal amount of direct and transitive dependents, meaning that a package is only involved in direct dependencies but not transitive ones.

3) Weakly connected components: Even though we limited our analysis to projects that have at least one dependency relation, the ecosystems under study are not fully connected for Rust and JavaScript. We calculated the number of weakly connected components in the dependency graphs for all languages. A weakly connected component in a directed graph is a subgraph where each node is connected with every other node in the subgraph via an undirected path. We observed the emergence of a giant weakly connected component in each of the three analyzed ecosystems. For Rust, JavaScript and Ruby, 96.14%, 98.2%, 100% of projects belong to the largest weakly connected component in the latest snapshot. Many real-world networks such as social networks exhibit the giant component property [28]. The remaining projects are part of components with a small number of projects. The existence of a giant component illustrates the fact that existing packages, even being developed by different developers, can be used together in applications. The ability to be used together makes the ecosystem valuable.

4) Dependency updates and constraint notation practices: We define explicit dependency version change as a manually changed version constraint for a dependency by a developer. The number of explicit changes is similar across ecosystems (Table III). The number of implicit changes denotes the number of times a dependency was resolved to a different version after each project release or dependency file commit, but without modifying the dependency requirement specification. An implicit update happens when dependencies are specified with flexible constraints, and there are newer versions released matching the constraints. The number of implicit updates has a larger variation across projects, with the highest mean of 2.17 for Ruby, 1.7 for Rust, and 1.44 for JavaScript. The mean number of implicit updates for the published packages are smaller than for applications, 1.91 and 1.1 for Ruby and JavaScript. We also see that the maximum values for both explicit and implicit updates are larger for applications which



(a) JS (b) Ruby (c) Rust

Fig. 2. Relationships between the number of direct dependents and total dependents in April 2016. A sample of project names are plotted.

can be explained by higher velocity in development as these projects do not have dependents. For both types of projects, packages and applications, Ruby seems to have higher update counts which can be explained by longer history. Another insight is that there are more implicit updates than explicit, indicating dependencies are updated more often than a developer would manually do this. In the following, we analyze more closely the popular ways of specifying dependency version requirements that lead to implicit updates.

Table IV lists the relative popularity of each requirement specification scheme in each ecosystem. Note that we distinguish here also between published packages and applications. The different ways to specify versions are: any or latest version (*any*), exact version (*exact*), explicitly specified version range such as $[2.0, 4)$, and one-sided ranges (*range*), the most recent minor version (*tilde*), the most recent major version (*caret*) or anything else, such as manually specified git version (*other*).

The dominating approaches for Rust version specifications are *exact* and *any* versions, used in 32% and 47.8% of the cases. Besides these, all different possible schemes for specification are used by developers. Rust developers prefer to specify specific versions or latest versions, as the ecosystem is growing.

Among the most popular approaches for JavaScript are the *caret*, *exact*, and the *tilde* notation. Exact versions are used only in 22% of the cases for different JavaScript projects. The difference between JavaScript GitHub projects and published packages is non-existent, whereas for Ruby, there are differences in the fraction of exact versions and range based specifications. We looked more into range usage in packages and found that a most of range specification in published packages comes from specifications that require larger than specific major version. We used Pearson's chi-squared test to confirm that Ruby's applications and published packages have different preferences in specifying version requirements ($\chi^2 = 884540$, $df = 5$, p-value $< 2.2 \cdot 10^{-16}$). Ruby also has the least amount of exact dependencies, which in turn can explain our observation of Ruby having the highest number of implicit version updates on average (Table III). In the end, we used Pearson's chi-squared on the full contingency table (Figure IV with absolute values) to confirm that dependency management preferences differ across languages ($\chi^2 = 8025600$, $df = 20$, p-value $< 2.2 \cdot 10^{-16}$).

TABLE IV
RELATIVE POPULARITY OF DEPENDENCY SPECIFICATION NOTATIONS.

Ecosystem \ Type	any(*)	caret(^)	exact	other	range	tilde(~)
JS Application	0.047	0.498	0.221	0.005	0.019	0.210
JS Package	0.037	0.536	0.217	0.007	0.029	0.174
Ruby Application	0.583	0.157	0.135	0.000	0.063	0.062
Ruby Package	0.360	0.178	0.070	0.000	0.249	0.143
Rust	0.320	0.034	0.478	0.109	0.007	0.052

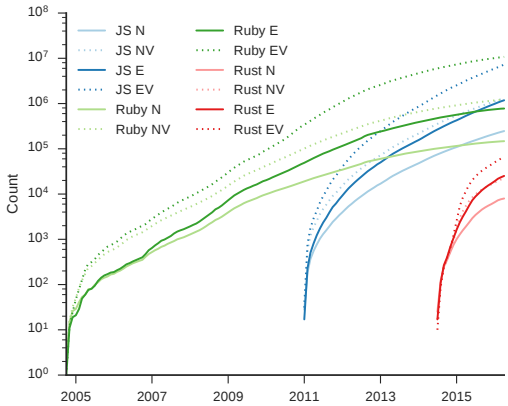


Fig. 3. Number of unique projects (N), dependencies between projects (E), the number of versions (NV) and dependencies between versions (EV).

B. Dependency network evolution (RQ2)

In this subsection, we will look in more detail the evolution of the dependency networks.

1) General growth: To understand how the ecosystems are growing, we first analyzed the number of projects and dependency relations between them. Figure 3 shows the number of projects and unique relations in the dependency network. We also show the number of releases and the number of dependency links between them. We see that in almost all cases, the speed at which the number of relations is growing is getting faster compared to the number of nodes in the network, especially visible for JavaScript (JS N on Figure 3), where the difference between the number of projects and dependencies is tenfold. The figure also indicates the growth of Rust is still continuing. JavaScript has become larger than Ruby, both in terms of versions and dependencies between versions. The

growth of Ruby in general is leveling off and becomes steady, whereas JavaScript is growing even at accelerated rate.

Figure 3 highlights the size differences when analyzing actual networks and aggregated networks with annotated edges. There is more than ten times difference between the number of nodes and edges in both networks and the difference is growing. Therefore, there are differences in the network structure which confirms our initial discussion on the choice of network modeling approach.

As the ecosystem is composed of multiple projects, we next analyzed the project level changes in dependencies. What is the number of dependencies and dependents for projects and the full size of the transitive dependency chain? Figure 4a shows the number of dependencies and dependents for each project release. We see a faster growth for the number of dependents in Ruby and JavaScript. The number of dependencies has been growing at a slower rate. When comparing JavaScript and Ruby, we see that the difference between the number of dependents is larger than the number of dependencies. One possible explanation could be that the overall number of packages published in `RubyGems` is smaller than in `npm` and there are fewer alternatives for packages, leading to higher number of dependents.

Figure 4b shows the total amount of dependencies for each project release. We observe fast growth for JavaScript projects and slower, steadier growth for Ruby and Rust projects. The average size of total dependencies for JavaScript was 34.3 in April 2015 but grew to 54.6 in April 2016, more than 60 % yearly growth. Growth at such speed is unlikely to continue and most likely will be lower in the future.

When comparing JavaScript's and Ruby's numbers of direct dependencies (Figure 4a) and the total amount of transitive dependencies, we see that JavaScript projects have more transitive dependencies, but less direct dependencies. This behavior indicates differences across these two ecosystems. Ruby has packages that are used mostly by applications and do not have dependencies themselves, but JavaScript published packages have dependencies themselves, making the ecosystem more connected and complex. One possible explanation for JavaScript's larger amount of transitive dependencies is the fact that `npm` allows multiple versions of the same project to be included through transitive dependencies.

Judging by these observations, it is hard to predict the exact number of transitive dependencies for Rust as both Ruby and JavaScript have shown different behavior. We argue that this may be because Rust is a very new ecosystem at its initial stages of evolution.

2) Conflict evolution: The ecosystems keep growing and the number of dependencies between projects is also growing. We analyze next what is the number projects that have a single dependency included through multiple packages, which could leave to conflicts if the package version requirement specification would not match.

We define a dependency overlap as a situation when a project appears as a dependency through multiple different paths for a single project. In practice, overlap could lead to conflict, which would occur only if the version specification would not match and it would not be possible to find the best matching version. Dependency overlap illustrates how much dependencies are co-used in projects. On the other hand, it illustrates the need for consistent usage of version number specification by package maintainers. Increasing dependency overlap should give developers a signal to look their dependency version requirements and use as loose criteria as possible, to allow dependency managers to find a suitable version.

Figure 4c lists the fraction of projects that have dependency overlap in their dependency chains. The overall trends are similar to the overall growth of the ecosystems. More than two-thirds of Ruby and half of JavaScript projects have a single dependency appear through multiple dependency chains. The result indicates package reuse, but also the event of dependency version conflicts might become more likely. Increasing overlap can lead to issues which prevent different packages to be used together due to not satisfiable dependencies. Similar behavior has been observed for Debian software packages [29].

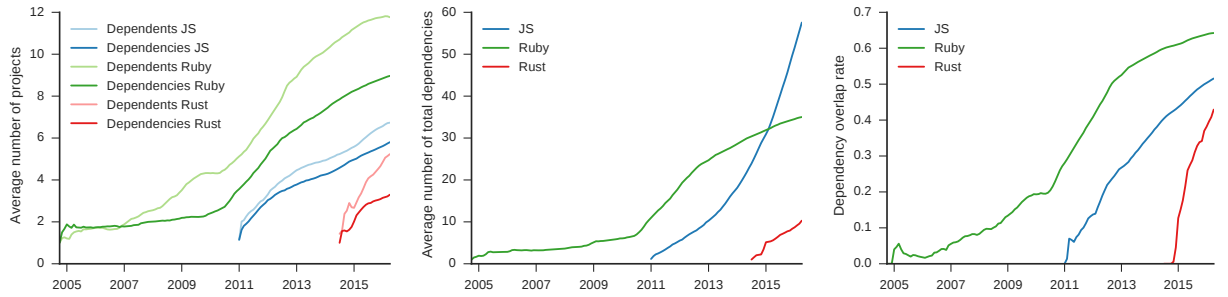
C. Fragility and vulnerability (RQ3)

Next we analyze dependency network tolerance to a removal of a single project or a single release. We define vulnerability of a package as the fraction of the network nodes that is impacted by a removal of a single package or a single package version. This approach enables to analyze the impact of incidents such as the *left-pad* project removal. While complete removal of a project removes all versions from the dependency networks, we can also study removal of a single version. For example, bugs or security vulnerabilities might not impact all project version, only selected specific of them might contain the bug.

We first calculate the vulnerability on the network where each node denotes the different version. For each package version, we calculate the number of total dependents. Next, we have the list with the number of total dependents for all packages. Among this list, we look at the maximum value and the 90th percentile value. We chose these values as the distribution of the number of dependents is skewed and the median value is typically either 0 or 1 depending on the snapshot date.

Figure 5a shows the maximum and 90th percentile vulnerability score normalized with respect to the full network size at each snapshot. We see that the maximum is fluctuating and having a positive trend, which means that there is a version in the network which importance is growing. Looking at the 90th percentile value, we see decreasing trend, which indicates that most of the other packages int he ecosystem are not central and are not included in the majority of dependency paths.

We also look at the vulnerability on the aggregated graph. Figure 5b shows the same vulnerability calculation on the aggregated network, meaning we remove a project and all its versions. It is evident that the maximum score is growing and impact a single project is growing. This is even interesting



(a) Evolution of the number direct dependencies and dependents for each project version. Average over those that have at least one dependency or one dependent.

(b) Average number of total dependencies, including the full transitive closure. Average calculated over projects that have at least one dependency.

(c) Dependency overlap evolution. Fraction of projects having at least one dependency required through multiple projects.

Fig. 4. Dependency network evolution

in the context of growing ecosystems, the absolute values are also increasing therefore. The 90th percentile vulnerability is again decreasing.

To find differences between packages and application, we analyzed the mean vulnerability rate for different types of JavaScript and Ruby projects. Figure 5c shows the average number of projects affected by a single package removal. The figure illustrates the dependence on a single package. We see that right after the creation of the package ecosystem, it starts to decrease. In a later phase, the positive trend of JavaScript is more visible. The average number of impacted applications remains larger than the packages.

Table V list the top five releases based on unique dependent projects and unique dependent releases. For JavaScript, the list is composed of unique utility packages, such isarray or inherits. For Ruby and Rust we see that multiple versions of a single packages have made to the top lists. The top five packages for ruby are related to webserver (rack) or templates (erubis, tilt). Rust packages are interface to system level types and libraries (libc), a serialization library (rustc-serialize), and a logging library(log).

VI. Discussion

In the following, we discuss our results, their practical implications, compare results with the related work and outline limitations of the research. The results differ to some extent for all studied languages, but generalizing conclusions can be brought out.

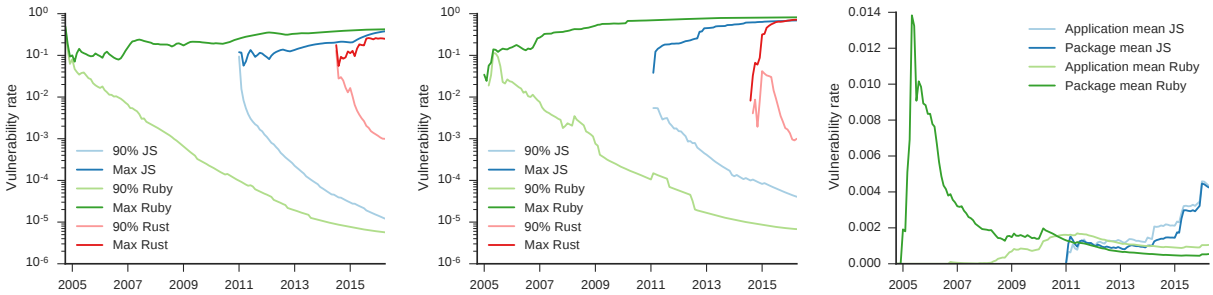
A. Results

Network modeling. Previous research on package dependency networks has not found an agreement on how to model dependencies using graphs. We propose an approach for modeling and constructing the network from dependency data. We believe that the chosen approach captures the actual network most accurately, enabling us to analyze dependencies on their version level. Although the analysis of aggregated network can yield similar conclusions [10], the real dependencies are still using version information and in future evolution stages it might not be sufficient anymore. We believe that our contribution in network modeling is a single step forward more unified software dependency network modeling.

Structure. Analysis of dependency network structure reveals differences between ecosystems. Although this has been observed before [9] for the dependencies, we have also shown differences in dependency version constraint specifications across ecosystems. The findings complement previous research [14] that found that different ecosystems approached API changes differently, which could impact dependency management. Our findings indicate that there are more implicit version updates than explicit, which suggests that there may be a need for tools to automatically monitor the dependencies that are included through implicit updates and reveal possible breaking API changes.

Evolution. Our evolution analysis revealed that the amount of transitive dependencies for JavaScript projects has been growing over 60% over the past year. A large amount of dependencies can lead to issues such as extended build time because of fetching the dependencies and increased software package size. Exponential growth has been observed inside Apache ecosystem as well [15]. Recently, a newer dependency management tool compatible with `npm` was introduced [30]. One of the key new functionality is improved concurrent dependency download. The tools tries to solve the dependency abundance problem by providing a faster download. Alternatively, a future solution could study how to reduce dependencies by better static code analysis. Our finding illustrates that observing network evolution, such troubles can be anticipated. Analysis of trends and number of transitive dependencies over time could be useful for other package based language ecosystems.

Vulnerability. Our vulnerability analysis, inspired by the *left-pad* incident [5], reveals that each studied ecosystem has packages whose removal could impact up to 30% of the other packages and applications. We showed that ecosystems have a few central packages that they depend on, which could enable bug spreading if they are not up to date. The high vulnerability score of a package should also alert developers and maintainers to make sure all security bugs are fixed quickly. A package with a high vulnerability score can be of interest to attackers



(a) Relative vulnerablity with respect to the actual dependency network size. The network size is the number of releases.

(b) Relative vulnerability with respect to the total number of unique projects in the network

(c) Relative vulnerability among published packages and GitHub projects

Fig. 5. Vulnerability of the ecosystems

TABLE V
TOP PROJECTS BASED ON TRANSITIVE DEPENDENTS(UNIQUE PROJECT RELEASES)

JavaScript				Ruby				Rust			
package	direct	transitive	vulnerability	package	direct	transitive	vulnerability	package	direct	transitive	vulnerability
inherits 2.0.1	8131	499254	0,38	erubis 2.7.0	9014	519555	0,42	libc 0.1.1	44	5520	0,25
isarray 0.0.1	727	384907	0,29	rack 1.4.1	4707	490329	0,40	rustc-serialize 0.3.16	1651	5379	0,24
core-util-is 1.0.1	524	371871	0,28	rack-test 0.6.2	1566	386937	0,31	libc 0.2.1	141	4840	0,22
string_decoder 0.10.25	39	303116	0,23	rack 1.3.10	3248	362810	0,29	libc 0.1.4	79	4598	0,21
sigmund 1.0.0	256	283319	0,21	tilt 1.3.3	2084	359862	0,29	log 0.3.1	1030	4415	0,20

as an opportunity to exploit projects depending on it.

B. Design implications

By using our findings, one could design a better package ecosystem and dependency management tooling. First, we would propose making dependency relations explicitly visible to understand the importance of packages in the ecosystems. Having an up to date view which packages are most popular and important in the ecosystem can make sure they receive maintenance and support effort from the community.

We would also investigate alternatives to semantic versioning to allow stricter dependency specification and version numbers from packages to help to minimize dependency conflicts. Overall, the ecosystem and tooling should improve awareness of what dependencies are used, make dependency listing explicit and help to minimize irrelevant dependencies.

C. Limitations

The limitation of our dependency network construction approach is that it will not compose the exact representation that the build tool would have. When resolving wildcard version specifications to a matching version, we look all dependencies separately for given projects. In practice, when using build tools, the whole transitive closure of dependencies would be resolved and if a package is included through multiple paths, a matching version would be calculated that shares all requirements. To recreate the exact dependencies for a project historically is complicated as dependency management tools do not support backdated retrieval.

VII. CONCLUSION AND FUTURE WORK

Our analysis of dependency networks of JavaScript, Ruby, and Rust shows that all analyzed ecosystems are alive and growing, with JavaScript having the fastest growth. JavaScript also shows the largest amount of transitive dependencies per project among studied languages. All ecosystems have some popular packages used in the majority of the projects. Yet, over time, ecosystems have become less dependent on a single popular package and a removal of a random project will not cause ecosystem collapse.

The main contributions of this paper are: (i) proposal of a network modeling approach specifically for dependency networks, (ii) insights into the structure and evolution of JavaScript, Ruby, and Rust, (iii) analysis of vulnerability reveals that ecosystems are not as vulnerable to a removal of a single package as they used to be.

This work opens up possibilities for multiple lines of future work. The dependency management process should be studied also studied by qualitatively to understand issues developers are facing. Second, based on the vulnerability measures and network aspects, a measure quantifying dependency health should be developed. Combining network information with data about testing efforts, code analysis, the number of maintainers etc, into a aggregated dependency health measure. The broad level goal of the future research is to support developers with tools in dependency management and maintenance and provide analytics for package maintainers about their packages and the overall ecosystem trends. Our next goal is to turn the code used in this paper into a set of reusable tools to analyze any package ecosystem based on GitHub and repository data.

SUPPLEMENTARY INFORMATION

Datasets used in this research are available at https://github.com/riivo/package-dependency-networks.

References

[1] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.

[2] J. Cox, E. Bouwers, M. v. Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 109–118.

[3] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" Master's thesis, TU Delft, Delft University of Technology, 2015.

[4] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," in *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 215–224.

[5] "left-pad issue #4," https://github.com/stevemao/left-pad/issues/4, last accessed 25.01.2017.

[6] P. Tripathy and K. Naik, *Software Evolution and Maintenance*. Wiley, 2014.

[7] T. Mens and S. Demeyer, *Software Evolution*, ser. SpringerLink: Springer e-Books. Springer Berlin Heidelberg, 2008.

[8] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Workshop on Mining Software Repositories*, ser. MSR '16. ACM, 2016, pp. 351–361.

[9] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *European Conference on Software Architecture Workshops*, 2016.

[10] ——, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 2–12. [Online]. Available: http://dx.doi.org/10.1109/SANER.2017.7884604

[11] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the r software ecosystem," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 243–252.

[12] A. Decan, T. Mens, M. Claes, and P. Grosjeanm, "When github meets cran: An analysis of inter-repository package dependency problems," in *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

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

[14] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, ser. FSE '16. ACM Press, 11 2016.

[15] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.

[16] ——, "The evolution of project inter-dependencies in a software ecosystem: The case of apache." in *ICSM*, 2013, pp. 280–289.

[17] R. G. Kula, D. M. Germán, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 520–524.

[18] M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A historical analysis of debian package incompatibilities," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. IEEE Press, 2015, pp. 212–223.

[19] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli, "Strong dependencies between software components," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 89–99.

[20] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon, "Maintaining large software distributions: new challenges from the foss era." in *Proceedings of the FRCSS 2006 workshop*. EASST, 2006, pp. 7–20.

[21] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in foss distributions: Details and challenges," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '08. ACM, 2008, pp. 7:1–7:5.

[22] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 516–519.

[23] A. Serebrenik and T. Mens, "Challenges in software ecosystems research," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ser. ECSAW '15. ACM, 2015, pp. 40:1–40:6.

[24] "How a student fooled 17,000 coders into running his sketchy programming code," https://fossbytes.com/typosquatting-technique-used-by-student-tricks-17000-coders/, accessed: 2016-06-19.

[25] "Npm api," https://registry.npmjs.org/-/all, accessed: 2016-05-01.

[26] "Rubygems api," https://rubygems.org/pages/data, accessed: 2016-05-01.

[27] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 233–236.

[28] D. Easley and J. Kleinberg, *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[29] P. Abate, R. Di Cosmo, L. Gesbert, F. Le Fessant, R. Treinen, and S. Zacchiroli, "Mining component repositories for installability issues," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 24–33.

[30] "Yarn: A new package manager for javascript," https://code.facebook.com/posts/1840075619545360/yarn-a-new-package-manager-for-javascript/, accessed 2016-10-27.