Vulnerability of Package Dependency Networks

Daniel Setó-Rey, José Ignacio Santos-Martín, Carlos López-Nozal

Abstract

Software reuse by importing packages from centralised repositories is an efficient and increasingly widespread way to develop software. Given the transitivity of dependencies, defects introduced in the repository can have extensive effects on the software ecosystem. Drawing from complex network theory, we define a model of repository vulnerability based on the statistically expected damage that the repository sustains from the random introduction of software defects. We test the model in stylized networks derived from real repositories, PyPI, Maven and npm, and show that the existence of a giant strongly connected component (SCC) explains most of the vulnerability. Indeed, we found that theoretical protection (immunization) of this entire component would remove almost all vulnerability from the network. Since repositories and their communities have limited resources to mitigate issues, we further model the problem of how to best apply these resources, finding sets much smaller than the giant SCC whose protection is nearly as good. Furthermore, we prove that the optimal selection of sets of given size is NP-Hard but can be approached with heuristics, yielding respectable results. Our model contributes to a better understanding of software package repositories and could also be applied to other systems with a similar structure.

Keywords: Complex network, Network structure, Network vulnerability, Package dependency networks, Software repositories

1 Introduction

The reuse of components is one of the cornerstones of software development. Software components can be maintained, fixed, and documented only once, which brings great benefits in quality, productivity, and efficiency [1]. Software component repositories, a collection of interdependent and reusable packages, are widely used by open-source projects. A repository works like a centralized market where a developer can seek and get a lot of functionalities necessary for their project. A survey conducted among developers in 2014 concluded that 90% of the lines of code in a typical application correspond to external components and that more than 80% of projects use centralized repositories of components [2].

The purpose of a package repository is the distribution of reusable software components. The term "package" refers to the appropriate artifact to facilitate this distribution, including the downloading, verification and installation of the components by the repository users. The development of software repositories is usually a bottom-up process. Programmers freely decide to create new functionalities by combining their own code with code already published and available inside any package in the repository to finally encapsulate it in a new package that is in turn added to the repository. In most repositories there are no requirements, beyond certain formats and technical protocols, as to who can publish and what can be published. In addition, for each language or system, there is usually a very popular repository that concentrates most of the community's development. For this reason, it is common to speak of *centralized* repositories. When a computer executes any part of a package, it processes its code lines and the rest of code lines of the dependent packages that may form long chains. In short, a code line of a package is virtually present in all packages that use it, all packages that use these second packages, and so on, until the end of the chains of dependencies. Therefore, any error in a code line may have effects on multiple packages depending on the dependency structure of the repository.

On March 20, 2016, Azer Koçulu, a Californian software developer, was involved in a dispute over the name of one of his JavaScript libraries in the npm repository, and after a questionable intervention of npm's managers, he decided to remove all his packages from the repository [3]. Among these packages, there was a script to justify text strings, called left-pad, widely used by other npm packages. After that, almost a million websites began to have difficulties updating their dependencies or deploying new versions, and these failures affected big firms such as Facebook and Netflix. Finally, npm's managers decided unilaterally to restore the left-pad package to solve the problem. Another remarkable example is Heartbleed, a big security vulnerability in the popular open-source cryptographic library OpenSSL used to provide secure connections on the Internet. A simple bug by one of its developers went unnoticed for two years jeopardizing all systems with this dependency in their software, and as result, hundreds of thousands of secure web servers were vulnerable to attacks including popular web sites such as Google, Youtube, Yahoo, Pinterest or Instagram among others [4]. These cases evince that the use of software repositories may involve risks due to the transitive dependency of packages.

Similar errors can be triggered by multiple issues, like bugs or security vulnerabilities. A recent report points out that security vulnerabilities detected in open source software almost double every two years, and nearly 80% of them are caused by indirect dependencies [5].

Although there is a growing interest in the community of developers and users [6], and there are some incipient works that examine particular issues and vulnerability cases (see details in the Sec. 2), in our opinion, the study of software package repositories as a whole that focuses on the networked structure has not received enough attention so far. The objective of our work is to understand the complexity of vulnerability of repositories and propose strategies to reduce their negative effects. To meet this goal we make use of network theory to model a software repository as a directed graph that we call *package dependency network*. Based on this network, we propose a formal definition of vulnerability considering all transitive connections, which measures how sensitive the network is to the introduction of random defects, assumed to be caused by diverse and unspecific phenomena. We call this kind of vulnerability *failure vulnerability*, in contrast to *attack vulnerability*, where failures are not random, but directed by a malicious actor. In the rest of this paper, unless otherwise stated, we will refer to the former simply as vulnerability.

The main contributions of our work are summarized in the next points:

- The definition of a novel model to quantify the vulnerability of package repositories based on the structure of the underlying dependency network.
- The study of possible methods to reduce the vulnerability of repositories.
- The analysis of a set of stylized models derived from three well-known real repositories. Our results show that the network's vulnerability is related to the size of the largest strongly connected component (SCC), caused by the appearance of cyclic dependencies. When the SCC has a significant size the vulnerability of the network is much higher. We found out that it is possible to greatly reduce the vulnerability of a repository by protecting a small fraction of packages, although the search for the optimal set of packages that maximize the protection is an NP-Hard problem. We study several heuristics and get significant reductions by acting on a suboptimal number of packages.

The rest of the paper is organised as follows. Section 2 summarizes a review of related work, paying special attention to the study of vulnerability and cascading failure phenomena in complex networks. Section 3 describes the formal model and the main concepts that allow us to quantify the vulnerability of a repository to random failures. Section 4 contains the test of the model on three major programming language repositories and the results obtained and Section 5 covers the discussion of its implications and possible applications. Finally, the last section summarises the main conclusions of this work.

To make the research method used in this work available to the scientific community, the code is published in the repository "Open-source Library Indexes Vulnerability Identification and Analysis" (OLIVIA) [7].

2 Literature review

In software engineering literature, the term vulnerability usually refers to a particular code defect that can be used by a malicious attacker to get a benefit. However, we use this term to refer to the potential damage to a repository due to any type of error or malfunction in its components. Note that in the model we propose in this article, a defect is not only a security flaw and that we pay attention to the function of the entire repository, not a particular package. In short, our interest is to study the vulnerability of a package repository as a whole and how simple errors trigger a cascade of failures affecting long chains of packages in the network of dependencies. In this section, we review first the literature related to package repositories, focusing on their networked structure, then the literature about the study of vulnerabilities in complex networks. It is not intended to be exhaustive, but to mention the main research lines in these topics.

The use of package repositories has grown in the last years in both proprietary and open-source software. With the expansion of collaborative projects, the importance and size of open-source repositories are becoming much more prominent, e.g., popular programming languages such as Java with its Maven repository, JavaScript with npm, Python with PyPI or R with CRAN, among others [8].

The use, management, and maintenance of these ecosystems have challenges and issues, some of them are related to the networked structure of dependencies among packages. For example, the colloquial term “dependency hell” [9] is often used to define issues caused by the long chains of dependencies an installer has to fetch, something particularly critical in large software compilations [10]. Some package-management systems do regular surveillance, e.g., CRAN, the primary source of R packages, checks the running of packages to find errors and inconsistencies in the versions. But this type of predictive maintenance is not common, and developers usually go blind choosing packages within a repository.

We also find security risks associated with the use of repositories that can depend on the dependency network. So, some authors identify malicious packages [11] and evaluate the propagation of a known set of security vulnerabilities [12, 13]. An interesting research line uses data mining techniques to exploit the heterogeneous information contained in repositories that improve the knowledge about software development [14, 15, 16]. These works can complement the intent to understand developer decisions and practices that affect the evolution of software ecosystems [17]. However, to explicitly formalize the network of dependencies as we do in this article is a recent and under-explored approach, although network theory seems to be suitable for studying the complexity of these systems [18]. Among these works, it is worth mentioning some some studies of software systems [19, 20] and comparative analyses of popular programming languages [21, 22, 23]. These works show that by formalizing the network of dependencies, it is possible to unveil relevant features such as changes in size and complexity over time, the little fragmentation due to the existence of a large component of packages, or the heterogeneous distribution of dependencies that shows how a few packages accumulate a large proportion of the dependencies.

The relationship between security risks and the topology of the dependency network in npm is analyzed in [24]. The authors find that some individual packages could impact a large part of the repository and propose some actions to reduce this risk by mainly vetting untrusted maintainers and code releases. We will see that these results are in line with the results of our work.

Aware of these risks, some European research groups have launched the HORIZON 2020 project “Fine-Grained Analysis of Software Ecosystems as Networks” (FASTEN), although they propose a higher level of granularity in the analysis than in our work, trying to map the function call dependencies in a repository [3]. Other related initiatives have recently emerged within the open source community, such as the Open Source Security Foundation’s “Securing Critical Packages” technical working group [25].

The term vulnerability is widely adopted in the network and complex systems literature. Within this literature, we first explore vulnerability by looking at the opposite term robustness. In network theory, robustness refers to the capability of a network to continue operating after a damage in a set of nodes [26]. Robustness has long been a topic of central interest with multiple applications on technological, economic, social, or biological systems whose dynamics depend on the topology of the interactions among their components. Take, for example, the electric power system, the transportation system, the Internet, or the financial system, among others, where a failure in a part can severely affect the system’s operativeness as a whole depending on the structure of the network. A common approach to study robustness is to model failures as removals of nodes and study how these removals, random or deliberate, affect the overall connectivity of the network [27, 28]. We find prolific research that uses mathematical and computational models to analyze phase transitions that separate the states of tolerable connectivity from unacceptable disconnection [29, 30, chapter 6]; Gross and Barth [31] provide a new review of the analytic framework used to study network robustness. Unfortunately, failures in package repositories do not fit well into this approach, since connectivity cannot be related to a measure of robustness in package networks.

Another approach to analyzing the vulnerability in networks is to model explicitly the spread dynamics [32]. In this case, we are interested in how the structure of a network affects the spread of a failure. Once again, we find a rich range of mainly socio-economic and biological phenomena that display a spreading process over a network, and that has generated a rich theoretical and empirical research (in a socio-economic context we usually talk about diffusion processes, while in a biological context we talk about epidemic processes) [33, Chap. 9], [34, Chap.15,16], [35, Chap. 10], [36]. We can group most of the theoretical models into three different classes that differ in the object and the mechanism of spreading. In all of them, an individual interacts with its neighbours (defined by a network), and as a result of this interaction, the individual changes its state. First, we find contagion-dependent models, i.e., an individual exposed to an infected neighbour can be infected with a probability. This mechanism similar to biological infections is the basis of many epidemic models [37, 38]. Second, frequency-dependent models (or threshold models), i.e., an individual changes its behaviour or belief if a fraction of its neighbours that follow another behaviour or belief is greater than a threshold. These models have been proposed to study technology (or innovation) adoption and opinion formation [39, 40]. And third, payoff-dependent models, i.e., an individual chooses a strategy and plays with its neighbours, then it gets a payoff depending on all players' strategies, and changes to another neighbouring strategy if it gives more payoff. These strategic models have long been studied by game theory [41, 42]. There is an excellent review of these models in [43, Chapters 3-4]. It is possible to sophisticate these models considering links of multiple classes (i.e., hypergraph) and formalize more complex interaction to study hypergraph robustness in higher-order networks. There is an emerging research line on contagion dynamics in high-order models (see the review by [32]).

However, the spread of failures in package repositories differs from these models: (i) there is no strategic interaction among packages like in payoff-dependent models; (ii) a failure in a package does not depend on the frequency of neighbouring failures like in frequency-dependent models; and (iii) unlike contagion-dependent models, we need to consider the number of defects that each package accumulates. In package repositories, unlike biological systems, it is only necessary to repair the package sourcing of error to avoid the propagation to all transitively dependent packages. Moreover, any of the packages in-between may solve the problem for its transitive dependants if there are no alternative paths from the origin of the error. We also note that, in contrast to conventional epidemic or social-economic models, propagation phenomena in a package repository are inherently asymmetric.

3 Method

3.1 Description of the network model

We propose a model where the network corresponding to a package repository is constructed using a simple directed graph $G = (V, E)$. We call G a package dependency network. The nodes (V) represent the packages in the repository and the arcs (E) represent the dependency between packages. The arc $(u, v) \in E$ expresses the relation "node v depends on node u". We will write $u \to v$ to express the existence of a dependency between the two nodes. Only packages that have at least one dependent or at least one dependency are considered. This kind of dependency is a reflexive and transitive binary relation, i.e. a preorder, over the domain of packages in the repository. That is, we consider that each package depends on itself, $v \to v$ for all v, and from $u \to v$ and $v \to w$ it follows that $u \to w$. In the latter case, we say that u is a transitive dependency of w, and that w is a transitive dependent of v. A defect or failure in a package may potentially affect itself and all of its direct and transitive dependants.

3.2 Modeling vulnerability

We consider that a package repository must fulfill the function of uninterruptedly providing a set of reusable, quality software components that operate according to its specifications. We shall say that the service level of the repository decreases when its operation deviates from this role. When a defect is introduced into a package of the repository, we will say that this node *fails* in the network model. This failure results in damage, a decrease in the service level of the repository.

We will not assume any implicit restrictions on the nature of failures arising in the repository. That is, we are not focusing on a particular kind of defect, nor on a particular set of possible causes. These causes are assumed to be diverse –human errors, hardware failures, communication or process errors, etc.– and the superposition of their effects is seen as a stochastic process: we will model the vulnerability on the assumption that the repository is subject to the continuous introduction of defects of a random and independent nature.

Our concept of vulnerability shall seek to reflect the statistically expected damage that the repository endures when facing these random defects. We start by analyzing the failure of a single package. We establish a mapping between each package and a measure of this damage –*what is the damage to the network caused by the introduction of a defect in some package?*–, i.e. a cost function over the domain of the network nodes. Thus, the expected damage caused to the network by a random failing package would be the average value of this function. However, considering a network that presents a single failure is unrealistic. Therefore, if we want a meaningful model, we need to generalize our cost function to the case of subsets –*what is the damage to the network caused by the (possibly concurrent) failure of some subset of packages?*–. But this abstraction is not easy to set up for any cost function. In general, the relationship between the value of the function for a set of packages and its subsets, including single packages, depends on the structure of the network and would be quite challenging to define explicitly for any combination of packages. Yet, this difficulty can be avoided by using an additive cost function, such that the sum of costs of the failure of two disjoint sets of packages is equal to the cost of the failure of the union set.

Definition 1 (Failure vulnerability). *Let $G = (V, E)$ be a package dependency network and consider the set function $Reach : \mathcal{P}(V) \to \mathbb{R}$ such that:*

$$Reach(W) = \left| \bigcup_{v \in W} (\{v\} \times out(v)) \right| \quad \textit{for any } W \subseteq V$$

where $out(v)$ denotes the out-component of $v \in V$, i.e. the set of nodes that can be reached by recursively following all outgoing edges from v, including v.

We define the failure vulnerability of G, measured using Reach as a cost function, and denote it by $\phi^{Reach}(G)$, as the average value of Reach over individual network nodes:

$$\phi^{Reach}(G) = \frac{1}{|V|} \sum_{v \in V} Reach(\{v\})$$

It holds that $Reach(\{v\}) = |\{v\} \times out(v)| = |out(v)|$ and therefore we have that

$$\phi^{Reach}(G) = \frac{1}{|V|} \sum_{v \in V} |out(v)|$$

Note that *Reach* is possibly the simplest function that evaluates the cost of failures on a structural level. For a single package, it maps to the number of packages potentially affected through direct or transitive dependencies. However, it should be noted that the definition of *Reach* does not explicitly account for the number of packages affected, but rather for the number of defects introduced into the network. Considering other failing packages, this effect is accumulative, since defects transitively inherited from one dependency are added to defects inherited from another, increasing the damage to the network.

It is immediate to check that *Reach* is additive, i.e. $Reach(X) + Reach(Y) = Reach(X \cup Y)$ for all disjoint X, Y. This property allows to compute the expected damage to the network of k random packages incorporating a defect as $k\phi^{Reach}(G)$ –the potential introduction of this number of defects in the network, with some packages possibly affected by more than one defect–.

Reach is useful for measuring the damage to the repository by defect propagation. However, other cost functions with different assumptions can also be considered. If we associate a specific cost $w(v)$ to each package v in the network, we could easily calculate the weighted version of *Reach* as the sum of individual costs of the elements in each out-component, added across the set of failing packages. The cost may include information external to the network, for example, $w(v)$ could be the number of open source projects (outside the repository) that depend on v.

The exact value of ϕ^{Reach} for any network can be obtained from the transitive closure of the underlying graph, which may be reasonably computed for real package dependency networks using Purdom's algorithm [44] or one of its variants, such as the marginally more efficient Goralcikova-Koubek version [45]. These require the prior computation of the condensed network, which is usually done using Tarjan's algorithm [46]. The entire process can be completed with quadratic worst-case time complexity in the number of nodes in the network.

3.3 Modelling immunization

We say that a node is immunized when any action is taken to eliminate the possibility of it failing or incorporating a defect. It is clear that immunizing certain nodes reduces the vulnerability of the network. Note that we use the term immunization as an abstraction of any preventive action, e.g., special monitoring of a package by the repository manager, that reduces the probability of failure in a package. Immunization at its maximum effectiveness would make this probability zero, although this is not possible in real systems. The analysis of the vulnerability of a dependency network would be greatly complicated if we were to consider a stochastic model. However, we will show in the results that the assumption of zero defects in an immunized package allows getting important insights into the complexity of these phenomena. Under this assumption, the effects of immunization should be interpreted as the maximum level of what could be achieved by preventive activities.

Considering a theoretical immunization that prevents the appearance of defects in a node from any cause, we must also eliminate the possibility of the defect appearing due to transitive propagation from one of its dependencies. Therefore, in our model, an immunized node also prevents the propagation of defects to its dependents. It should be noted that each node plays an intermediary role, forwarding the defects of all its transitive dependencies. Node immunization prevents this transmission, but the defects could still reach all or part of its transitive dependents through alternative paths.

Since an immunized node can neither present nor propagate defects, the vulnerability of the immunized network can be analyzed simply by defining an ancillary network where we eliminate the node together with its incoming and outgoing edges see Figure 1).

Definition 2 (Immunization delta)**.** *Let $G = (V, E)$ be a package dependency network, $W \subset V$ a set of nodes in the network and $G \dot{-} W$ the network resulting from removing all nodes of W from G. We define the immunization delta $\Delta_G^\mu(W)$ as the reduction in the vulnerability of G, measured by a cost function $\mu : \mathcal{P}(V) \to \mathbb{R}$, due to the immunization of W:*

$$\Delta_G^\mu(W) = \phi^\mu(G) - \frac{|V| - |W|}{|V|}\, \phi^{\mu|_{\mathcal{P}(V-W)}}(G \dot{-} W)$$

Here the correcting factor $\frac{|V|-|W|}{|V|}$ accounts for $G \dot{-} W$ having $|W|$ less nodes than G.

In our model, we use *Reach* as the cost function, so the immunization delta Δ_G^{Reach} is the decrease in the average of packages potentially affected by a random failure due to the immunization of a given set of packages in G. Figure 1 shows some examples of immunization in simple dependency networks with the numerical values of the immunization delta.

Definition 2 gives a direct method for calculating the immunization delta for arbitrary cost functions and immunization sets on any package dependency network, computing vulnerability twice, first in G and then in $G \dot{-} W$.

From a practical point of view, it is interesting to determine which set of nodes to choose to achieve the greatest possible immunization delta, so that vulnerability can be efficiently reduced. For example, a package manager might

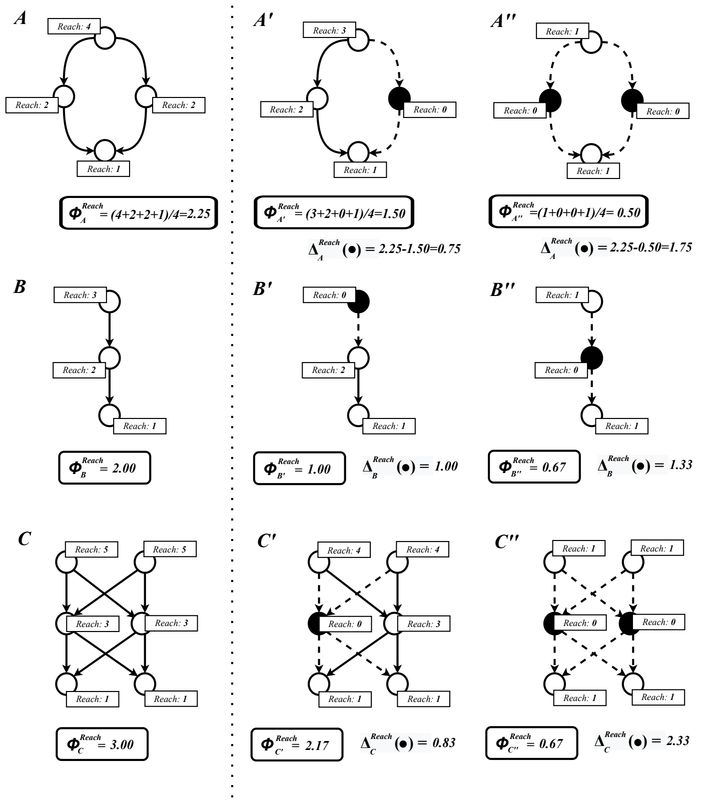
A
Reach: 4
Reach: 2
Reach: 2
Reach: 1
Φ_{A}^{Reach} = (4+2+2+1)/4=2.25
A′
Reach: 3
Reach: 2
Reach: 0
Reach: 1
$\Phi_{A'}^{Reach}$ = (3+2+0+1)/4=1.50
$\Delta_{A}^{Reach}(\bullet)$ = 2.25-1.50=0.75
A″
Reach: 1
Reach: 0
Reach: 0
Reach: 1
$\Phi_{A''}^{Reach}$ =(1+0+0+1)/4= 0.50
$\Delta_{A}^{Reach}(\bullet)$ = 2.25-0.50=1.75
B
Reach: 3
Reach: 2
Reach: 1
Φ_{B}^{Reach} = 2.00
B′
Reach: 0
Reach: 2
Reach: 1
$\Phi_{B'}^{Reach}$ = 1.00
$\Delta_{B}^{Reach}(\bullet)$ = 1.00
B″
Reach: 1
Reach: 0
Reach: 1
$\Phi_{B''}^{Reach}$ = 0.67
$\Delta_{B}^{Reach}(\bullet)$ = 1.33
C
Reach: 5
Reach: 5
Reach: 3
Reach: 3
Reach: 1
Reach: 1
Φ_{C}^{Reach} = 3.00
C′
Reach: 4
Reach: 4
Reach: 0
Reach: 3
Reach: 1
Reach: 1
$\Phi_{C'}^{Reach}$ = 2.17
$\Delta_{C}^{Reach}(\bullet)$ = 0.83
C″
Reach: 1
Reach: 1
Reach: 0
Reach: 0
Reach: 1
Reach: 1
$\Phi_{C''}^{Reach}$ = 0.67
$\Delta_{C}^{Reach}(\bullet)$ = 2.33

Figure 1: Examples of immunization in simple package dependency networks. The figure shows three initial networks, i.e., A, B and C, and two immunization cases for each one, i.e., A', A'' and so on, in which one or two nodes are immunized. We show *Reach* values for each node, the network vulnerability before and after the immunization of different sets of nodes (highlighted in black) and the immunization delta. The symbol $\bullet$ denotes the set of immunized nodes in each case. The calculations of these magnitudes are detailed in the first row of cases. The *Reach* of a node (see Definition 1) is the size of the out-component (i.e., the set of nodes that can be reached by recursively following all outgoing edges from a node, including itself). The *Reach* value for an immunized node is 0 because neither itself nor its outgoing arcs (drawn by dotted lines) are considered in the calculation. For the same reason, *Reach* values for dependencies of an immunized node are decreased. The network vulnerability is the average size of the out-component of the nodes. The immunization delta (see Definition 2) is computed as the difference between the network vulnerability of the initial network and the immunized network.

be interested in providing additional assets, performing special monitoring, or imposing certain restrictions on key package developers, so that the network as a whole is less vulnerable to random failures. Depending on the type of protection strategy to be implemented and the resources available, it would be desirable to obtain an optimal set of k elements to immunize for an arbitrary k parameter. Clearly the difficulty of this task depends on the chosen cost function, but in general it seems a hard combinatorial problem, similar to other optimization problems on graphs. In the following we formalize it for the *Reach* case.

Definition 3 (*MAX-k-DELTA-REACH* problem)**.** *Let $G = (V, E)$ be a package dependency network and $k < |V|$ a natural number. We define the problem of maximum immunization measured by the Reach cost function (MAX-k-DELTA-REACH) as the problem of finding $V' \subset V$ such that $|V'| = k$ and $\Delta_G^{Reach}(V')$ is maximized.*

Indeed, *MAX-k-DELTA-REACH* can be proved NP-hard (Appendix A, Theorem 1 in the supplemental file). Dealing with an optimization problem on set functions it is interesting to check if Δ^{Reach} is submodular, in which case we would have a specific and powerful set of tools available for maximization [47, 48]. Unfortunately, it is not, as can be easily proved by counterexample (Appendix A, Proposition 1, in the supplemental file).

The OLIVIA repository [7] includes a Python library of functions with the definitions of this section together with a collection of Jupyter Notebooks as a user guide for researchers.

4 Results

We study three real reference package repositories using the previously defined theoretical framework, focusing on the *Reach* cost function. In this section, we outline the results and our analysis in relation to them.

From now on, when we use the term *vulnerability* of a network, it should be understood that we are referring to ϕ^{Reach}, the vulnerability to failure measured using the *Reach* cost function.

4.1 Sample networks

We have tested our model in three package dependency networks derived from *PyPI*, *npm* and *Maven*, the centralized package repositories of Python, node.js (JavaScript) and Java, respectively. These package repositories are associated with very popular programming languages (see for example TIOBE index [49]). On the other hand, we want heterogeneous repositories in size, which a priory could have different dependency structures, to get more general results.

The dependency information was constructed from the January 12, 2020, *libraries.io* data dump [8]. *Libraries.io* is an online service that aggregates information related to open source package repositories. It has been used with good results in other works [21, 50, 51, 52].

To extract the necessary information for our model we have used only the dependencies file included in the data distribution (*dependencies-1.6.0-2020-01-12.csv*). For this purpose, a simple script was utilized to read the source data, filter the dependencies corresponding to a specific repository and build a directed network model. Our theoretical approach is highly unconstrained in relation to the causes and types of defects. Thus, we consider dependencies in their broadest sense, in line with the UML standard definition of usage dependency [53], in which one package requires another package (or set of packages) for its full implementation or operation. So all kind of dependencies present in the file –runtime, test, development and build– are extracted and incorporated into the model.

We have only considered the dependencies at the package level and not the version level. Most package managers allow the definition of dependencies with version constraints. Solving a set of this kind of dependencies is a problem that does not generally have a unique solution. Moreover it is intractable [54], so package managers use heuristic techniques to approximate a valid set of dependency versions. Hence, the exact version of code that will end up being incorporated into a project depends on factors that are largely impossible to control at the network definition stage and so the dependency problem with versions generates arbitrariness in the determination of the edges between packages. Version-level models can be proposed on different sets of assumptions (which will inevitably introduce additional biases) to answer specific research questions. Given that we aim to draw interpretable conclusions about how structural features contribute to defect propagation, we do not consider that our work will benefit from this level of detail. Also, our modelling at the package level obviates the analysis of the specific use of subroutines or library methods. We believe that these assumptions, however, allow us to get a stylized model with enough complexity to understand the potential damage that defects in some packages can generate in others, compromising the functionality of the repository. All the proposed results are in any case applicable to network models of higher granularity, for example, in the FASTEN project mentioned in the literature review.

Therefore, the network corresponding to a package repository consists of the packages, i.e., nodes, in the repository and the set of arcs that represent the dependencies among packages. There is an arc from package A to package B if at least one version of B depends on some version of A. The implementation of the construction of the dependency network can be found in the OLIVIA repository [7].

4.2 Vulnerability characteristics of repositories

The sizes of the three models are markedly different, both in the number of nodes (packages) and arcs (dependency relationships) (Table 1). In addition, the dependency structure contains, particularly in Maven and npm, a significant number of cycles. We note that the occurrence of cycles in the dependency network is possible due to the presence of dynamic processes that not only create but also modify the packages in the repository. If packages were published once, at a single point in time, developers would only be able to include as a dependency the already existing packages, and cycles would be impossible. This is the case, for example, in citation networks. But packages can be updated at any time after publication, eventually adding new dependency relationships. The aggregation of cycles in a network results in the emergence of strongly connected components (henceforth SCC), i.e., sets of nodes such that there is a directed path between any pair of them. In package dependency networks, this means that within an SCC all packages are transitively dependent on all others.

If we measure ϕ^{Reach} in our models, we observe large differences that seem to be related to the presence of a giant SCC, of significant size relative to the magnitude of the network and much bigger than the second largest (Table 1). For example, introduction of k defects in PyPI can potentially replicate across the network structure $15.73k$ times, but this expected value is $1805.54k$ in Maven and $27193.83k$ in npm. So, even when npm is around 21 times bigger in size than PyPI, the effect of the introduction of k defects in npm is approximately 1723 times higher than in PyPI. This shows the multiplicative effect of the size of the SCC and therefore the importance of taking into account the complexity of the dependency structure when studying the vulnerability of any repository.

The emergence of a giant SCC is a well-known phenomenon in the field of network theory. In the case of directed networks it can be modelled by such a simple mechanism as random directed networks [55], a variation of the classical Erdös–Rényi model [56]. In the evolution of random networks, a critical point –dependent on the average number of edges per node– separates the subcritical regime, where the largest SCC has on the order of $log(n)$ elements for a network of n nodes, from the supercritical regime, where there is a distinctly larger SCC. Although it is clear that the structure of package dependency networks cannot be explained solely by the random network generation mechanism, we will use the terms subcritical and supercritical to refer to the two regimes in which we can classify a particular repository. Using the criterion of the order of the size of the SCC, PyPI would be in a subcritical regimen, without a giant SCC, and its vulnerability is therefore low. Maven and npm, on the other hand, would be in a supercritical regime, with a large SCC, and its vulnerability is therefore much higher.

Some authors use the concepts of weakly connected component (WCC), i.e., subgraph where all nodes are connected by some path, ignoring the direction of edges, and SCC to characterize the structure of directed networks. There is evidence [57, 58] that many directed networks show a structure where most nodes belong to a WCC that can be partitioned into three main subsets: the largest SCC, the set of nodes that can reach the SCC, i.e. the IN set, and the set of nodes that can be reached from the SCC, i.e. the OUT set, which is usually represented in the form of a bow-tie diagram [57]; the partition of the nodes is completed with other less relevant subsets (see [59] for a complete definition). The results show that Maven and npm present a bow-tie structure (Table 1) with a very large OUT module. This asymmetric bow-tie structure (a significant SCC with a huge OUT) helps to understand the fact that the vulnerability appears to be of the order of the size of the largest SCC (Table 1). In the bow-tie model, let us consider a network of size n and (i) an SCC of significant size but small compared to the network, (ii) an OUT set of size comparable to that of the network and (iii) negligible vulnerability caused by nodes outside the SCC (i.e., very sparse connectivity within the remaining components). Then the vulnerability of the network is dominated by the contribution to the propagation of defects from nodes inside the SCC, each including nodes in OUT, of size similar to n. Therefore, the average defect propagation can be approximated by the size of the SCC. Note how this phenomenon we observed in our case studies requires the relative balance specified in (i) and (ii). If the SCC were very small, its contribution to vulnerability is diluted by other sources of vulnerability included in the rest of the model components. On the contrary, if the SCC is

Table 1: Bow-tie components and vulnerability to failure of reference package dependency networks. n: number of packages, m: number of arcs (dependency relations), *2nd* and *1st-SCC*: size of second largest and largest strongly connected component present. IN, OUT, Tubes, Tendrils and Disconnected are bow-tie network structural components (see [59] for a definition of these terms). ϕ^{Reach}: vulnerability to failure measured by the *Reach* cost function and next to it vulnerability ratio in relation to the size of the network.

	n	m	2nd-SCC	1st-SCC	IN	OUT	Tubes	Tendrils	Disconnected	ϕ^{Reach}	ϕ^{Reach}/n
PyPI	50 766	155 369	4	14						15.73	0.0003
Maven	126 752	644 207	49	981	2 163	70 239	3 749	34 373	15 247	1 805.54	0.0142
npm	1 074 508	13 052 831	175	26 486	3 849	936 295	3 745	87 495	16 638	27 193.83	0.0253

of a size comparable to the network, the aggregate effect of defect propagation grows quadratically, and therefore its average could be approximated by a linear function over n. We remark that (iii) is something we observe in our models of real networks, where immunization of the SCC almost completely eliminates network vulnerability (Table 2).

The definition of vulnerability with an additive cost function (*Reach*) allows us to get interesting statistical results about the risk a project faces by using a repository. Assuming a known rate of defects r per period T that occur in a repository of size n, the expected number of defects for a project with m direct dependencies will be $mr\phi^{Reach}/n$, where ϕ^{Reach} is the vulnerability of the repository. The application of the above formula yields a value of approximately $0.025mr$ for our model of npm (see ϕ^{Reach} in Tab. 1), so for $mr \geq 40$ a client project may expect to incorporate a defect from the repository. For example, if a project imports $m = 4$ packages from npm, it will be sufficient to consider a rate r of approximately 10 defects per T for the project to expect the incorporation of a defect over T. For the same $m = 4$, r would need to be slightly higher ($r \approx 18$) in our model of Maven but more than 800 in PyPI. Note that r may refer to any type of specific defect of interest, as security vulnerabilities or bugs.

It is also straightforward to precisely estimate the odds of a project using the repository incorporating at least one defect after f failures in the repository. The probability that any given package is not affected by a random failure in the network is $1 - \phi^{Reach}/n$, and hence the probability that it incorporates no defects after f failures in the repository is $(1 - \phi^{Reach}/n)^f$. If a project imports m packages from the repository, the probability that it incorporates at least one defect after f failures is therefore $1 - (1 - \phi^{Reach}/n)^{fm}$. If we take for example $m = 4$ in addition to ϕ^{Reach} and n in Table 1, we have that this probability exceeds 0.99 after 46 failures (out of more than one million packages) and 82 failures (out of more than one hundred thousand packages) in npm and maven respectively, but the same level of risk only occurs in the smaller PyPI after more than 3800 failures. We conclude that developers have no hope of remaining free of defects when using supercritical repositories, even considering high-quality ones.

These simple results underline the important effect that the vulnerability of repositories may have on the ecosystem and the striking gap between low (subcritical) and very high (supercritical) vulnerability repositories.

Table 2: Decrease in vulnerability in supercritical package dependency networks by immunizing the largest strongly connected component (SCC) and the set of strong articulation points of it (SAP).

	ϕ^{Reach}	$\|SCC\|$	$\Delta^{Reach}(SCC)$		$\|SAP\|$	$\Delta^{Reach}(SAP)$	
Maven	1 805.54	981	1 738.99	96.31%	351	1 695.32	93.90%
npm	27 193.83	26 486	27 168.86	99.91%	5 902	24 702.74	90.84%

4.3 Optimization of immunization sets

In our model of PyPI, which as we have seen is in a subcritical regime, it is possible to significantly reduce ϕ^{Reach} by immunizing a small set of nodes. Let $P = (V, E)$ be the package dependency network corresponding to PyPI. We select a set $W \subset V$ to immunize. The selection can be done heuristically according to simple criteria. Choosing the 10 packages with the highest out-degree, i.e., the number of direct dependents, we have that $\Delta_P^{Reach}(W) = 3.73$. This represents a reduction in vulnerability of 24% by immunizing 0.02% of the network. It is easy to improve this result. For example, let us define $Upper : V \to \mathbb{N}$ such that

$$Upper(v) = |out(v)| \cdot |in(v)|$$

where $out(v)$ and $in(v)$ denote respectively the out-component –i.e., the set of nodes that can be reached by recursively following all outgoing edges from a node– and in-component –the set of nodes that reach a node by recursively following all their outgoing edges– of v. $Upper(v)$ is an upper bound on $\Delta_P^{Reach}(\{v\})$, so it is a rough heuristic about the potential of immunization for individual packages. If we take W as the top 10 packages according to $Upper$ values, we achieve a $\Delta_P^{Reach}(W) = 5.45$, a 35% reduction in the vulnerability.

We consider immunization of subcritical networks to be an interesting problem, but their vulnerability is negligible when compared to supercritical networks, and hence in this paper we will focus on the latter.

In the case of supercritical networks, we have seen that immunization of the entire SCC results in a drastic reduction in vulnerability (Table 2). However, not all packages within the largest SCC are equally important. It is foreseeable that the immunization of many sets does not alter the connectivity of the component as a whole, i.e., the rest of the packages are still connected by alternative routes. Since the strong connectivity of a giant component is the cause of the network vulnerability, to reduce it we must eliminate this property by choosing immunization sets that considerably decrease the size of the largest SCC. This task belongs to the family of problems of locating *feedback* sets in a network, which are generally intractable [60]. However, we may still employ approximate alternatives. We can, for example, look at the set of strong articulation points (SAP) of the largest SCC. Strong articulation points –whose detection can be performed in linear time relative to the number of nodes and arcs of the component [61]– are those whose single removal disintegrates the SCC, decreasing its size and creating at least one new SCC. In both Maven and npm, the SAP set of the larger SCC is much smaller than the SCC itself, but still its immunization achieves a reduction of the same order in the vulnerability (Table 2).

SAP is much smaller than the network, but is of considerable size and determined by the structure of the largest SCC itself. If we wanted to reduce the effort that a repository manager would have to make to take care of these packages and reduce their failure probability, we would need to get a smaller target set, i.e., solve an instance of the NP-Hard *MAX-k-DELTA-REACH* problem (Definition 3).

In this paper, we make an initial approach to the problem of finding approximate solutions. We test some well-known network centrality measures: out-degree –i.e., number of direct dependants– and betweenness –i.e., the importance of a package as an intermediary among the rest of the packages through paths of transitive dependencies, the greater the number of dependency paths a package is a part of the greater its betweenness [62]–, to test their performance in relation to the problem of choosing a given number of elements to immunize.

We note that the out-degree is expected to be correlated with the size of the out-component and with the probability of belonging to the giant SCC and, within it, to the SAP set. Moreover, the out-degree is an upper bound on the number of new SCCs that can be created by removing a node from the SAP, or the ability to disaggregate the SCC.

High betweenness centrality nodes are located in a large number of shortest paths between pairs of nodes in the network and correspond to packages that are in the middle of numerous chains of transitive dependency in the repository. To understand the heuristic value of this measure for our problem, consider that a high betweenness value for a package p guarantees a high number of transitive dependent and dependency pairs, corresponding to paths that pass through p, and potentially protected by the immunization of it. In addition, the existence of many different paths in the network between its dependents and dependencies would decrease the probability that those passing through p are the shortest. Therefore, for high betweenness nodes we can deem the probability of alternative paths for defect propagation to be smaller.

Our approach is to select the top-k elements with respect to centrality values. The computation of betweenness is not feasible –or at least very difficult– on the whole network, so we will limit the calculation to the SAP set, precisely to the induced subgraph. It contains, as we have seen, the structures responsible for most of the network vulnerability.

In short, the heuristics we test to solve the *MAX-k-DELTA-REACH* problem are:

- *Ranking out-degree*: selects the k elements with the highest out-degree (number of direct dependents) in the network.

- *SAP + Ranking out-degree*: selects the k elements with the highest out-degree within the SAP set of the largest SCC of the network.

- *SAP + Ranking betweenness*: selects the k elements with the highest betweenness centrality within the SAP set of the largest SCC of the network.

Table 3 shows the results of the application of the proposed heuristics for immunization target sets of sizes $k = 1, 10, 200$ and 1000, on the two supercritical reference networks considered. We highlight the fact that a relatively good immunization set of size k can be obtained simply by selecting the k nodes with the highest out-degree centrality. In general, it appears that this result can be slightly improved by limiting the selection to the SAP set.

Table 3: Immunization results on reference networks of target sets of size k obtained using different heuristics. For each value of k and each repository, three columns headed by shorthand symbols are shown. If we consider in each case a network G and an immunization set S of size k, we write Δ for $\Delta_G^{Reach}(S)$, the reduction in vulnerability of G as a result of S immunization, Δ/ϕ for $100 \cdot \Delta_G^{Reach}(S)/\phi^{Reach}(G)$, the ratio of reduction in vulnerability relative to the original vulnerability (shown in percentage), Δ/k for $\Delta_G^{Reach}(S)/k$, the vulnerability reduction per immunized package. Values in bold correspond to the maximum reduction in vulnerability (when two or more heuristics are tested). The absence of values in some cells corresponds to k equal to the sizes of the SAP in Maven ($k = 351$) and npm ($k = 5902$) because there is no point to use rankings when the whole SAP is immunized. Similarly, it makes no sense SAP+rankings heuristics with $k = 1000$ in Maven because the size of the SAP is 351.

	Maven			**npm**		
n	126752			1074508		
ϕ^{Reach}	1805.54			27193.83		
k=1	Δ	Δ/ϕ	Δ/k	Δ	Δ/ϕ	Δ/k
Ranking out-degree	0.88	0.05	0.88	**175.56**	**0.65**	**175.56**
SAP + Ranking out-degree	**67.22**	**3.72**	**67.22**	**175.56**	**0.65**	**175.56**
SAP + Ranking betweenness	23.19	1.28	23.19	**175.56**	**0.65**	**175.56**
k=10	Δ	Δ/ϕ	Δ/k	Δ	Δ/ϕ	Δ/k
Ranking out-degree	190.87	10.57	19.09	590.40	2.17	59.04
SAP + Ranking out-degree	235.78	13.06	23.58	**650.33**	**2.39**	**65.03**
SAP + Ranking betweenness	**471.43**	**26.11**	**47.14**	483.98	1.78	48.40
k=200	Δ	Δ/ϕ	Δ/k	Δ	Δ/ϕ	Δ/k
Ranking out-degree	1 489.70	82.51	7.45	4 211.30	15.49	21.06
SAP + Ranking out-degree	**1 682.74**	**93.20**	**8.41**	4 624.35	17.01	23.12
SAP + Ranking betweenness	1 671.78	92.59	8.36	**6 755.98**	**24.84**	**33.78**
k=351	Δ	Δ/ϕ	Δ/k			
SAP	1 695.32	93.90	4.83	—	—	—
k=1000	Δ	Δ/ϕ	Δ/k			
Ranking out-degree	**1 782.31**	**98.71**	**1.78**	12 663.66	46.57	12.66
SAP + Ranking out-degree	—	—	—	12 637.78	46.47	12.64
SAP + Ranking betweenness	—	—	—	**16 995.20**	**62.50**	**17.00**
k=5902				Δ	Δ/ϕ	Δ/k
SAP	—	—	—	24 702.74	90.84	4.19

In some cases, considerably better figures can be obtained by employing betweenness centrality, but not always. Overall, the markedly heterogeneous results suggest that either each network and/or each value of k require a different heuristic, or that there are much more efficient methods than those used here to select immunization sets, which would explain the variability obtained. In any case, it seems clear that the immunization performance per package decreases as k increases. Although our methods are approximate, this suggests that most of the network vulnerability is generated by a small combination of packages.

In any case, some of the results of this exploratory analysis are remarkable. For example, using *SAP+Ranking betweenness* we have reduced the vulnerability of Maven by 26.11% by immunizing only 10 packages out of more than 125000, or by 62.5% the vulnerability of npm by acting on one thousandth of its 1074508 packages.

5 Discussion

The repository vulnerability based on the cost function *Reach* shows how sensitive a repository is to the introduction of random defects. Considering the large number of failures that can affect the functionality of a repository, e.g. dependency issues [21], security vulnerabilities [12, 13], the assumption of independent and random errors seems to be reasonable.

We can state that in repositories with high vulnerability, e.g. Maven and npm, defects inevitably introduced at different stages of package development propagate more easily through the dependency structure, potentially affecting a larger number of other packages. For example, we see these dependency problems with packages updates [21, 63].

The causes of vulnerability are related to the structure of the dependency network as some authors have already pointed out [21]. Our work goes one step further and proves that within the complexity of this network, the existence of a large set of packages that are transitively dependent on each other, i.e., a large SCC, explains most of the vulnerability of a repository. In fact, the vulnerability appears to be of the order of the size of the largest SCC present, moreover this fact seems to be related to the asymmetric bow-tie structure already commented on. This is a crucial conclusion from which important implications can be drawn. These vary depending on the role one plays in the management, development and use of repositories. We distinguish four roles, i.e., the repository manager, the package developer, the user and the scholar, which are discussed below.

We have empirically learned that the size of SCCs is directly related to the vulnerability of a repository. The existence of an SCC significantly larger than $log(n)$ in a repository of size n is reason for concern. Our results suggest that repository managers should prevent the growth of dependency vulnerability by impeding or slowing down the emergence of a large SCC. This task would limit the propagation of defects and can be achieved by implementing cycle control measures, such as prohibiting or subjecting to special requirements the release of packages whose inclusion in the repository creates a cycle of dependencies or promotes the growth of complex cyclic structures. It can also be useful to involve the community by publishing code guides and specific tools, helping developers understand their individual responsibility for the health of the repository.

On the other hand, in repositories that already show a significantly large

SCC, the probability that random errors affect a large number of packages could be reduced by decreasing the likelihood that the packages that contribute most to the vulnerability contain or propagate defects. Clearly, it is not possible to make this probability zero, as we have assumed in our theoretical model of immunization, but it would be worth dedicating special resources or prioritizing those currently devoted to preventive actions that could reduce this probability. We must also make the important point that discussing these critical packages contributing significantly to the vulnerability only makes sense once the size of the target set to be immunized has been fixed. The problem of immunizing the dependency network is combinatorial in nature: for each target set size there are one or more optimal solutions that do not necessarily share elements between them and with other set sizes.

We can think of low-cost preventive measures, such as awareness campaigns for developers or running automated code checks. In these actions, the largest SCC in the repository or its SAP subset are targets to consider. Both sets of packages can be efficiently obtained using well-known algorithms. Our approach allows calculating the theoretical maximum reduction of the vulnerability, although the impact will be smaller in the real repositories. Even so, these packages are the ones that should reasonably be considered for these preventive actions.

It is also feasible to employ costly but very effective measures, such as human code reviews or audits to reduce the probability of defect presence and propagation in some specific packages. The option of providing additional financial or human resources to development teams could also be considered, in line with recent initiatives such as the OSSF "Securing Critical Packages" technical working group, that seeks to identify "critical projects we all rely on" and provide them with enhanced support [25]. These actions should focus on (i) detecting or preventing defects in a package and its transitive dependencies or (ii) reducing the number of direct package dependencies. In this case, it is necessary to select a small set of key packages. We have shown that this problem is computationally intractable and must be tackled with approximate methods. We have proposed an initial set of techniques that allow significant reductions of vulnerability for such small sets.

There are other interesting implications of the results for repository users and the software industry. Developers that use software repositories should be aware of the problems related to the package dependency, among other reasons, because the growth of the dependencies seems to be exponential in many repositories and therefore the potential issues related to them [63].

In this context, it is easier to understand the growing interest in the development of tools for scanning software, for example assessing vulnerability to security issues in the dependencies of a project, e.g. npm-audit, Eclipse Steady [64]. Recalling that our approach to vulnerability goes beyond security failures, some metrics proposed in our work could complement any vulnerability analysis. Moreover, we can combine these measures with other statistics, such as defect introduction rates, to compare different repositories. This could help decision-makers in the software industry to assess the risks associated with using a repository, by quantifying the probability that packages imported into a project incorporate defects.

Finally, we would discuss some research questions that arise from results and can be interesting for scholars. We have seen that vulnerability is significantly conditioned by the presence of a giant SCC in the dependency network. Not all repositories have a dominant SCC, in our work, for example, we have proved that there is a giant SCC in Maven and npm but not in PyPI. The question is to explain the causes of the appearance of a large SCC. Our results show that the size of the repository is correlated to the existence of a giant SCC, but this fact does not mean that size is the cause. As in other networked phenomena already studied [65], we can expect that there is a critical point in the evolution of a repository that separates the absence/existence of a giant SCC. According to our results, PyPI would not have passed this point while Maven and npm would have. However, we do not know the formation mechanism for this type of dependency network that could be related to the practices and behaviour of the community of developers [17, 66, 67]. The possible answers to this question are also of interest to managers and developers, who, as we have already pointed out, have important incentives to avoid the emergence of this sort of structure.

The results of the three repositories analysed should be discussed in the light of the assumptions made in the construction of their models. This leads to some limitations in their interpretation. Thus, the vulnerability measures should be interpreted as upper bounds of the real values on the reference of the historical snapshot used. These values, although cautious, can provide an order of magnitude on the cost implications of dependencies, for example, as a proxy of the revision costs that a bug in a package can cause in the development of a software project. More accuracy could be achieved by modelling the network using versioning and dependency resolution, considering function-level calls, usage patterns, semantic dependency types, package release dynamics or any other concept [68]. However, taking into account the complexity of solving the dependency problem [54] grows with the level of detail and must be resolved by using approximate techniques that do not guarantee a unique solution, which may hinder the interpretation and generalisation of the results.

Another interesting question is the optimization problem of choosing a group of packages that maximizes the reduction of vulnerability. We have proved the complexity of this problem and shown that by using simple heuristics it is feasible to get quite good results. However, there is room for improvement, considering the size of some repositories, e.g. npm. Further study of this optimization problem would be necessary. The development of more sophisticated and efficient heuristics would be of interest to repository managers.

The asymmetric bow-tie structure in Maven and npm raises an interesting question about the nature of this macroscopic pattern. The SCC may correspond to packages with general or abstract functionalities. Its global utility increases the probability of cycles, since the implemented functionality will end up incorporated directly or transitively in a large number of packages, including those that eventually fall within the SCC. The OUT module may correspond to more specialized packages less likely to be reused. Since in a given domain there are by definition fewer possible functionalities that are abstract than specialized, this argument could also explain a limiting factor to the growth of the largest SCC. Clearly the concept of functionality is very dependent on the type of repository and would require further research to validate this hypothesis.

There would be a final research question related to the nature of failures and errors in repositories. All results of this work are based on the assumption of random errors. The analysis could be different if we considered intentional failures that, for example, pretended to cause damage to the repository [69].

The study of the vulnerability of software package repositories to this type of attack is also an interesting research line.

6 Conclusions

The widespread use of centralised package repositories in software development is of increasing concern in a world that is entirely dependent on software products. In our view, this concern is not caused by the massive reuse of components, which is desirable and has enabled enormous technological development in this century. The underlying issue is the pressing sense that there are no clearly defined processes or controls in place to assure quality, nor do we have a fundamental understanding of the structure and dynamics of these repositories on which we critically depend.

With this work, we want to join the growing number of voices warning of the situation and the urgent need for appropriate action. Our contribution is a formal model which analyses the vulnerability to failures of software repositories, in a conceptually similar way as previous work has modelled the vulnerability of other complex technological, biological or social networks. By studying the potential for random defect replication through the dependency structure, our model makes it possible to quantify the vulnerability of any repository, carry out comparative studies, and explain why some repositories are much more vulnerable than others. Our vulnerability model could be applied to other phenomena that share a similar dependency structure, for example, metabolic networks [70] financial networks [71], and distribution networks [72].

We have shown in the paper that simple metrics are not sufficient to explain the vulnerability of a software repository due to the structure of the dependency network. Most of the vulnerability is due to the presence of a large strongly connected component (SCC), a set of packages in which all depend transitively on each other. Our knowledge of other complex networks leads us to suspect that this structure does not appear gradually, but that there is a breaking point where a large SCC emerges rapidly, causing the vulnerability of the network to skyrocket. We have discussed some approaches for repository managers to detect and prevent the emergence of a large SCC. Also, if it is present, to reduce its size, which can drastically cut down the propagation of defects within the repository and to external projects.

We hope our results will be useful not only to repository managers, but to many of the agents that interact with repositories, such as package developers, final users and scholars. Still, we acknowledge their limitations and propose some possible lines of future work, such as higher granularity representations, the use of weighted cost functions or the study of vulnerability to attacks. We have also formulated some open research questions such as the causes of the emergence of a large SCC and the optimization problem of choosing a group of packages that maximizes the reduction of vulnerability.

Acknowledgment

The authors acknowledge financial support from the Spanish Ministry of Science, Innovation and Universities (excellence network RED2018-102518-T), the Spanish State Research Agency (PID2020-119894GB-I00 and PID2020-118906GB-I00/AEI/10.13039/501100011033) and the Junta de Castilla y León, Consejería de Educación through BDNS 425389.

References

[1] J. Sametinger, *Software engineering with reusable components.* Springer Science & Business Media, 1997.

[2] A. Lane, "Analysis of the 2014 Open Source Development and Application Security Survey," 2014. [Online]. Available: https://securosis.com/research/publication/analysis-of-the-2014-open-source/-development-and-application-security-surve

[3] P. Boldi, "How network analysis can improve the reliability of modern software ecosystems," in *Proceedings - 2019 IEEE 1st International Conference on Cognitive Machine Intelligence, CogMI 2019*, 2019.

[4] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 475–488.

[5] A. Miller and S. Zitzman, "The State of Open Source Security 2020 \ Snyk," 2020. [Online]. Available: https://snyk.io/open-source-security/

[6] G. Kou, Y. Shi, and G. Dong, "Data mining for software trustworthiness," *Information Sciences*, vol. 191, pp. 1–2, 2012, data Mining for Software Trustworthiness.

[7] D. Setó-Rey, J. I. Santos-Martín, and C. López-Nozal, "dsr0018/olivia: OLIVIA - Open-source Library Indexes Vulnerability Identification and Analysis," Nov. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7358391

[8] Tidelift, "Libraries.io Open Data," 2020. [Online]. Available: https://libraries.io/data

[9] M. Jang, *Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It.* O'Reilly Media, Inc., Apr. 2006.

[10] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, Jun. 2009.

[11] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," 2020.

[12] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 446–457.

[13] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, no. 4, p. 59, Apr 2021.

[14] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 351–361.

[15] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the untriviality of trivial packages: An empirical study of npm javascript packages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[16] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, "Analysing Software Repositories to Understand Software Evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Berlin, Heidelberg: Springer, 2008, pp. 37–67.

[17] 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 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–120.

[18] L. Šubelj and M. Bajec, "Software systems through complex networks science: Review, analysis and applications," in *Proceedings of the First International Workshop on Software Mining*, ser. SoftwareMining '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 9–16. [Online]. Available: https://doi.org/10.1145/2384416.2384418

[19] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, p. 046116, Oct 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.68.046116

[20] L. Šubelj and M. Bajec, "Community structure of complex software systems: Analysis and applications," *Physica A: Statistical Mechanics and its Applications*, vol. 390, no. 16, pp. 2968–2975, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037843711100269X

[21] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[22] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[23] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 288–299.

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

[25] O. S. S. F. (OpenSSF), "Wg securing critical projects," 2020. [Online]. Available: https://github.com/ossf/wg-securing-critical-projects

[26] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, "Complex networks: Structure and dynamics," *Physics Reports*, vol. 424, no. 4, pp. 175–308, 2006.

[27] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *nature*, vol. 406, no. 6794, pp. 378–382, 2000.

[28] J.-P. Onnela, J. Saramäki, J. Hyvönen, G. Szabó, D. Lazer, K. Kaski, J. Kertész, and A.-L. Barabási, "Structure and tie strengths in mobile communication networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 104, no. 18, pp. 7332–7336, May 2007.

[29] D. S. Callaway, M. E. Newman, S. H. Strogatz, and D. J. Watts, "Network robustness and fragility: Percolation on random graphs," *Physical review letters*, vol. 85, no. 25, p. 5468, 2000.

[30] R. Pastor-Satorras and A. Vespignani, *Evolution and Structure of the Internet: A Statistical Physics Approach.* Cambridge University Press, Jul. 2007.

[31] T. Gross and L. Barth, "Network robustness revisited," *arXiv preprint arXiv:2202.07911*, 2022.

[32] S. Majhi, M. Perc, and D. Ghosh, "Dynamics on higher-order networks: A review," *Journal of the Royal Society Interface*, vol. 19, no. 188, p. 20220043, 2022.

[33] A. Barrat, M. Barthélemy, and A. Vespignani, *Dynamical Processes on Complex Networks.* Cambridge University Press, 2008.

[34] M. Newman, *Networks: An Introduction.* USA: Oxford University Press, Inc., 2010.

[35] A.-L. Barabási and M. Pósfai, *Network science.* Cambridge: Cambridge University Press, 2016.

[36] M. Jusup, P. Holme, K. Kanazawa, M. Takayasu, I. Romić, Z. Wang, S. Geček, T. Lipić, B. Podobnik, L. Wang *et al.*, "Social physics," *Physics Reports*, vol. 948, pp. 1–148, 2022.

[37] M. E. Newman, “Spread of epidemic disease on networks,” *Physical review E*, vol. 66, no. 1, p. 016128, 2002.

[38] G. E. Leventhal, A. L. Hill, M. A. Nowak, and S. Bonhoeffer, “Evolution and emergence of infectious diseases in theoretical and real-world networks,” *Nature Communications*, vol. 6, no. 1, p. 6101, Jan. 2015.

[39] D. J. Watts, “A simple model of global cascades on random networks,” *Proceedings of the National Academy of Sciences*, vol. 99, no. 9, pp. 5766–5771, 2002.

[40] P. Gai and S. Kapadia, “Contagion in financial networks,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 466, no. 2120, pp. 2401–2423, Aug. 2010.

[41] V. Bala and S. Goyal, “Learning from neighbours,” *The review of economic studies*, vol. 65, no. 3, pp. 595–621, 1998.

[42] M. O. Jackson and Y. Zenou, “Chapter 3 - Games on Networks,” in *Handbook of Game Theory with Economic Applications*, H. P. Young and S. Zamir, Eds. Elsevier, Jan. 2015, vol. 4, pp. 95–163.

[43] F. Vega-Redondo, *Complex social networks.* Cambridge University Press, 2007, no. 44.

[44] P. Purdom, “A transitive closure algorithm,” *BIT Numerical Mathematics*, vol. 10, no. 1, pp. 76–94, 1970.

[45] A. Goralčíková and V. Koubek, “A reduct-and-closure algorithm for graphs,” in *International Symposium on Mathematical Foundations of Computer Science.* Springer, 1979, pp. 301–307.

[46] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[47] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, “An analysis of approximations for maximizing submodular set functions—I,” *Mathematical programming*, vol. 14, no. 1, pp. 265–294, 1978.

[48] B. Goldengorin, “Maximization of submodular functions: Theory and enumeration algorithms,” *European Journal of Operational Research*, vol. 198, no. 1, pp. 102–112, 2009.

[49] T. T. S. Q. Company., “Tiobe index.” 2021. [Online]. Available: https://www.tiobe.com/tiobe-index/

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

[51] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, “On the impact of using trivial packages: an empirical case study on npm and pypi,” *Empirical Software Engineering*, vol. 25, no. 2, pp. 1168–1204, Mar 2020.

[52] F. Nagle, J. Wilkerson, J. Dana, and J. L. Hoffman, "Vulnerabilities in the core.preliminary report and census ii of open source software," The Linux Foundation & The Laboratory for Innovation Science at Harvard (LISH), Tech. Rep., 2020. [Online]. Available: https://www.coreinfrastructure.org/programs/census-program-ii/

[53] O. Management Group, "Omg unified modeling language – version 2.5.1," https://www.omg.org/spec/UML/2.5.1, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[54] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli, "Dependency solving is still hard, but we are getting better at it," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2020, pp. 547–551.

[55] I. Palásti, "On the strong connectedness of directed random graphs," *Studia Sci. Math. Hungar*, vol. 1, pp. 205–214, 1966.

[56] P. Erdös and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[57] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, no. 1, pp. 309–320, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128600000839

[58] R. Tanaka, M. Csete, and J. Doyle, "Highly optimised global organisation of metabolic networks," *IEE Proceedings - Systems Biology*, vol. 152, pp. 179–184(5), December 2005. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/ip-syb_20050042

[59] R. Yang, L. Zhuhadar, and O. Nasraoui, "Bow-tie decomposition in directed graphs," in *14th International Conference on Information Fusion*, 2011, pp. 1–5.

[60] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations.* Springer, 1972, pp. 85–103.

[61] G. F. Italiano, L. Laura, and F. Santaroni, "Finding strong bridges and strong articulation points in linear time," *Theoretical Computer Science*, vol. 447, pp. 74–84, 2012.

[62] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[63] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Engg.*, vol. 20, no. 5, p. 1275–1317, Oct. 2015.

[64] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, Sep 2020.

[65] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin, "Giant strongly connected component of directed networks," *Physical Review E*, vol. 64, no. 2, Jul 2001.

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

[67] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021.

[68] P. Boldi and G. Gousios, "Fine-grained network analysis for modern software ecosystems," *ACM Trans. Internet Technol.*, vol. 21, no. 1, Dec. 2020.

[69] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," 2019.

[70] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, Oct 2000. [Online]. Available: https://doi.org/10.1038/35036627

[71] D. Y. Kenett, T. Preis, G. Gur-Gershgoren, and E. Ben-Jacob, "Dependency network and node influence: Application to the study of financial markets," *International Journal of Bifurcation and Chaos*, vol. 22, no. 07, p. 1250181, 2012.

[72] P. R. Garvey and C. A. Pinto, "Introduction to functional dependency network analysis," in *The MITRE Corporation and Old Dominion, Second International Symposium on Engineering Systems, MIT, Cambridge, Massachusetts*, vol. 5, 2009.