A Strategic Model of Software Dependency Networks*

Cornelius Fritz[1], Co-Pierre Georg[2], Angelo Mele[3], and Michael Schweinberger[1]

[1]The Pennsylvania State University – Department of Statistics

[2]University of Cape Town

[3]Johns Hopkins University – Carey Business School

Modern software development involves collaborative efforts and reuse of existing code, which reduces the cost of developing new software. However, reusing code from existing packages exposes coders to vulnerabilities in these dependencies. We study the formation of dependency networks among software packages and libraries, guided by a structural model of network formation with observable and unobservable heterogeneity. We estimate costs, benefits, and link externalities of the network of 696,790 directed dependencies between 35,473 repositories of the Rust programming language using a novel scalable algorithm. We find evidence of a positive externality exerted on other coders when coders create dependencies. Furthermore, we show that coders are likely to link to more popular packages of the same software type but less popular packages of other types. We adopt models for the spread of infectious diseases to measure a package's systemicness as the number of downstream packages a vulnerability would affect. Systemicness is highly skewed with the most systemic repository affecting almost 90% of all repositories only two steps away.

*This version: February 20, 2024. Contact: cfritz@psu.edu, co-pierre.georg@uct.ac.za, angelo.mele@jhu.edu, and michael.schweinberger@psu.edu. We wish to thank participants at the Network Science in Economics 2023 at Virginia Tech, 2023 BSE Summer Forum Networks Workshop, Southern Economic Association Annual Meeting 2023 as well as seminar participants at Georgetown University and UPenn for helpful comments and suggestions. The authors acknowledge support from the German Research Foundation award DFG FR 4768/1-1 (CF), Ripple's University Blockchain Research Initiative (CPG), the U.S. Department of Defense award ARO W911NF-21-1-0335 (MS, CF), and the U.S. National Science Foundation awards NSF DMS-1513644 and NSF DMS-1812119 (MS). Mele acknowledges support from an IDIES Seed Grant from the Institute for Data Intensive Engineering and Science at Johns Hopkins University and partial support from NSF grant SES 1951005.

Lastly, we show that protecting only the ten most important repositories reduces vulnerability contagion by nearly 40%.

Keywords: software development, dependency graphs, strategic network formation, exponential random graphs

JEL Classification: D85, L17, L86, O31

1 Introduction

Modern software development is a collaborative effort that results in sophisticated software packages that makes extensive use of already existing software packages and libraries (Schueller et al., 2022). This results in a complex network of dependencies among software packages, best described as dependency graphs.[1] While this results in significant efficiency gains for software developers (Lewis et al., 1991; Barros-Justo et al., 2018), it also increases the risk that a vulnerability in one software package renders a large number of other software packages also vulnerable. In line with this view, the number of vulnerabilities listed in the Common Vulnerabilities and Exposure Database has increased almost twelve-fold between 2001 and 2019.[2] A recent estimate by Krasner (2018) put the cost of losses from software failures at over USD 1 Trillion, up from USD 59 Billion in 2002, estimated by the National Institute of Standards and Technology (2002).[3]

Because vulnerabilities can spread from one software package to all packages depending on it, coders can create an externality when deciding to re-use code from existing packages rather

[1] Dependency graphs have been studied for many different programming languages using information from package managers. For an empirical comparison of the most common ones, see Decan et al. (2019).

[2] The Common Vulnerabilities and Exposure database is a public reference for known information-security vulnerabilities which feeds into the U.S. National Vulnerability Database. See `https://cve.mitre.org/`. The reported number of incidents has increased from 3,591 for the three years from 1999 to 2001 to 43,444 for the three-year period from 2017 to 2019.

[3] Industry group Cybersecurity Ventures estimates that the damages incurred by all forms of cyber crime, including the cost of recovery and remediation, totalled USD 6 Trillion in 2021, and could reach USD 10.5 Trillion annually by 2025 (Source).

than implementing the required functionality themselves. Whether this is the case and how large the externality is, is ultimately an empirical question. We model coders' decisions to create a dependency on another software package as an equilibrium game of network formation with observed and unobserved heterogeneity. The equilibrium characterization provides a likelihood of observing a particular architecture of the software dependency network in the long run. Using data on open source software projects of the Rust programming language, we obtain all 696,790 dependencies among 35,473 repositories of software projects contained in Rust's software package manager Cargo. We find evidence that a maintainer's decision to allow a dependency exerts a positive externality on other maintainers: Highly interdependent software packages are likely to become even more interdependent. This means that it is particularly important to ensure that such packages are free of vulnerabilities that potentially can affect a large number of other software packages.

Open source software is an ideal laboratory to study software dependencies. Since the actual source code of a software package can be inspected and modified by others, code reuse is not only possible, it is encouraged. Modern software packages are not developed in isolation, but within an ecosystem of interdependent packages. A package manager keeps track of code dependencies and ensures that these are automatically resolved so that when a user installs one package, all of the package's dependencies are also installed and automatically satisfied. There are many programming languages which provide software using package managers. We restrict our analysis to repositories of software packages written in the programming language Rust and managed by the Cargo package manager.[4] Rust is a modern programming language that allows particularly safe software development without compromising performance.[5] We use data from `https://libraries.io`, which, for each software repository includes a full list of all dependencies.

We model the development of software as a process where coders reuse existing code, creating a network of dependencies among software packages—organized in repositories, which can

[4] Cargo is Rust's official package manager. See here for the official documentation.

[5] Rust has been voted the most loved programming language by developers on the popular coding knowledge exchange website `http://stackoverflow.com`.

include several closely related software packages—in the process. This is typical for today's prevalent object-oriented software development paradigm. Our model describes a system of N individual software repositories. Each repository is managed by a group of coders who decide which dependencies to form. Coders obtain a net benefit of linking to another package which depends on how active this dependency is maintained, how mature, popular, and large it is. We also allow a coders' utility to be affected by a local, i.e. type-specific, externality: since dependent packages have dependencies themselves, they are susceptible to vulnerabilities imported from other packages. So we assume that coders care about the direct dependencies of the packages they link to. This specification excludes externalities that are more than two links away as in the network formation models of DePaula et al. (2018), Mele (2017) and Mele and Zhu (forthcoming). The network of dependencies forms over time and in each period a randomly selected package needs an update, so the coders decide whether to form a link or update the software in-house. Before updating the link, the package receives a random match quality shock. We characterize the equilibrium distribution over networks as a mixture of exponential random graphs (Schweinberger and Handcock, 2015; Mele, 2022), which can be decomposed into within- and between-type contribution to the likelihood.

Estimation of this model is complicated because the likelihood depends on a normalizing constant that is infeasible to compute in large networks. Moreover, the model's unobserved heterogeneity has to be integrated out in the likelihood, thus further complicating the computations. To alleviate these problems, we resort to an approximate two-step approach. In the first step, we estimate the discrete types of the nodes through approximations of the likelihood via a variational mean-field algorithm for stochastic blockmodels (Vu et al., 2013). In the second step, we estimate the structural payoff parameters using a fast Maximum Pseudo-Likelihood Estimator (MPLE), conditioning on the estimated types (Babkin et al., 2020; Dahbura et al., 2021).[6]

Our main result is that coders exert a positive externality on other coders when creating a dependency on another package. Other coders are then also more likely to create a dependency with that package. As a result, packages that are already highly interdependent tend to become

[6]These methods are implemented in a highly scalable open source `R` package `bigergm`, available on Github at: `https://github.com/sansan-inc/bigergm`. See the Appendix for details on the implementation.

even more interdependent. This increases the risk that a vulnerability in a single package has large adverse consequences for the entire ecosystem. One example of how a vulnerability in one package impacted a significant part of critical web infrastructure is Heartbleed, the infamous bug in the widely used SSL/TLS cryptography library, resulting from an improper input variable validation (Durumeric et al., 2014). When Heartbleed was disclosed, it rendered up to 55% of secure webservers vulnerable to data theft, including of the Canadian Revenue Agency, and Community Health Systems, a large U.S. hospital chain. Our model provides a further argument for ensuring the security of highly interdependent software packages. Not only can a vulnerability in such a package affect a large fraction of the entire ecosystem, because of the externality we identify, it is also likely that this fraction increases as time goes on.

Another important question in the study of network formation processes is whether linked packages are similar or dissimilar with respect to their covariates. We find evidence of similarity among the same type of software packages in terms of their Size, Popularity, and Maturityas well as between different-type software packages in terms of their Size. We further find evidence of *dis*-similarity between different-type software packages in terms of their Popularity, Num.Contributors (as an alternative popularity measure), and Num.Forks. In other words, mature, large and popular software packages of one type are likely to depend on similar packages of the same type, but on less popular and mature software packages of another type. This is intuitive, because mature, popular, and large packages are likely to have a large user base with high expectations of the software and coders try to satisfy this demand by providing sophisticated functionality with the help of mature, popular, and large software packages of the same type. These packages are often built using a lot of low-level functionality from large but fairly generic libraries, which is why large popular and mature software packages of one type depend on larger, but less popular and less mature software packages of another type. One example of this is the inclusion of a payment gateway in an e-commerce application. The e-commerce application itself can be sophisticated and complex, like ebay is in the provision of their auction mechanism. For such an application it is particularly valuable to provide users with the ability to use a variety of different payment methods, including credit card, paypal, or buy now pay later solutions like Klarna. Our result is also in line with the recent trend in software de-

velopment away from large monolithic applications and towards interconnected microservices (Traore et al., 2022).

Lastly, we adapt the susceptible-infected-recovered (SIR) model from epidemiology to examine the spread of vulnerabilities in the Rust Cargo dependency graph. We measure a repository's *k-step systemicness* as the number of downstream packages that are potentially rendered vulnerable by a vulnerability in the upstream repository. We find that systemicness is highly skewed across repositories: While the average vulnerable repository affects 52 repositories two steps away, the most systemic repository affects 26,631 repositories. This heterogeneity underscores the existence of key nodes within the software dependency graph whose compromise could lead to widespread vulnerability exposure. We then study the efficacy of targeted interventions to mitigate vulnerability spread, drawing parallels with vaccination strategies in public health. By focusing on securing the most critical nodes—determined by a combination of betweenness centrality and expected fatality—we assess the potential to significantly reduce the risk of vulnerability contagion. Our findings suggest that protecting the top ten most critical repositories can reduce the extent of vulnerability contagion by nearly 40%.

Our paper relates to several strands of literature in both economics and computer science. First, our paper contributes to a growing literature on open source software, which has been an interest of economic research.[7] In an early contribution, Lerner and Tirole (2002) argue that coders spend time developing open source software—for which they are typically not compensated—as a signal of their skills for future employers. Likewise, one reason why companies contribute to open source software is to be able to sell complementary services. Open source projects can be large and complex, as Zheng et al. (2008) point out. They study dependencies in the Gentoo Linux distribution, and show that the resulting network is sparse, has a large clustering coefficient, and a fat tail. They argue that existing models of network growth do not capture the Gentoo dependency graph well and propose a preferential attachment model as alternative.[8]

[7] For an overview of the broad literature in the emerging field of digital economics, see Goldfarb and Tucker (2019).

[8] In earlier work, LaBelle and Wallingford (2004) study the dependency graph of the Debian Linux distribution and show that it shares features with small-world and scale-free networks. However, LaBelle and Wallingford (2004) do not strictly check how closely the dependency graph conforms with either network growth model.

The package manager model is nowadays adopted by most programming languages which makes it feasible to use dependency graphs to study a wide variety of settings. Kikas et al. (2017), for example, study the structure and evolution of the dependency graph of `JavaScript`, `Ruby`, and also Rust. The authors emphasize that dependency graphs of all three programming languages have become increasingly vulnerable to the removal of a single software package. An active literature studies the network structure of dependency graphs (Decan et al., 2019) to assess the vulnerability of a software ecosystem (see, for example, Zimmermann et al. (2019)).

These papers show the breadth of literature studying open source ecosystems and dependency graphs. However, the literature considers the network either as stochastic or even as static and given. By contrast, we model the formation of dependencies as coders' *strategic* choice in the presence of various and competing mechanisms that either increase or reduce utility.[9]

The theoretical literature on strategic network formation has pointed out the role of externalities in shaping the equilibrium networks (Jackson, 2008; Jackson and Wolinsky, 1996). Estimating strategic models of network formation is a challenging econometric task, because the presence of externalities implies strong correlations among links and multiple equilibria (Mele, 2017; Snijders, 2002; DePaula et al., 2018; Chandrasekhar, 2016; DePaula, 2017; Boucher and Mourifie, 2017; Graham, 2017, 2020). In this paper, we model network formation as a sequential process and focus on the long-run stationary equilibrium of the model (Mele, 2017, 2022). Because the sequential network formation works as an equilibrium selection mechanism, we are able to alleviate the problems arising from multiple equilibria.

Adding unobserved heterogeneity further complicates identification, estimation and inference (Schweinberger and Handcock, 2015; Graham, 2017; Mele, 2022). Other works have considered conditionally independent links without externalities (Graham, 2017, 2020; DePaula, 2017; Chandrasekhar, 2016), providing a framework for estimation and identification in random and fixed effects approaches. On the other hand, because link externalities are an important feature in this contexts, we move away from conditionally independent links, and model nodes' unob-

[9] Blume et al. (2013) study how possibly contagious links affect network formation in a general setting. While we do not study the consequences of the externality we identify for contagion, this is a most worthwhile avenue for future research.

served heterogeneity as discrete types, whose realization is independent of observable characteristics and network, in a random effect approach. We can thus adapt methods of community discovery for random graphs to estimate the types, following approximations of the likelihood suggested in Babkin et al. (2020) and improved in Dahbura et al. (2021) to accomodate for observed covariates. Our two-steps method scales well to large networks, thus improving the computational challenges arising in estimation of these complex models (Boucher and Mourifie, 2017; Vu et al., 2013; Bonhomme et al., 2019).

Our model is able to identify externalities as well as homophily (Currarini et al., 2010; Jackson, 2008; Chandrasekhar, 2016), the tendency of individuals to form links to similar individuals. On the other hand, our model can also detect heterophily (or competition) among maintainers. We also allow the homophily to vary by unobservables, while in most models the homophily is estimated only for observable characteristics (Currarini et al., 2010; Schweinberger and Handcock, 2015; DePaula et al., 2018; Chandrasekhar, 2016; Graham, 2020).

2 A network description of code

2.1 Nomenclature and definitions

The goal of computer programs is to implement algorithms on a computer. An algorithm is a terminating sequence of operations which takes an input and computes an output using memory to record interim results.[10] We use the term broadly to include algorithms that rely heavily on user inputs and are highly interactive (e.g. websites, spreadsheet and text processing software, servers). To implement an algorithm, programming languages need to provide a means of reading input and writing output, have a list of instructions that can be executed by a computer, and provide a means of storing values in memory. More formally:

Definition 1. *Code is a sequence of operations and arguments that implement an algorithm. A*

[10] Memory to record interim instructions is sometimes called a "scratch pad", in line with early definitions of algorithms which pre-date computers. See, for example, Knuth (1997) (Ch.1).

computer program is code that can be executed by a computer system.

In order to execute a program, a computer provides resources—processing power and memory—and resorts to a compiler or interpreter, which in themselves are computer programs.[11]

Software developers, which we refer to as coders, use programming languages to implement algorithms. Early programs were written in programming languages like `FORTRAN` and later in `C`, both of which adhere to the *procedural programming* paradigm. In this paradigm, code is developed via procedures that can communicate with each other via calls and returns.

Definition 2. *A procedure is a sequence of programming instructions, which make use of the resources provided by the computer system, to perform a specific task.*

Procedures are an integral tool of almost all programming languages because they allow a logical separation of tasks and abstraction from low-level instructions, which greatly reduces the complexity of writing code. We use the term procedure broadly and explicitly allow procedures to return values. The first large software systems like the UNIX, Linux, and Windows operating systems or the Apache web server have been developed using procedural programming.

Today, however, the dominant modern software development paradigm is *Object-oriented programming* (OOP).[12] Under this paradigm, code is developed primarily in *classes*. Classes contain data in the form of variables and data structures as well as code in the form of procedures (which are often called methods in the OOP paradigm). Classes can communicate with one another via procedures using calls and returns.

[11] The term "compiler" was coined by Hopper (1952) for her arithmetic language version 0 (A-0) system developed for the UNIVAC I computer. The A-0 system translated a program into machine code which are instructions that the computer can execute natively. Early compilers were usually written in machine code or assembly. Interpreters do not translate program code into machine code, but rather parse it and execute the instructions in the program code directly. Early interpreters were developed roughly at the same time as early compilers, but the first widespread interpreter was developed in 1958 by Steve Russell for the programming language LISP (see McCarthy (1996).

[12] For a principal discussion of object-oriented programming and some differences to procedural programming, see Abelson et al. (1996). Kay (1993) provides an excellent historical account of the development of early object-oriented programming.

Definition 3. *A class is a code template defining variables, procedures, and data structures. An object is an instance of a class that exists in the memory of a computer system.*

Access to a computer's memory is managed in most programming languages through the use of variables, which are memory locations that can hold a value. A variable has a scope which describes where in a program text a variable can be used.[13] The extent of a variable defines when a variable has a meaningful value during a program's run-time. Depending on the programming language, variables also have a type, which means that only certain kinds of values can be stored in a variable.[14]

Similar to variables, data structures are memory locations that hold a collection of data, stored in a way that implements logical relationships among the data. For example, an array is a data structure whose elements can be identified by an index. Different programming languages permit different operations on data structures, for example the appending to and deleting from an array.

Classes can interact in two ways. First, in the traditional *monolithic* software architecture, widely used for enterprise software like the one developed by SAP, for operating systems like Microsoft's Windows, and even in earlier versions of e-commerce platforms like Amazon, individual components cannot be executed independently. In contrast, many modern software projects use a *microservices* software architecture, which is a collection of cohesive, independent processes, interacting via messages. Both architectures result in software where individual pieces of code depend on other pieces, either within a single application or across various microservices. These dependencies form a network which we formalize in the next section.

[13] There are three different types of variables in object-oriented code. First, a *member variable* can take a different value for each object of a class. Second, a *class variable* has the same value for all objects of a class. And third, a *global variable* has the same value for all objects (i.e. irrespective of the object's class) in the program.

[14] For example, `C` is a statically typed programming language, i.e. the `C` compiler checks during the compilation of the source code that variables are only passed values for storage that the variable type permits. Similarly, `Rust` is a statically typed programming language that, unlike `C`, is also object oriented. `Python`, on the other hand, is dynamically typed, and checks the validity of value assignments to variables only during run-time.

2.2 Dependency Graphs

Most open source software is organized by package managers like Cargo (for the Rust programming language) that standardize and automate software distribution. To make the installation of complex software projects easier for users, package managers also keep track of a package's dependencies and allow users to install these alongside the package they are installing. The existence of package managers and the level of automation they provide is necessary because modern software is frequently updated and different versions of the same package are not always compatible. Packages are logically grouped into repositories, controlled and managed by a so-called maintainer.[15]

Our unit of analysis is the network of dependencies among repositories. Specifically, we describe a software system consisting of $\mathcal{N} = \{1, \ldots, N\}$ repositories with $N = |\mathcal{N}| \geq 3$, each of which is managed by a different maintainer who decides whether to implement code herself or to re-use existing code from other repositories. This (re-)use of existing code gives rise to linkages between repositories. The resulting network $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is called the *dependency graph* of the software system. Denote $g_{ij} = 1$ if $(i, j) \subseteq \mathcal{E}$ is an existing link from repo i to repo j, indicating that i depends on j. Denote as $g = \{g_{ij}\}$ the resulting $N \times N$ adjacency matrix of the repository dependency graph $\mathcal{G}$. We exclude self-loops by defining $g_{ii} := 0$, because, technically, a package cannot depend on itself.

There are two reasons for constructing the dependency graph on the repository rather than the package level. First, information about the popularity of software packages is only created on the repository level, not the package level.[16] And second, repositories provide a logical grouping of related software packages. Alternatively, we could construct the dependency graph on the level of individual software packages. However, the breadth of code and functionality between packages can be much smaller than the breadth of code and functionality within a package. Yet another alternative is to study *call graphs* arising from procedures within the same software

[15] Maintainers can also be organizations. Some repositories are also controlled by a group of maintainers, but for our purposes, this is immaterial.

[16] For example, users can "star" a repository on GitHub if they find it particularly useful.

Table 1: Network statistics for the dependency graph of the Cargo Rust ecosystem. #Nodes and #Edges is the number of nodes and edges for the total network and for the largest weakly connected component, respectively. #Components is the number of connected components. LCC indicates the largest (weakly) connected components.

#Nodes	#Edges	#Components	#Nodes (LCC)	#Edges (LCC)
35,473	696,790	91	35,274	696,679

package (see, for example, Grove et al. (1997)). But these are state-dependent, i.e., dependencies arise during runtime and depending on how the package is executed and interacted with.

Conveniently, the website libraries.io provides information collected from various open source software package mangers, including Cargo.[17] The data provided includes information on projects—essentially the same as a Cargo package—as well as repositories and for each repository a list of all dependencies, defined on the project level. Each project belongs to exactly one repository, which allows us to construct dependencies among repositories from the provided data.

On the repository level, this data includes information about the Size of the repository in kB, its Popularity, measured as the number of stars it has on the website hosting the repository, and as an alternative measure of popularity the number of contributors, i.e. the number of individual coders who have added code to a repository, raised or answered an issue, or reviewed code submitted by others.[18]

Table 1 gives a high-level overview of the repo-based dependency graph constructed in this way. While there are 91 weakly connected components, the largest weakly connected component covers almost all (over 99%) nodes and edges. This means that we will still cover all relevant dynamics even if we focus on the largest weakly connected component of the dependency graph only.

[17] They obtain this data by scraping publicly available repositories hosted on GitHub, GitLab, or Bitbucket.

[18] We restrict ourselves to repositories that have a size larger than zero and that have no more than 50,0000 stars where the latter restriction is included to eliminate 154 repositories where the number of stars seems to have been artificially and unreasonably increased using e.g. a script.

Table 2: Distribution of in- and out-degree and eigenvector centrality for the largest weakly connected component of the repo-based dependency graph of the Cargo Rust ecosystem with $N = 35,274$ nodes.

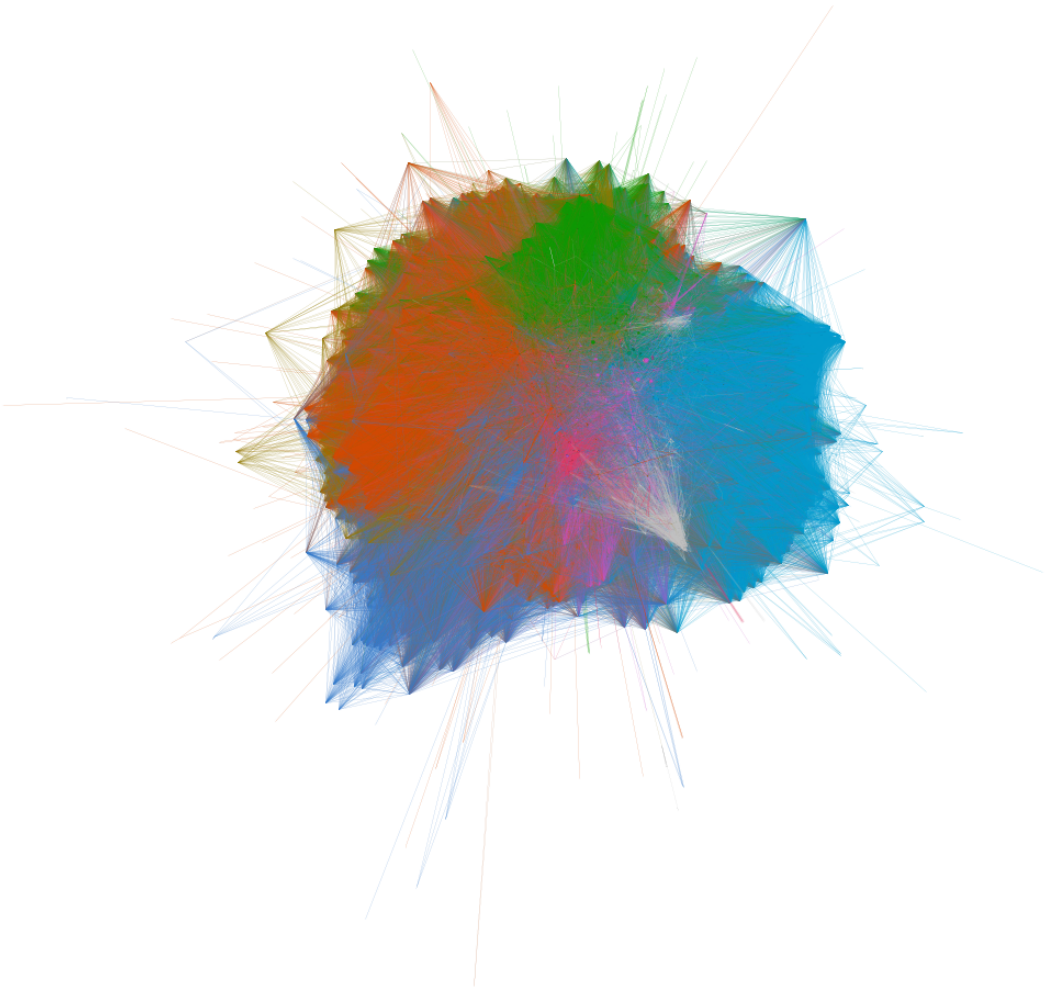
	In-Degree	Out-Degree	Eigenvector Centrality
Mean	19.75	19.75	0.0013
Std. Dev.	267.57	35.91	0.018
Minimum	0	0	0.0
p5	0	0	0.0
Median	0	5	0.0
P95	8	100	0.0005
Maximum	14,585	316	1.0

Furthermore, there is a lot of heterogeneity in how interconnected repositories are, as Table 2 shows. The standard deviation of the out-degree, i.e. how many dependencies a given repository uses, is almost twice as large as the mean, meaning that there is substantial heterogeneity, with the median being that a repository uses 5 dependencies, while the maximum is 316 dependencies. This heterogeneity is even more stark for the in-degree, i.e. how often a given repository is a dependency for another repository. The standard deviation is more than ten times as large as the mean and the median is 0, while the maximum is 14,585.

In Table 3 we show the 10 most depended-on packages (i.e. with the highest in-degree). For each of them it is quite intuitive why the repository is used so often. The most-depended on repository, for example, is libc, which allows Rust code to interoperate with C code, which is particularly important for high-performance and security relevant computations. The second-most depended repository is rand, which is used to generate random numbers, while log is used for run-time logging, a crucial task during code development.

The largest weakly connected component of the repo-based dependency graph is shown in Figure 1, where we apply the community detection algorithm of Blondel et al. (2008) to color-code packages in the same community. The algorithm identifies 32 distinct communities where community members are relatively highly connected to other community members and less connected to nodes outside the community.

Figure 1: Dependency graph of the Cargo Rust ecosystem. Each node is a repository in the libraries.io dataset, and each edge is a dependency between two package versions, each of which belongs to a different repository. Color indicates different modularity classes identified using Blondel et al. (2008).



2.3 Covariate Data

In addition to the software dependency data, we also obtain additional data from the from `libraries.io` website for each package (e.g. `https://libraries.io/cargo/libc`), which we show for the ten most-depended on repositories in Table 3. We use four types of covari-

Table 3: List of the ten most depended-on repositories for the Rust Cargo ecosystem. Popularity is measured as the number of stars a repository received on the website hosting it. Size is measured in Kilobytes of all code in the repository. #Contrib. is the number of coders who have contributed to a repository.

			Covariates		
Repo name	Eigenvector Centrality	In-Degree	Popularity	Size	#Contrib.
rust-lang/libc	1	14,585	674	14,323	338
rust-random/rand	0.764	11,244	546	5,224	546
rust-lang/log	0.723	10,628	674	1,106	68
retep998/winapi-rs	0.679	10,117	705	19,643	128
bitflags/bitflags	0.686	10,105	248	782	47
serde-rs/serde	0.738	10,072	2,764	6,468	132
r[...]y/lazy-static.rs	0.692	10,045	709	920	41
BurntSushi/byteorder	0.531	7,831	464	250	38
serde-rs/json	0.498	7,272	1,293	1,654	69
rust-lang/regex	0.490	7,261	1,291	13,175	116

ates, summarized in a vector of observable attributes x_i. First, we use a repository's Size of all code in the repository, measured in Kilobytes. Second, Popularity is measured as the number of stars–an expression of how many people like a repository or find it useful–a repository received on the website hosting it, e.g. github. And, third, we use an alternative popularity measure, Num.Contributors, defined as the number of coders who have contributed to a repository. A contribution is not only writing code, but contributing to the code generation process more broadly. Contributiions are defined as code reviews, code commits, creating issues related to a repository, and pull requests, i.e. proposing changes to a repository.

To facilitate the estimation of our model, we create a categorical variable for each of our covariates using quartiles of the distribution. The reason to discretize the variables is mostly computational, as our algorithm is based on stochastic blockmodels with discrete types (Bickel et al., 2013; Vu et al., 2013). If the covariates are discrete, then the whole machinery for estimation of blockmodels can be adapted to estimate the unobserved heterogeneity (Vu et al., 2013; Babkin et al., 2020; Dahbura et al., 2021); while with continuous variables the computational costs of

Table 4: Distribution of covariates for the $N = 35,274$ repositories in the largest connected component of our sample. Size is measured in Kilobytes of all code in the repository. Popularity is measured as the number of stars a repository received on the website hosting it. #Contributors is the number of coders who have contributed to a repository.

	Mean	Std.dev	Min	p5	Median	p95	Max
Size [kB]	3,903.91	28,152.89	1	3	79	9,672	1,025,320
Popularity	30.85	392.90	0	0	0	63	41,652
#Contributors	3.49	17.86	0	0	1	10	1,247

estimation become prohibitive for such large networks.[19]

3 Model

We model the decisions of a team of coders contributing to a software repository consisting of closely related software packages. Coders decide whether to use functionality from other packages or develop it themselves. In addition to the four observables introduced above, also assume that there are K discrete types, unobservable to the researchers but observable to other coders, $\boldsymbol{z}_i = (z_{i1}, ..., z_{iK})$. The type of a software package can be thought of as the basic function of the package and examples include operating system components, graphical user interfaces, text processing software, compilers, and so on. It can also capture unobservable quality of the code and/or developers. A package of type k is denoted by $z_{ik} = 1$ and $z_{i\ell} = 0$ for all $\ell \neq k$. We use notation $\boldsymbol{x}$ and $\boldsymbol{z}$ to denote the matrix of observable and unobservable characteristics for all the packages.

The utility function of coder i from network $\boldsymbol{g}$, observables $\boldsymbol{x}$, unobservables $\boldsymbol{z}$ and parameters $\boldsymbol{\theta} = (\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma})$ is:

$$U_i\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) = \sum_{j=1}^{N} g_{ij} u_{ij}(\alpha, \beta) + \sum_{j=1}^{N} \sum_{r \neq i,j}^{N} g_{ij} g_{jr} w_{ijr}(\gamma) + \sum_{j=1}^{N} \sum_{r \neq i,j}^{N} g_{ij} g_{ri} v_{ijr}(\gamma), \tag{1}$$

[19] We provide more details on this in the model and the appendix. More technical details are provided in Babkin et al. (2020) and Dahbura et al. (2021).

The payoff $u_{ij}(\boldsymbol{\alpha},\boldsymbol{\beta}) := u\left(\boldsymbol{x}_i,\boldsymbol{x}_j,\boldsymbol{z}_i,\boldsymbol{z}_j;\boldsymbol{\alpha},\boldsymbol{\beta}\right)$ is the direct utility of linking to package j. It is a function of observables $(\boldsymbol{x}_i,\boldsymbol{x}_j)$, unobservables $(\boldsymbol{z}_i,\boldsymbol{z}_j)$ and parameters $(\boldsymbol{\alpha},\boldsymbol{\beta})$. This is the benefit of creating the dependencies to package j, net of costs—a coder will have to audit the code of the linked package and determine it's quality—of maintaining the link. The cost also includes modification and adaptation of the code that the coder has to do to be able to use the functions and methods available in package j. Furthermore, using another coder's code also increases the risk that the code contains an undetected bug (assuming that coders care more about the code of their own package than the code of their dependencies), which can be captured as a cost. And lastly, re-using code, while common practice in modern software development, means coders require fewer skills, which might negatively impact their future productivity and can thus be interpreted as another cost.

We assume that the utility function $u_{ij}(\boldsymbol{\alpha},\boldsymbol{\beta})$ is parameterized as follows

$$u_{ij}(\boldsymbol{\alpha},\boldsymbol{\beta}) = \begin{cases} \alpha_w + \sum_{p=1}^{P} \beta_{wp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} & \text{if } z_i = z_j \\ \alpha_b + \sum_{p=1}^{P} \beta_{bp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} & \text{otherwise,} \end{cases} \tag{2}$$

where the intercept α is interpreted as the cost of of forming the dependencies, and it is a function of unobservables only; and $\mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\}$ are indicators functions equal to one when the p-th covariates are the same for i and j. In this parsimonious specification we allow the net benefits of link to vary with both observed and unobserved heterogeneity.

The second term and third terms of the utility function (1) are externalities generated while linking to package j. Indeed, package j may be linked to other packages r as well. This means that the coder needs to check the quality and features of the code in these packages, to make sure they are free of bugs and compatible with the coder's own code. On the other hand, any update to package r may compromise compatibility to package j and i, so this also includes costs of maintaining. In short, $w_{ijr}(\gamma)$ measures the net benefits of this externality when coders of i form a dependency to library j.

Furthermore, if package i creates a dependencies to library j it may compromise compatibility

with other packages r that have a dependencies to i. This is accounted for by the third term $v_{ijr}(\gamma)$ in utility function (1).

We assume that the second and third term in the utility take the form

$$w_{ijr}(\gamma) = v_{ijr}(\gamma) = \begin{cases} \gamma & \text{if } z_i = z_j = z_r \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

This specification assumes that the externalities are local and have the same functional form. The reason for this normalization is that it guarantees the existence of a potential function that characterizes the equilibrium. Furthermore, this assumptions facilitates identification for parameter γ, through variation within types (Mele, 2017, 2022; Schweinberger and Handcock, 2015; Schweinberger and Stewart, 2020; Babkin et al., 2020).

To summarize, the coders need to make decisions on whether to link another package or not. When making the decision, they take into account some externalities from this network of dependencies, but not all the possible externalities.

We assume a dynamic process of package development. Time is discrete and in each time period t only one coder is making decisions. At time t, a randomly chosen package i needs some code update. Package j is proposed for the update. If package j is not already linked to package i, we have $g_{ij,t-1} = 0$. If the coder decides to write the code in-house there is no update and $g_{ij,t} = 0$. On the other hand, if the coder decides to link to package j, the network is updated and $g_{ij,t} = 1$. If the dependency already exists (i.e. $g_{ij,t-1} = 1$), then the coder decision is whether to keep the dependency ($g_{ij,t} = 1$) or unlink package j and write the code in-house instead ($g_{ij,t} = 0$). Here, we make the simplifying assumption that coders cannot substitute a dependency for another dependency at the same time. Formally, with probability $\rho_{ij} > 0$ package i coders propose an update that links package j. The coder decides whether to create this dependency or write the code in-house. Before linking the coder receives a random shock to payoffs $\varepsilon_{ij0}, \varepsilon_{ij1}$, which is i.i.d. among packages and time periods. This shock captures that unexpected things can happen both when developing code in house and when linking to an existing package.

So the coders of i form a dependency link to package j if:

$$U_i\left(\boldsymbol{g}', \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) + \varepsilon_{ij1} \geq U_i\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}\right) + \varepsilon_{ij0} \tag{4}$$

where $\boldsymbol{g}'$ is network $\boldsymbol{g}$ with the addition of link g_{ij}.

Throughout the paper we maintain the following assumptions (Mele, 2017):

1. The probability that coders of i propose an update that links to package j is strictly positive, $\rho_{ij} > 0$ for all pairs i, j. This guarantees that any dependencies can be considered.

2. The random shock to payoffs, ε_{ij}, follows a logistic distribution. This is a standard assumption in many random utility models for discrete choice.

As shown in Mele (2017) and Mele (2022), after conditioning on the unobservable types $\boldsymbol{z}$, the sequence of networks generated by this process is a Markov Chain and it converges to a unique stationary distribution that can be expressed in closed-form as a discrete exponential family with normalizing constant. In our model, the stationary distribution is

$$\pi(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}) \quad := \quad \prod_{k=1}^{K} \frac{e^{Q_{kk}}}{c_{kk}} \left[\prod_{l \neq k}^{K} \prod_{i=1}^{N_k} \prod_{j=1}^{N_l} \frac{e^{u_{ij}(\alpha_b, \beta_b)}}{1 + e^{u_{ij}(\alpha_b, \beta_b)}} \right], \tag{5}$$

where the potential function Q_{kk} can be written as:

$$\begin{aligned} Q_{kk}(\boldsymbol{g}_{kk}) \quad &:= \quad Q\left(\boldsymbol{g}_{kk}, \boldsymbol{x}, \boldsymbol{z}; \alpha_w, \boldsymbol{\beta}_w\right) \\ &= \quad \sum_{i=1}^{N} \sum_{j \neq i}^{N} g_{ij} z_{ik} z_{jk} \left(\alpha_w + \sum_{p=1}^{P} \beta_{wp} \mathbf{1}\{\boldsymbol{x}_{ip} = \boldsymbol{x}_{jp}\} \right) + \gamma \sum_{i=1}^{N} \sum_{j \neq i}^{N} \sum_{r \neq i,j}^{N} g_{ij} g_{jr} z_{ik} z_{jk} z_{rk}, \end{aligned} \tag{6}$$

and $\boldsymbol{g}_{kk}$ is the network among nodes of type k; the normalizing constant is:

$$c_{kk} := \sum_{\omega_{kk} \in \mathcal{G}_{kk}} e^{Q_{kk}(\omega_{kk})}. \tag{7}$$

Here, $\omega_{kk} \in \mathcal{G}_{kk}$ is one network in the set of all possible networks among nodes of type k.[20]

[20] As noted in Mele (2017), the constant c_{kk} is a sum over all $2^{N_k(N_k-1)/2}$ possible network configurations for the

The stationary distribution π represents the long-run distribution of the network, after conditioning on the unobservable types z. The second product in (5) represents the likelihood of links between packages of different unobservable types, while the first part is the likelihood of links among libraries of the same type. This simple decomposition is possible because the externalities have been normalized to be local. So the model converges to K independent exponential random graphs for links of the same type, and implies conditionally independent links for between-types connections.

Because of this simple characterization of the stationary equilibrium, the incentives of the coders are summarized by a potential function whose maxima are pairwise stable networks. Because of the externalities, in general, the equilibrium networks will be inefficient, as they will not maximize the sum of utilities of the coders.[21]

4 Estimation and Empirical Results

4.1 Estimation

Our novel method for estimation of a directed network formation model with observable and unobservable types is a crucial contribution of this paper. Although scalable estimation methods are available for simpler models, like stochastic blockmodels (Vu et al., 2013), conditionally independent links models (Mele et al., 2023), or models for undirected networks (Dahbura et al., 2021; Babkin et al., 2020; Schweinberger and Stewart, 2020), there is no scalable method for directed networks readily available.

To understand the computational problem involved in this estimation, we write the likelihood

nodes of type k. This makes the computation of the constant impractical or infeasible in large networks.

[21] This is evident when we compute the sum of utilities. Indeed, the welfare function will include all the externalities and, therefore, will be, in general, different from the potential, where the externalities are multiplied by 1/2. See also Mele (2017) for more details.

of our model as

$$\mathscr{L}(\boldsymbol{g},\boldsymbol{x};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}) = \sum_{z\in\mathscr{Z}} L\big(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\big) = \sum_{z\in\mathscr{Z}} p_{\boldsymbol{\eta}}(z)\,\pi(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma). \tag{8}$$

where each node's type distribution $p_{\boldsymbol{\eta}}(z)$ is multinomial, and types are i.i.d.

$$Z_i|\eta_1,...,\eta_K \overset{\text{iid}}{\sim} \text{Multinomial}\big(1;\eta_1,...\eta_K\big) \text{ for } i = 1,...,N$$

Maximizing (8) involves a sum over all possible type allocations of each node. Furthermore, the stationary distribution $\pi(\boldsymbol{g},\boldsymbol{x},z;\boldsymbol{\alpha},\boldsymbol{\beta},\gamma)$ (the conditional likelihood) is a function of an an intractable normalizing constant. Therefore direct computation of the objective function is infeasible.

Our strategy is to separate the estimation of unobserved types from the structural utility parameters according to a two-step algorithm. First, we estimate the type of each node and, second, we estimate the structural parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and γ, conditional on the type of each node (Babkin et al., 2020).

In the first step, we extend variational approximations with maximization minorization updates to directed dependencies for approximating the intractable likelihood and estimate the posterior distribution of types (Vu et al., 2013). We then assign the types to each node according to the highest posterior probability. This step exploits the fact that our model corresponds to a stochastic blockmodel with covariates when $\gamma = 0$, i.e. when there are no link externalities. Although the variational mean-field approximation algorithm leads to a sequential algorithm, we augment a lower bound of the likelihood by maximizing a minorizing function, which is quadratic. Thereby, we have to solve a quadratic maximization problem whose terms are calculated via fast sparse matrix operations and whose solution is found via quadratic problem solvers with box constraints (Stefanov, 2004). Through this approximation, we can perform the estimation of unobserved types at scale while preserving the complex correlation of dependency links in our data (more details are provided in Appendix A).

In the second step, we estimate the remaining structural parameters using a pseudolikelihood

estimator that maximizes the conditional choice probabilities of links conditional on the estimated types (Schweinberger and Stewart, 2020; Dahbura et al., 2021). This is a computationally feasible estimator that scales well with the size of the network. We provide the theoretical and computational details on the scalable estimation procedure in Appendix A.

4.2 Results

The results of our estimation are reported in Table 5. The first 3 columns show estimates of the structural parameters for links of the same type, while the remaining 3 columns are the estimates for links of different types.

Following the 32 identified library types in Figure 1, we estimate a model with $K = 32$ unobserved types and initialize the type allocation for our algorithm using InfoMap (Rosvall et al., 2009).[22]

We estimate a positive externality (γ), suggesting that a simple stochastic blockmodel is unable to account for the dependence of the links in our data. This result aligns with our strategic model, suggesting that links are highly interdependent. Essentially, when coders form connections, they (partially) consider the indirect impacts of their actions. However, this behavior leads to a network architecture that might not be as efficient as one where a central planner aims to maximize the overall utility of each participant. Indeed, the externality encourages coders to establish more connections than would be ideal from an aggregate utility-maximization perspective. This also has implications for contagion, as we explain in the next section.

The parameter α governs the density of the network and is negative both between and within types. This implies that—other things being equal— in equilibrium sub-networks of libraries of the same type tend to be more sparse than between-types. One interpretation is that α measures costs of forming links. By contrast, the parameter β_1 is positive both within and across

[22] As suggested in the literature on variational approximation (Wainwright and Jordan, 2008), we started the algorithm from different initial guesses of the type allocations but only report the results that achieved the highest lower bound.

Table 5: Parameter estimates and standard errors. Estimates obtained by setting $K = 32$ types and running the algorithm for 300 iterations, after initializing with InfoMap.

	Within			Between		
	Est.	Std. Err.	z value	Est.	Std. Err.	z value
Externality (γ)	.264	.002	121.996			
Edges (α)	−12.070	.065	−186.323	−6.602	.002	−3843.154
Size(β_1)	.166	.049	3.411	.432	.003	167.775
Popularity(β_2)	.546	.048	11.326	−.060	.003	−19.365
Num.Forks(β_3)	−1.012	.050	−2.332	−2.266	.004	−507.326
Num.Contributors(β_4)	.063	.047	1.327	−1.12	.004	−314.475
Maturity(β_5)	2.659	.061	43.751	−.448	.003	−141.542

types, indicating that coders are more likely to link to larger packages of the same and of different types.

Interestingly, Popularity has a positive coefficient β_2 within types and a—much smaller—negative coefficient between types. The same is true for Num.Contributors, which is an alternative measure of popularity. Coders are, therefore, likely to link to more popular packages of the same type but less popular packages of other types. One interpretation consistent with these results is that coders use popular features of same-type packages to include in their own code but use infrastructure-style code from different-type packages, which is not necessarily very popular. This interpretation is also consistent with the coefficients β_4 and β_5 for Num.Contributors and Maturity, which are positive within types and negative between-type packages. Lastly, the coefficient β_3 for Num.Forks is negative both within and across projects, but not only significantly for between-type links. This could be a mechanical effect, though: If a project has more forks, there are more similar versions of the same package, meaning that a coder has more possibilities to link to it.

5 Contagious Vulnerabilities

A growing literature explores contagion on networks, including the statistical literature on infectious diseases (e.g., the contagion of HIV and coronaviruses). Such models (see, e.g., Schweinberger et al., 2022; Groendyke et al., 2012; Britton and O'Neill, 2002) usually first use a network formation model to generate a network of contacts among agents and then study how an infectious disease following a stochastic process—like the susceptible-infected-recovered model (Kermack and McKendrick, 1972; Hethcote, 2000)—spreads within this network.

We adapt these ideas to the contagion of vulnerabilities in software dependency graphs. The prevalence of vulnerabilities in software have led to the creation of the Common Vulnerabilities and Exposure (CVE) program and the National Vulnerability Database (NVD) by the National Institute of Standards and Technology, which is an agency of the United States Department of Commerce (NIST, 2024). A software package depending on a vulnerable package is potentially vulnerable itself. To see how, consider, for example, the Equifax data breach (Federal Trade Commission, 2024) in 2017, caused by a vulnerability in the popular open source web application framework Apache Struts 2.[23] Equifax did not itself develop or maintain Apache Struts 2, but instead used it as part of its own code base. The vulnerability allowed attackers to execute malicious code providing access to the server hosting the web application developed by Equifax using Apache Struts 2 and costing the company more than $1.5 Billion to date.[24]

Package managers are part of a large suite of tools that help systems administrators to keep their systems up to date and patch any known vulnerabilities. It takes time for vulnerable systems to be updated (Edgescan, 2022),[25] not all vulnerabilities are publicly disclosed (Arora et al., 2008),

[23] Apache Struts 2 did not properly validate the Content-Type header of incoming http requests when processing file uploads, which allowed attackers to create http requests that include malicious code. Due to the lack of content validation, this allowed attackers to execute arbitrary code on the web server, ultimately granting them full control of the server. For details see the official NIST announcement on CVE-2017-5638 (https://nvd.nist.gov/vuln/detail/CVE-2017-5638).

[24] See https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473. Article accessed 27 January 2024.

[25] It takes organizations between 146 and 292 days to patch cyber security vulnerabilities (Source: Statista. Accessed on 2024-01-31).

and it could even be optimal to release vulnerable software packages (Arora et al., 2006). This leaves time for attackers to exploit vulnerabilities in downstream software packages and thereby attack the systems hosting code that depends on the vulnerable packages.

Since it is not ex-ante clear which part of the dependency code is vulnerable, we assume that package i uses vulnerable code in dependency j with probability $\rho_{ij} \in [0,1]$ and, for simplicity, we take $\rho_{ij} = 1$. This avoids relying on statistical measures of the extent of contagion. The resulting contagion process is very similar to the traditional susceptible-infected-recovered (SIR) model in epidemiology (Kermack and McKendrick, 1972; Hethcote, 2000) applied to contact networks.[26]

How contagious a vulnerability is can be measured in this setting by the number of downstream repositories it affects. While github and other repository hosting services make it easy to see the number of packages directly depending on a given repository, direct dependencies are only imperfect proxies for the total number of vulnerable repositories two or three steps away. Formally, the d-neighborhood $\mathcal{N}_i^d(g)$ of node i in network $\mathcal{G}$ is defined via:

$$\mathcal{N}_i^1(g) = N_i(g) \quad , \quad \text{and } \mathcal{N}_i^k(g) = \mathcal{N}_i^{k-1}(g) \cup \left(\bigcup_{j \in \mathcal{N}_i^{k-1}(g)} N_j(g) \right)$$

where g is the adjacency matrix of $\mathcal{G}$ and $N_i(g) = \{j \in \mathcal{G} | g_{ij} = 1\}$ is the (in)-neighborhood of node i. Using this, we define a repository's *k-step systemicness* as Syst.$k_i(g) \equiv \mathcal{N}_i^k(g)$. We usually omit the network reference when it is clear from context which network we study and throughout this section we use the dependency graph of the Rust Cargo ecosystem.

Table 6 shows the distribution of the k-step systemicness for the full sample of all repositories (Panel A) and for only the 1% of repositories with the highest in-degree. The most immediate result is that the distribution of k-step systemicness is highly skewed. While, on average, a vulnerable repository renders 52 repositories two steps away vulnerable, the most systemic repository affects 26,631 other repositories two steps away. In other words, a vulnerability in the most systemic repository would affect 75% of all existing repositories within two steps.

[26] See, among others, Rocha et al. (2023).

Table 6: Distribution of systemicness measures for all repositories and repositories in the top 1% of In-Degree. Syst.k is computed as the number of downstream repositories that are potentially vulnerable if a given repository contained a vulnerability. $N = 35,274$ for the All repositories sample.

PANEL A: All repositories

	Mean	SD	Median	p5	p95	p99	Max
Syst.2	52.08	610.48	1	1	13	916	26,631
Syst.3	97.12	1,079.22	1	1	14	1,504	30,477
Syst.4	152.44	1,596.91	1	1	14	2,303	31,751
Syst.5	215.96	2,072.12	1	1	14	5,187	31,951

PANEL B: Top 1% In-Degree repositories

	Mean	SD	Median	p5	p95	p99	Max
Syst.2	505.22	1,860.83	13	3	2,906	9,724	26,631
Syst.3	947.24	3,269.01	13	3	6,574	17,926	30,477
Syst.4	1,491.98	4,814.18	13	3	12,920	25,282	31,751
Syst.5	2,117.42	6,191.37	13	3	18,903	29,122	31,951

This considerable heterogeneity across packages can also be seen from the standard deviation, which is almost twelve times as large as the mean. As we take into account how a vulnerable repository affects other repositories further away, the mean increases by a factor of four for $k = 5$, while the maximum reaches 31,951.

Table 7 shows the systemicness of the ten repositories with the highest number of dependent packages (in-degree). Among those, `serde-rs/serde` is the package that has the largest impact two and three steps away, while `rust-lang/libc` is the package that has the largest impact four and five steps away.

Now that we have a measure for how "infectious" a vulnerable software package is, we turn to the question of how to prevent the spread of vulnerabilities, staying with our analogy to epidemiological SIR models. Chatterjee and Zehmakan (2023) compare different vaccination strategies in network-based SIR models. Since it is not feasible to find an optimal vaccination strategy—the optimization problem is NP hard—the authors compare various heuristics based on a node's network position and pathogen parameters. They find that a vaccination strategy

Table 7: The ten repositories with the highest number of dependent packages. Syst.k is computed as the number of downstream repositories that are potentially vulnerable if a given repository contained a vulnerability. In-degree corresponds to Syst.1.

Repo name	In-Degree	Syst.2	Syst.3	Syst.4	Syst.5
rust-lang/libc	14,586	23,298	30,477	31,751	31,951
rust-random/rand	11,245	17,658	23,310	27,071	28,766
rust-lang/log	10,629	19,048	22,988	26,673	28,289
retep998/winapi-rs	10,118	12,426	18,946	25,652	27,730
bitflags/bitflags	10,106	15,673	21,106	25,933	28,002
serde-rs/serde	10,073	26,631	30,050	30,440	30,501
r[...]y/lazy-static.rs	10,046	17,550	25,337	28,060	30,359
BurntSushi/byteorder	7,832	13,888	21,815	26,228	28,540
serde-rs/json	7,273	17,196	26,976	29,673	30,414
rust-lang/regex	7,262	14,786	22,525	29,203	30,388

based on a node's betweenness centrality and a heuristic they call *expected fatality* is most effective in preventing fatalities.

The process of how a vulnerability in one software package affects other packages, as we discuss it above, is a special case of a SIR process. First, by setting $\rho_{ij} = 1 \forall i, j \in N$, we assume that all upstream neighbors of a vulnerable node (repository) are also vulnerable. And second, nodes cannot "die" in our setup, so they are not removed from the population. They can, in principle, recover from being exposed to a vulnerable package if the vulnerability is patched and the dependency is updated, but this takes time, as discussed above. Consequently, software systems are vulnerable for a period of time once a vulnerability is discovered. We are interested in contagion processes occurring during this time.

Consequently, we study the effectiveness of a prevention strategy based on securing the most important nodes against vulnerabilities (e.g. by through government funding or regulation). Chatterjee and Zehmakan (2023) propose to use an equally-weighted combination of a node's betweenness and expected fatality $ef(i)$ of node i, which in our case is computed as:

$$ef(i) = \sum_{j \in N(i)} \frac{1}{|N(j)|}.$$

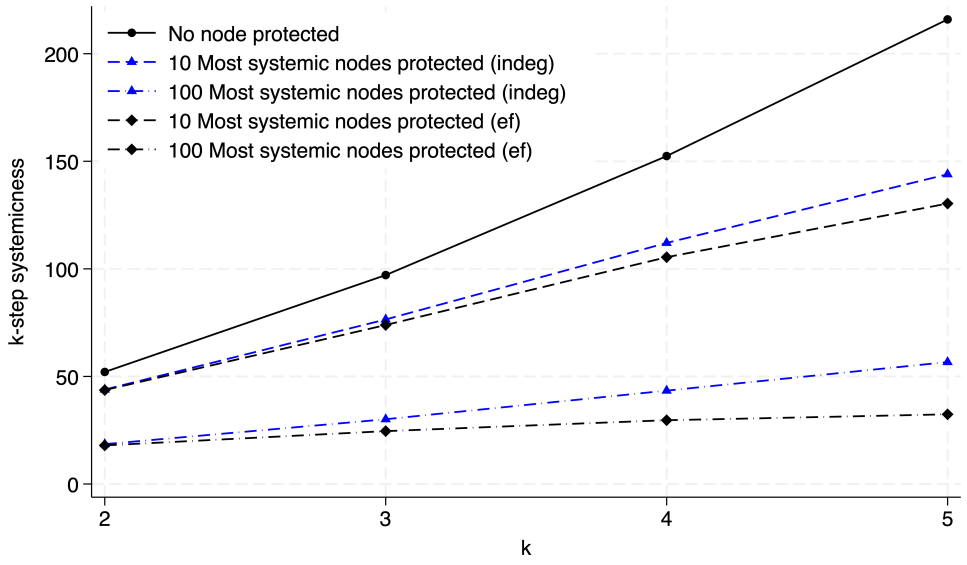
Next, we apply this method to the Rust Cargo ecosystem. We normalize betweenness centrality and expected fatality by dividing through the maximum of each so that they are more directly comparable. This gives us a value of *expected systemicness* for each repository i. We measure how effective this measure can be used to prevent vulnerability contagion by taking the ten and one hundred repositories with the largest expected systemicness and assuming that so much debugging, exception handling and error preventing effort is exerted that these repositories are invulnerable even if they depend on a vulnerable repository. We then compute the average k-step systemicness again and show it in Figure 2. Protecting l nodes is effective if it reduces the k-step systemicness by more than l, i.e. if there are amplification effects from the protection.

We can take away five things from Figure 2. First, using the expected systemicness as an ex-ante measure of importance is effective for $k \geq 3$.[27] Second, the effect of protecting only ten repositories can be significant. For $k = 5$, protecting the ten repositories with the largest expected systemicness reduces the average systemicness (the extent of vulnerability contagion) from 215.6 to 130.4, i.e. by almost 40%.[28] Third, the effect is even more pronounced when protecting 100 repositories, but with diminishing returns. The average systemicness for $k = 5$ is reduced to 32.4, which is an 85% reduction. However, protecting 100 repositories reduces the amplification effect only marginally more than protecting 10 repositories. Fourth, we compare the expected systemicness with an alternative measure of a node's importance to get a sense of how effective the refined measure is relative to other measures. Specifically, we use a node's in-degree as a measure of expected systemicness and show this in blue in Figures 2 and A2. For $k = 5$, the in-degree reduces systemicness by about 10% less when the 10 nodes with the largest number of dependent repositories is protected relative to when the ten repositories with the highest expected systemicness are protected. Lastly, as we can see from Figure A2, where we restrict the sample to the 1% of repositories with the greatest number of dependent repositories, that the results are qualitatively similar and quantitatively even more pronounced.

[27] We show the k-step systemicness for the 1% of repositories with the largest number of dependents in Figure A2 in Appendix A. For those repositories, the expected systemicness is efficient for any k.

[28] Or 35% when only considering the amplification effect of the protection, i.e. without counting the 10 initially protected repositories.

Figure 2: Comparison of k-step systemicness for all repositories in the Rust Cargo ecosystem. The k-step systemicness measures how many repositories the average vulnerable repository renders vulnerable k steps away. The solid black line shows the average k-step systemicness when no node is protected, while the dashed and dot-dashed black line shows the average k-step systemicness when the 10 and 100 most systemic repositories are perfectly protected against vulnerabilities, respectively. The dashed and dot-dashed blue line shows the average k-step systemicness when systemicness for protected nodes are measured as the number of dependents a repository has (in-degree).



6 Conclusion

The network of dependencies among software packages is an interesting laboratory to study network formation and externalities. Indeed, the creation of modern software gains efficiency by re-using existing libraries; on the other hand, dependencies expose new software packages to bugs and vulnerabilities from other libraries. This feature of the complex network of dependencies motivates the interest in understanding the incentives and equilibrium mechanisms driving the formation of such networks.

In this paper, we estimate a directed network formation model to undertake a structural analysis of the motives, costs, benefits and externalities that a maintainer faces when developing a new software package. The empirical model allows us to disentangle observable from unobservable characteristics that affect the decisions to form dependencies to other libraries.

We find evidence that coders create positive externalities for other coders when creating a link. This raises more questions about the formation of dependency graphs. We study how vulnerable the observed network is to vulnerability contagion and show that ensuring that even a relatively small number of packages is vulnerability free effectively curbs the extent of contagion. As a next step, we could study how changing the parameters of the estimated network formation model affects vulnerability contagion. The presence of an externality implies that maintainers may be inefficiently creating dependencies, thus increasing the density of the network and the probability of contagion. The extent of this risk and the correlated damage to the system is of paramount importance and we plan to explore it in future versions. Second, while we have focused on a stationary realization of the network, there are important dynamic considerations in the creation of these dependencies. While the modeling of forward-looking maintainers may be useful to develop intuition about intertemporal strategic incentives and motives, we leave this development to future work. Finally, we have focused on a single language, but the analysis can be extended to other languages as well, such as `Java`, `C++` and others.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1996), *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA.

Arora, A., Caulkins, J. P. and Telang, R. (2006), 'Sell first, fix later: Impact of patching on software quality', *Management Science* **52**(3), 465–471.

Arora, A., Telang, R. and Xu, H. (2008), 'Optimal policy for software vulnerability disclosure', *Management Science* **54**(4), 642–656.

Babkin, S., Stewart, J. R., Long, X. and Schweinberger, M. (2020), 'Large-scale estimation of random graph models with local dependence', *Computational Statistics & Data Analysis* **152**, 1–19.

Barros-Justo, J. L., Pinciroli, F., Matalonga, S. and Martínez-Araujo, N. (2018), 'What software reuse benefits have been transferred to the industry? a systematic mapping study', *Information and Software Technology* **103**, 1–21.

Bickel, P., Choi, D., Chang, X. and Zhang, H. (2013), 'Asymptotic normality of maximum likelihood and its variational approximation for stochastic blockmodels', *The Annals of Statistics* **41**(4), 1922 – 1943.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R. and Lefebvre, E. (2008), 'Fast unfolding of communities in large networks', *Journal of Statistical Mechanics: Theory and Experiment* **P1008**.

Blume, L., Easley, D., Kleinberg, J., Kleinberg, R. and Tardo's, E. (2013), 'Network formation in the presence of contagious risk', *ACM Transactions on Economics and Computation* **1**(2), 6:1–6:20.

Bonhomme, S., Lamadon, T. and Manresa, E. (2019), 'A distributional framework for matched employer employee data', *Econometrica* **87**(3), 699–739.

Boucher, V. and Mourifie, I. (2017), 'My friend far far away: A random field approach to exponential random graph models', *Econometrics Journal* **20**(3), S14–S46.

Britton, T. and O'Neill, P. D. (2002), 'Statistical inference for stochastic epidemics in populations with network structure', *Scandinavian Journal of Statistics* **29**, 375–390.

Chandrasekhar, A. G. (2016), Econometrics of network formation, *in* yann bramoulle, andrea galeotti and brian rogers, eds, 'Oxford handbook on the economics of networks.', Oxford Univerisity Press.

Chatterjee, S. and Zehmakan, A. N. (2023), Effective vaccination strategies in network-based sir model, mimeo.

Currarini, S., Jackson, M. O. and Pin, P. (2010), 'Identifying the roles of race-based choice and chance in high school friendship network formation', *the Proceedings of the National Academy of Sciences* **107**(11), 4857ï¿½4861.

Dahbura, J. N. M., Komatsu, S., Nishida, T. and Mele, A. (2021), 'A structural model of business cards exchange networks'.

Decan, A., Mens, T. and Grosjean, P. (2019), 'An empirical comparison of dependency network evolution in seven software packaging ecosystems', *Empirical Software Engineering* **24**, 381–416.

DePaula, A. (2017), Econometrics of network models, *in* B. Honore, A. Pakes, M. Piazzesi and L. Samuelson, eds, 'Advances in Economics and Econometrics: Eleventh World Congress', Cambridge University Press.

DePaula, A., Richards-Shubik, S. and Tamer, E. (2018), 'Identifying preferences in networks with bounded degree', *Econometrica* **86**(1), 263–288.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M. and Halderman, J. A. (2014), The matter of heartbleed, *in* 'Proceedings of the 2014 Conference on Internet Measurement Conference', IMC '14, Association for Computing Machinery, New York, NY, USA, pp. 475–488.

Edgescan (2022), 2022 vulnerability statistics report, mimeo. Accessed: 2024-01-31.
URL: *https://www.edgescan.com/resources/vulnerability-stats-report/*

Federal Trade Commission (2024), 'Equifax data breach settlement', Federal Trade Commission - Enforcement.
URL: *https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement*

Goldfarb, A. and Tucker, C. (2019), 'Digital economics', *Journal of Economic Literature* **57**(1), 3–43.

Graham, B. (2017), 'An empirical model of network formation: with degree heterogeneity', *Econometrica* **85**(4), 1033–1063.

Graham, B. (2020), Network data, *in* S. Durlauf, L. Hansen, J. Heckman and R. Matzkin, eds, 'Handbook of econometrics 7A', Amsterdam: North-Holland,.

Groendyke, C., Welch, D. and Hunter, D. R. (2012), 'A network-based analysis of the 1861 Hagelloch measles data', *Biometrics* **68**, 755–765.

Grove, D., DeFouw, G., Dean, J. and Chambers, C. (1997), Call graph construction in object-oriented languages, *in* 'Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications', OOPSLA '97, Association for Computing Machinery, New York, NY, USA, p. 108–124.

Hethcote, H. (2000), 'The mathematics of infectious diseases', *SIAM Review* **42**(4), 599–653.

Hopper, G. M. (1952), The education of a computer, *in* 'Proceedings of the Association for Computing Machinery Conference', pp. 243–249.

Jackson, M., ed. (2008), *Social and economic networks*, Princeton.

Jackson, M. O. and Wolinsky, A. (1996), 'A strategic model of social and economic networks', *Journal of Economic Theory* **71**, 44–74.

Kay, A. C. (1993), The early history of smalltalk, Url: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

Kermack, W. O. and McKendrick, A. G. (1972), 'A contribution to the mathematical theory of epidemics', *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* **115**(772), 700–721.

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

Knuth, D. E. (1997), *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Krasner, H. (2018), The cost of poor quality software in the us: A 2018 report, Report.
URL: *https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-repo*
The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

LaBelle, N. and Wallingford, E. (2004), 'Inter-package dependency networks in open-source software'.

Lerner, J. and Tirole, J. (2002), 'Some simple economics of open source', *The Journal of Industrial Economics* **50**(2), 197–234.

Lewis, J. A., Henry, S. M., Kafura, D. G. and Schulman, R. S. (1991), An empirical study of the object-oriented paradigm and software reuse, *in* 'OOSPLA 91', pp. 184–196.

McCarthy, J. (1996), The implementation of LISP, Stanford AI Lab: The History of LISP.
URL: *http://www-formal.stanford.edu/jmc/history/lisp/lisp.html*

Mele, A. (2017), 'A structural model of dense network formation', *Econometrica* **85**(3), 825–850.

Mele, A. (2022), 'A structural model of homophily and clustering in social networks', *Journal of Business & Economic Statistics* **40**(3), 1377–1389.

Mele, A., Hao, L., Cape, J. and Priebe, C. (2023), 'Spectral estimation for large stochastic blockmodels with nodal covariates', *Journal of Business and Economic Statistics* **41**(4), 1364–1376.

Mele, A. and Zhu, L. (forthcoming), 'Approximate variational estimation for a model of network formation', *Review of Economics and Statistics* .

National Institute of Standards and Technology (2002), Software errors cost u.s. economy $59.5 billion annually, Press release.

URL: *https://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm*

NIST (2024), 'The National Vulnerabilities Database', https://nvd.nist.gov/vuln. Accessed: 2024-01-31.

Rocha, J. L., Carvalho, S. and Coimbra, B. (2023), 'Probabilistic procedures for sir and sis epidemic models on erdős-rényi contact networks: Analyzing the relationship between epidemic threshold and topological entropy', *Applied Mathematics* **3**(4).

Rosvall, M., Axelsson, D. and Bergstrom, C. T. (2009), 'The map equation', *The European Physical Journal Special Topics* **178**(1), 13–23.

Schueller, W., Wachs, J., Servedio, V., Thurner, S. and Loreto, V. (2022), 'Curated data on the rust ecosystem: Collaboration networks and library dependencies', *Scientific Data* **9**(703).

Schweinberger, M. (2020), 'Consistent structure estimation of exponential family random graph models with block structure', *Bernoulli* **26**, 1205–1233.

Schweinberger, M., Bomiriya, R. P. and Babkin, S. (2022), 'A semiparametric Bayesian approach to epidemics, with application to the spread of the coronavirus MERS in South Korea in 2015', *Journal of Nonparametric Statistics* **34**, 628–662.

Schweinberger, M. and Handcock, M. S. (2015), 'Local dependence in random graph models: characterization, properties and statistical inference', *Journal of the Royal Statistical Society–Statistical Methodology Series B* **77**, 647–676.

Schweinberger, M. and Stewart, J. (2020), 'Concentration and consistency results for canonical and curved exponential-family models of random graphs', *Annals of Statistics* **48**(1), 374–396.

Snijders, T. A. (2002), 'Markov chain monte carlo estimation of exponential random graph models', *Journal of Social Structure* **3**(2).

Stefanov, S. M. (2004), 'Convex quadratic minimization subject to a Linear constraint and box constraints', *Applied Mathematics Research eXpress* **2004**(1), 17–42.

Stewart, J. R. and Schweinberger, M. (2023), 'Pseudo-likelihood-based M-estimators for random graphs with dependent edges and parameter vectors of increasing dimension', *arxiv.org/abs/2012.07167* . Invited major revision by *The Annals of Statistics*.

Traore, S. A. K., Valero, M., Shahriar, H., Zhao, L., Ahamed, S. and Lee, A. (2022), Enabling cyber-analytics using iot clusters and containers, *in* '2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)', pp. 1798–1803.

Vu, D. Q., Hunter, D. R. and Schweinberger, M. (2013), 'Model-based clustering of large networks', *The Annals of Applied Statistics* **7**(2), 1010 – 1039.

Wainwright, M. and Jordan, M. (2008), 'Graphical models, exponential families, and variational inference', *Foundations and Trends@ in Machine Learning* **1**(1-2), 1–305.

Zheng, X., Zeng, D., Li, H. and Wang, F. (2008), 'Analyzing open-source software systems as complex networks', *Physica A: Statistical Mechanics and its Applications* **387**(24), 6190–6200.

Zimmermann, M., Staicu, C.-A., Tenny, C. and Pradel, M. (2019), Small world with high risks: A study of security threats in the npm ecosystem, *in* '2019 USENIX Security Symposium, USENIX Security '19'.

A Computational details for estimation

Estimation of our model is challenging because of the normalizing constants c_{kk} in the likelihood and the discrete mixture model for the block assignments. We bypass these issues by extending recently developed approximate two-step estimation methods to directed networks, estimating the structural parameters in two steps.

Formally, the full likelihood of our model can be written as follows

$$\mathscr{L}(\boldsymbol{g},\boldsymbol{x};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}) = \sum_{z\in\mathscr{Z}} L\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) = \sum_{z\in\mathscr{Z}} p_{\boldsymbol{\eta}}\left(\boldsymbol{Z}=\boldsymbol{z}\right)\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma). \tag{9}$$

where each firm's type is i.i.d. multinomial, so the distribution $p_{\boldsymbol{\eta}}(\boldsymbol{z})$ is

$$Z_i|\eta_1,...,\eta_K \overset{\text{iid}}{\sim} \text{Multinomial}\left(1;\eta_1,...\eta_K\right) \text{ for } i=1,...,N$$

Since maximizing (9) involves a sum over all possible type allocations of each node as well as a nested normalizing constant in $\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma)$, evaluating the function is infeasible. To ameliorate this issue, we divide estimation into two parts. First, we estimate the type of each node and, second, we estimate the parameters $\boldsymbol{\alpha},\boldsymbol{\beta}$, and γ conditional on the type of each node. This two-step algorithm exploits the fact that our model corresponds to a stochastic blockmodel with covariates when $\gamma=0$, i.e. when there are no link externalities.

A.1 STEP 1: Approximate estimation of unobserved block structure

A.1.1 Variational EM algorithm with MM updates

To recover the types and the block structure of the network, we approximate the model using a stochastic blockmodel. Define

$$L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) := p_{\eta}(\boldsymbol{z})\pi(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma=0) \tag{10}$$

as the likelihood of a stochastic blockmodel, which means that

1. Each node belongs to one of K blocks/types;

2. Each link is conditionally independent, given the block structure

Then under some conditions – namely, that the network is large and each block/type is not too large with respect to the network (Babkin et al., 2020; Schweinberger, 2020; Schweinberger and Stewart, 2020) – we have

$$L\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) \approx L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) \tag{11}$$

Variational methods for the stochastic blockmodel are relatively standard and involve estimating an approximating distribution $q_\xi(\boldsymbol{z})$ that minimizes the Kulback-Leibler divergence from the true likelihood. This can be achieved in several ways, but usually, the set of distributions is restricted to the ones that can be fully factorized to simplify computations.

Then the log-likelihood of our model stated in (9) can be lower bounded as follows. Let $q_\xi(\boldsymbol{z})$ be the auxiliary variational distribution characterized by parameter $\boldsymbol{\xi}$ approximating the distribution $p_\eta(\boldsymbol{z})$, then the log-likelihood has a lower-bound, calculated as follows:

$$\begin{aligned}
\ell(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}) &:= \log \sum_{\boldsymbol{z}\in\mathcal{Z}} L\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\gamma,\boldsymbol{\eta}\right) \\
&\approx \log \sum_{\boldsymbol{z}\in\mathcal{Z}} L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) \\
&= \log \sum_{\boldsymbol{z}\in\mathcal{Z}} q_\xi(\boldsymbol{z}) \frac{L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_\xi(\boldsymbol{z})} \\
&\geq \sum_{\boldsymbol{z}\in\mathcal{Z}} q_\xi(\boldsymbol{z}) \log\left[\frac{L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_\xi(\boldsymbol{z})}\right] \\
&= \sum_{\boldsymbol{z}\in\mathcal{Z}} q_\xi(\boldsymbol{z}) \log L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) - \sum_{\boldsymbol{z}\in\mathcal{Z}} q_\xi(\boldsymbol{z}) \log q_\xi(\boldsymbol{z}) \\
&= \mathbb{E}_q \log L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right) + \mathbb{H}(q) \\
&=: \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}).
\end{aligned} \tag{12}$$

where

$$\mathbb{H}(q) = -\sum_{\boldsymbol{z}\in\mathcal{Z}} q_\xi(\boldsymbol{z}) \log q_\xi(\boldsymbol{z})$$

is the entropy of auxiliary distribution $q_\xi(\boldsymbol{z})$. In the third row, we multiply and divide by $q_\xi(\boldsymbol{z})$ and apply Jensen's inequality in the consecutive line.

The best lower bound is obtained by choosing $q_\xi(z)$ from the set of distributions $\mathcal{Q}$ that solves the following variational problem

$$\ell(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}) = \sup_{\boldsymbol{\xi}\in[0,1]^{N\times K}} \ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi})$$

However, this variational problem is usually intractable unless we impose more structure on the problem. In practice, researchers restrict the set $\mathcal{Q}$ to a smaller set of tractable distributions (Wainwright and Jordan, 2008; Mele and Zhu, forthcoming). In the case of the stochastic blockmodel, it is useful and intuitive to restrict $\mathcal{Q}$ to the set of multinomial distributions

$$\boldsymbol{Z}_i \overset{\text{ind}}{\sim} \text{Multinomial}\,(1;\xi_{i1},...\xi_{iK}) \text{ for } i = 1,...,n$$

with $\boldsymbol{\xi}_i$ being the variational parameters. We collect the vectors of variational parameters in the matrix $\boldsymbol{\xi}$. This leads to a tractable lower bound that can be written in closed-form

$$\begin{aligned}
\ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi}) &\equiv \sum_{\boldsymbol{z}\in\mathcal{Z}} q_{\boldsymbol{\xi}}(\boldsymbol{z}) \log\left[\frac{L_0\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{z};\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta}\right)}{q_{\boldsymbol{\xi}}(\boldsymbol{z})}\right] \\
&= \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{K}\sum_{l=1}^{K} \xi_{ik}\xi_{jl} \log \pi_{ij,kl}(g_{ij},\boldsymbol{x}) \\
&+ \sum_{i=1}^{N}\sum_{k=1}^{K} \xi_{ik}\left(\log\eta_k - \log\xi_{ik}\right)
\end{aligned} \tag{13}$$

where the function $\pi_{ij,kl}$ is the conditional probability of a link between i and j of types k and

l, respectively,

$$\begin{aligned} \log \pi_{ij,kl}(g_{ij}, \boldsymbol{x}) &:= g_{ij} \log \left[\frac{\exp\left[u_{ij,kl}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + u_{ji,lk}(\boldsymbol{\alpha}, \boldsymbol{\beta})\right]}{1 + \exp\left[u_{ij,kl}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + u_{ji,lk}(\boldsymbol{\alpha}, \boldsymbol{\beta})\right]} \right] \\ &+ (1 - g_{ij}) \log \left[\frac{1}{1 + \exp\left[u_{ij,kl}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + u_{ji,lk}(\boldsymbol{\alpha}, \boldsymbol{\beta})\right]} \right] \end{aligned} \tag{14}$$

and

$$u_{ij,kl}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = u(\boldsymbol{x}_i, \boldsymbol{x}_j, z_{ik} = z_{jl} = 1, \boldsymbol{z}; \boldsymbol{\alpha}, \boldsymbol{\beta})$$

Note that $g_{ii} := 0$ for all i by definition. The covariate information $\boldsymbol{x}$ encodes in our setting categorical information on the level of repositories. We, therefore, introduce for the pair of repositories (i, j) the covariate vector $\boldsymbol{x}_{ij} = (x_{ij,1}, \ldots, x_{ij,p})$ encoding in the kth entry whether the respective repositories match on the kth covariate. Observing (15), we can write $\pi_{kl}(g_{ij}, \boldsymbol{x}_{ij}, \boldsymbol{z}) = \pi_{ij,kl}(g_{ij}, \boldsymbol{x})$ by dropping the dependence on the particular pair of repositories and the parameters α and β.

Generally, we iteratively maximize (14) in terms of $\boldsymbol{\eta}, \boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ conditional on $\boldsymbol{\xi}$ and then the other way around. We denote the values of $\boldsymbol{\xi}, \boldsymbol{\eta}, \boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ in the tth iteration by $\boldsymbol{\xi}^{(s)}, \boldsymbol{\eta}^{(s)}, \boldsymbol{\alpha}^{(s)}$, and $\boldsymbol{\beta}^{(s)}$ and the two steps comprising the Variational Expectation Maximization algorithm are:

Step 1: Given $\boldsymbol{\xi}^{(s)}$, find $\boldsymbol{\alpha}^{(s+1)}$ and $\boldsymbol{\beta}^{(s+1)}$ satisfying:

$$\ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}^{(s+1)}, \boldsymbol{\beta}^{(s+1)}, \boldsymbol{\eta}^{(s+1)}; \boldsymbol{\xi}^{(s)}) \geq \ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}^{(s)}, \boldsymbol{\beta}^{(s)}, \boldsymbol{\eta}^{(s)}; \boldsymbol{\xi}^{(s)});$$

Step 2: Given $\boldsymbol{\alpha}^{(s+1)}$ and $\boldsymbol{\beta}^{(s+1)}$, find $\boldsymbol{\xi}^{(s)+1}$ satisfying:

$$\ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}^{(s+1)}, \boldsymbol{\beta}^{(s+1)}, \boldsymbol{\eta}^{(s+1)}; \boldsymbol{\xi}^{(s+1)}) \geq \ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}^{(s+1)}, \boldsymbol{\beta}^{(s+1)}, \boldsymbol{\eta}^{(s+1)}; \boldsymbol{\xi}^{(s)}).$$

Step 1: Taking first order conditions with respect to each parameter implies the following closed-form update rules for $\boldsymbol{\eta}$, and $\pi_{kl}^{(s+1)}(d, x_1, \ldots, x_p, \boldsymbol{z})$ follow

$$\eta_k^{(s+1)} := \frac{1}{n} \sum_{i=1}^{n} \xi_{ik}^{(s+1)}, \quad k = 1, \ldots, K,$$

and

$$\pi_{kl}^{(s+1)}(d, x_1, \ldots, x_p, \boldsymbol{z}) := \frac{\sum_{i=1}^{n} \sum_{j \neq i} \xi_{ik}^{(s+1)} \xi_{jl}^{(s+1)} \mathbf{1}\{g_{ij} = d, X_{1,ij} = x_1, \ldots, X_{p,ij} = x_p\}}{\sum_{i=1}^{n} \sum_{j \neq i} \xi_{ik}^{(s+1)} \xi_{jl}^{(s+1)} \mathbf{1}\{X_{1,ij} = x_1, \ldots, X_{p,ij} = x_p\}},$$

for $k, l = 1, \ldots, K$ and $d, x_1, \ldots, x_p \in \{0, 1\}$, respectively. The variables $X_{p,ij} := \mathbf{1}\{x_{ip} = x_{jp}\}$ are indicators for homophily. Generalizations are possible, as long as we maintain the discrete nature of the covariates $\boldsymbol{x}$. Including continuous covariates is definitely possible, but it may result in a significant slowdown of this algorithm. We note that when running the Variational EM (VEM) algorithm, we are not interested in estimating the parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and γ, but only the estimation of probabilities $\pi_{kl}(g, \boldsymbol{x}, \boldsymbol{z})$ for $g \in \{0, 1\}$ and $\boldsymbol{x} \in \mathscr{X}$, where $\mathscr{X}$ is the set of all possible outcomes of the categorical covariates used in Step 1. This feature allows us to speed up the computation of several orders of magnitude (Dahbura et al., 2021).

Step 2: For very large networks, maximizing the lower bound in the second step with respect to $\boldsymbol{\xi}$ for given estimates of $\boldsymbol{\eta}, \boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ is still cumbersome and impractically slow. We thus borrow an idea from Vu et al. (2013) and find a minorizing function $M\left(\boldsymbol{\xi}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}, \boldsymbol{\xi}^{(s)}\right)$ for the lower bound. This consists of finding a function approximating the tractable lower bound $\ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}; \boldsymbol{\xi})$, but simpler to maximize. Such function $M\left(\boldsymbol{\xi}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}, \boldsymbol{\xi}^{(s)}\right)$ minorizes $\ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}; \boldsymbol{\xi})$ at parameter $\boldsymbol{\xi}^{(s)}$ and iteration s of the variational EM algorithm if

$$\begin{aligned} M\left(\boldsymbol{\xi}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}, \boldsymbol{\xi}^{(s)}\right) &\leq \ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}; \boldsymbol{\xi}) \quad \text{for all } \boldsymbol{\xi} \\ M\left(\boldsymbol{\xi}^{(s)}; \boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}, \boldsymbol{\xi}^{(s)}\right) &= \ell_B(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}; \boldsymbol{\xi}^{(s)}) \end{aligned}$$

where $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\eta}$ and $\boldsymbol{\xi}^{(s)}$ are fixed. Maximizing the function M guarantees that the lower bound does not decreases. Extending the approach of Vu et al. (2013) to directed networks, the follow-

ing expression minorizes $\ell_B(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta};\boldsymbol{\xi})$

$$M\left(\boldsymbol{\xi};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right) := \sum_{i=1}^{N}\sum_{j\neq i}^{N}\sum_{k=1}^{K}\sum_{l=1}^{K}\left(\xi_{ik}^2\frac{\xi_{jl}^{(s)}}{2\xi_{ik}^{(s)}}+\xi_{jl}^2\frac{\xi_{ik}^{(s)}}{2\xi_{jl}^{(s)}}\right)\log\pi_{kl}(g_{ij},\boldsymbol{x}_{ij},\boldsymbol{z}) \\ +\sum_{i=1}^{N}\sum_{k=1}^{K}\xi_{ik}\left(\log\eta_k^{(s)}-\log\xi_{ik}^{(s)}-\frac{\xi_{ik}}{\xi_{ik}^{(s)}}+1\right). \tag{15}$$

In Section A.1.2, we detail how sparse matrix multiplication can be exploited to speed up the updates of $\boldsymbol{\xi}^{(s+1)}$

We run this algorithm until the relative increase of the variational lower bound in below a certain threshold to obtain estimates for $\widehat{\boldsymbol{\xi}}$ and $\widehat{\boldsymbol{\eta}}$. We then assign each node to its modal estimated type, such that $\widehat{z}_{ik}=1$ if $\widehat{\boldsymbol{\xi}}_{ik}\geq\widehat{\boldsymbol{\xi}}_{i\ell}$ for all $\ell\neq k$ and for all is.

A.1.2 Exploiting sparse matrix operations for updating variational parameters

Next, we detail how the updates of ξ in Step 1 can be carried out in a scalable manner based on sparse matrix operations. Equation (15) can be written as a quadratic programming problem in ξ with the side constraints that $\sum_{k=1}^{K}\xi_{i,k}=1$ for all $i=1,\ldots,N$ needs to hold. Therefore, we first rearrange the first part of (15):

$$\begin{aligned}
&\sum_{i=1}^{N}\sum_{j\neq i}^{N}\sum_{k=1}^{K}\sum_{l=1}^{K}\left(\xi_{ik}^2\frac{\xi_{jl}^{(s)}}{2\xi_{ik}^{(s)}}+\xi_{jl}^2\frac{\xi_{ik}^{(s)}}{2\xi_{jl}^{(s)}}\right)\log\pi_{kl}(g_{ij},\boldsymbol{x}_{ij})\\
&=\sum_{i=1}^{N}\sum_{k=1}^{K}\sum_{j\neq i}^{N}\sum_{l=1}^{K}\frac{\xi_{ik}^2}{2\xi_{ik}^{(s)}}\xi_{jl}^{(s)}\left\{\log\pi_{kl}(g_{ij},\boldsymbol{x}_{ij})+\log\pi_{lk}^{(s)}(g_{ji},\boldsymbol{x}_{ji})\right\}\\
&=\sum_{i=1}^{N}\sum_{k=1}^{K}\frac{\Omega_{ik}^{(s)}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)})}{2\xi_{ik}^{(s)}}\xi_{ik}^2
\end{aligned}$$

with

$$\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right) := \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(s)}\left\{\log\pi_{kl}^{(s)}(g_{ij},\boldsymbol{x}_{ij})+\log\pi_{lk}^{(s)}(g_{ji},\boldsymbol{x}_{ji})\right\}. \tag{16}$$

This yields for (15)

$$M\left(\boldsymbol{\xi};\boldsymbol{g},\boldsymbol{x},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\eta},\boldsymbol{\xi}^{(s)}\right) = \sum_{i=1}^{N}\sum_{k=1}^{K}\left(\frac{\Omega_{ik}^{(s)}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)})}{2\,\xi_{ik}^{(s)}} - \frac{1}{\xi_{ik}^{(s)}}\right)\xi_{ik}^{2} + \left(\log\eta_{k}^{(s)} - \log\xi_{ik}^{(s)} + 1\right)\xi_{ik}$$
$$= \sum_{i=1}^{N}\sum_{k=1}^{K} A_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)\xi_{ik}^{2} + B_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)\xi_{ik}$$

where

$$A_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right) \coloneqq \frac{\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)}{2\,\xi_{ik}^{(s)}} - \frac{1}{\xi_{ik}^{(s)}}$$

is the quadratic term and

$$B_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right) \coloneqq \log\eta_{k}^{(s)} - \log\xi_{ik}^{(s)} + 1$$

the linear term of the quadratic problem.

To update the estimate of $\boldsymbol{\xi}$, we need to evaluate $A_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ and $B_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ for $i = 1,\ldots,N$ and $k = 1,\ldots,K$, which, when done naively, is of complexity $O(N^2\,K^2)$. Computing $\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ for $A_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ is the leading term driving the algorithmic complexity. It is thus prohibitively difficult for a large number of population members and communities in the network. Note that in our application, this problem is exasperated as we have more than 35,000 nodes and our theoretical result on the adequacy of the estimation procedure assumes that K grows as a function of N. To avoid this issue, we show how $\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ can be evaluated through a series of matrix multiplications. The underlying idea of our approach is that $\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ has an easy form if $\mathbf{g}$ is an empty network with all pairwise covariates set to zero, i.e., $g_{ij} = 0$ and $x_q = 0\;\forall\; i,j = 1,\ldots,N$ and $q = 1,\ldots,p$. Given that, in reality, these are seldom the observed values, we, consecutively, go through all connections where either $g_{ij} = 1$ or $X_{ij,1} = x_1,\ldots,X_{ij,p} = x_p$ for $\sum_{q=1}^{p} x_q \neq 0$ holds

and correct for the resulting error. In formulae, we employ the following decomposition:

$$\begin{aligned}
\Omega_{ik}^{(s)}(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}) \coloneqq & \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(s)}\left\{\log\pi_{kl}^{(s)}(g_{ij},\boldsymbol{x}_{ij})+\log\pi_{lk}^{(s)}(g_{ji},\boldsymbol{x}_{ji})\right\} \\
= & \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(s)}\left\{\log\pi_{kl}^{(s)}(0,0)+\log\pi_{lk}^{(s)}(0,0)\right\} \\
& +\sum_{j\neq i}^{N}\sum_{l=1}^{K}\left[g_{ij}\xi_{jl}^{(s)}\left\{\log\frac{\pi_{kl}^{(s)}(1,\boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0,0)}\right\}+g_{ji}\xi_{jl}^{(s)}\left\{\log\frac{\pi_{lk}^{(s)}(1,\boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0,0)}\right\}\right] \\
& +\sum_{j\neq i}^{N}\sum_{l=1}^{K}\left[(1-g_{ij})\xi_{jl}^{(s)}\left\{\log\frac{\pi_{kl}^{(s)}(0,\boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0,0)}\right\}+(1-g_{ji})\xi_{jl}^{(s)}\left\{\log\frac{\pi_{lk}^{(s)}(0,\boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0,0)}\right\}\right] \\
= & \,\Omega_{ik}^{(s)}\left(\mathbf{0},\mathbf{0},\boldsymbol{\xi}^{(s)}\right)+\Lambda_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right),
\end{aligned} \tag{17}$$

with

$$\begin{aligned}
\Lambda_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right) = & \sum_{j\neq i}^{N}\sum_{l=1}^{K}\left[g_{ij}\xi_{jl}^{(s)}\left\{\log\frac{\pi_{kl}^{(s)}(1,\boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0,0)}\right\}+g_{ji}\xi_{jl}^{(s)}\left\{\log\frac{\pi_{lk}^{(s)}(1,\boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0,0)}\right\}\right] \\
& +\sum_{j\neq i}^{N}\sum_{l=1}^{K}\left[(1-g_{ij})\xi_{jl}^{(s)}\left\{\log\frac{\pi_{kl}^{(s)}(0,\boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0,0)}\right\}+(1-g_{ji})\xi_{jl}^{(s)}\left\{\log\frac{\pi_{lk}^{(s)}(0,\boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0,0)}\right\}\right]
\end{aligned} \tag{18}$$

being the error in $\Omega_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$ arising from assuming an empty network with all categorical covariates set to zero. Following, we show how both terms, $\Omega_{ik}^{(s)}\left(\mathbf{0},\mathbf{0},\boldsymbol{\xi}^{(s)}\right)$ and $\Lambda_{ik}^{(s)}\left(\boldsymbol{g},\boldsymbol{x},\boldsymbol{\xi}^{(s)}\right)$, can be computed through matrix operations, completing our scalable algorithmic approach.

We, first, assume an empty network with all categorical covariates set to zero, $g_{ij}=x_q=0$ for all $i\neq j$ and $q=1,...,p$, to compute $\Omega_{ik}^{(s)}\left(\mathbf{0},\mathbf{0},\boldsymbol{\xi}^{(s)}\right)$ through matrix multiplications. Observe that

$$\begin{aligned}
\Omega_{ik}^{(s)}\left(\mathbf{0},\mathbf{0},\boldsymbol{\xi}^{(s)}\right) &= \sum_{j\neq i}^{N}\sum_{l=1}^{K}\xi_{jl}^{(s)}\left(\log\pi_{kl}^{(s)}(0,\mathbf{0})+\log\pi_{lk}^{(s)}(0,\mathbf{0})\right) \\
&= \sum_{l=1}^{K}\sum_{j\neq i}^{N}\xi_{jl}^{(s)}\left(\log\pi_{kl}^{(s)}(0,\mathbf{0})+\log\pi_{lk}^{(s)}(0,\mathbf{0})\right) \\
&= \sum_{l=1}^{K}\left(\sum_{j=1}^{N}\xi_{jl}^{(s)}-\xi_{il}^{(s)}\right)\left(\log\pi_{kl}^{(s)}(0,\mathbf{0})+\log\pi_{lk}^{(s)}(0,\mathbf{0})\right) \\
&= \sum_{l=1}^{K}\left(\tau_{l}^{(s)}-\xi_{il}^{(s)}\right)\left(\log\pi_{kl}^{(s)}(0,\mathbf{0})+\log\pi_{lk}^{(s)}(0,\mathbf{0})\right)
\end{aligned}$$

holds, where

$$\tau_l^{(s)} := \sum_{j=1}^{N} \xi_{jl}^{(s)}.$$

With

$$\boldsymbol{A}_{\mathbf{0}}^{(s)} := \begin{bmatrix} \tau_1^{(s)} - \xi_{11}^{(s)} & \tau_2^{(s)} - \xi_{12}^{(s)} & \dots & \tau_K^{(s)} - \xi_{1K}^{(s)} \\ \tau_1^{(s)} - \xi_{21}^{(s)} & \tau_2^{(s)} - \xi_{22}^{(s)} & \dots & \tau_K^{(s)} - \xi_{2K}^{(s)} \\ \vdots & \vdots & \ddots & \vdots \\ \tau^{(s)}(1) - \xi_{n1}^{(s)} & \tau_2^{(s)} - \xi_{n2}^{(s)} & \dots & \tau_K^{(s)} - \xi_{nK}^{(s)} \end{bmatrix}$$

and

$$\boldsymbol{\Pi}_{\mathbf{0}}^{(s)} := \begin{bmatrix} \log \pi_{11}^{(s)}(0,\mathbf{0}) & \log \pi_{12}^{(s)}(0,\mathbf{0}) & \dots & \log \pi_{1K}^{(s)}(0,\mathbf{0}) \\ \log \pi_{21}^{(s)}(0,\mathbf{0}) & \log \pi_{22}^{(s)}(0,\mathbf{0}) & \dots & \log \pi_{2K}^{(s)}(0,\mathbf{0}) \\ \vdots & \vdots & \ddots & \vdots \\ \log \pi_{K1}^{(s)}(0,\mathbf{0}) & \log \pi_{K2}^{(s)}(0,\mathbf{0}) & \dots & \log \pi_{KK}^{(s)}(0,\mathbf{0}) \end{bmatrix},$$

it follows that

$$\boldsymbol{A}_{\mathbf{0}}^{(s)} \left\{ \left(\boldsymbol{\Pi}_{\mathbf{0}}^{(s)} \right)^{\top} + \boldsymbol{\Pi}_{\mathbf{0}}^{(s)} \right\} = \left(\Omega_{ik}^{(s)}(0,0,\boldsymbol{\xi}^{(s)}) \right)_{ik}$$

holds. Put differently, we are able to write $\Omega_{ik}^{(s)}(0,0,\boldsymbol{\xi})$ as the (i,k)th entry of multiplying $\boldsymbol{A}_{\mathbf{0}}^{(s)}$ and $\boldsymbol{\Pi}_{\mathbf{0}}^{(s)}(\boldsymbol{z})^{\top} + \boldsymbol{\Pi}_{\mathbf{0}}^{(s)}(\boldsymbol{z})$. The matrix $\pi^{(s)}(1,\mathbf{0})$ can be computed via sparse matrix operations:

$$\begin{aligned} \boldsymbol{\pi}^{(s+1)}(1,\mathbf{0}) = \left\{ \left(\boldsymbol{\xi}^{(s+1)} \right)^{\top} \boldsymbol{g} \circ (\boldsymbol{J} - \boldsymbol{X}_{\mathbf{1}}) \circ \cdots \circ (\boldsymbol{J} - \boldsymbol{X}_{\boldsymbol{p}}) \left(\boldsymbol{\xi}^{(s+1)} \right) \right\} \oslash \\ \left\{ \left(\boldsymbol{\xi}^{(s+1)} \right)^{\top} (\boldsymbol{J} - \boldsymbol{X}_{\mathbf{1}}) \circ \cdots \circ (\boldsymbol{J} - \boldsymbol{X}_{\boldsymbol{p}}) \left(\boldsymbol{\xi}^{(s+1)} \right) \right\}, \end{aligned} \tag{19}$$

where $\boldsymbol{A} \circ \boldsymbol{B}$ denotes the Hadamard (i.e., entry-wise) product of the conformable matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, $\boldsymbol{J}$ is a $N \times N$ matrix whose off-diagonal entries are all one and whose diagonals are all zero. We follow the approach Dahbura et al. (2021) to calculate (19) without breaking the

sparsity of the covariate matrices and $\boldsymbol{g}$. We can then set

$$\boldsymbol{\Pi}_{\mathbf{0}}^{(s)}(\boldsymbol{z}) = \log\left(1 - \boldsymbol{\pi}^{(s+1)}(d = 1, X_1 = 0, \ldots, X_p = 0)\right)$$

and calculate $\Omega_{ik}^{(s)}\left(\mathbf{0}, \mathbf{0}, \boldsymbol{\xi}^{(s)}\right)$ for $i = 1, \ldots, N$ and $k = 1, \ldots, K$.

Next, we correct for the error arising from assuming that $\boldsymbol{g} = \mathbf{0}$ and $\boldsymbol{x} = \mathbf{0}$ holds. Since all covariates are assumed to be categorical, the pairwise covariate vector $\boldsymbol{x}_{ij}$ can have at most 2^p possible outcomes, which we collect in the set $\mathscr{X}$. Given this, one may rewrite the sum over $j \neq i$ as a sum over all possible outcomes of the pairwise covariate information:

$$\begin{aligned}
\Lambda_{ik}^{(s)}\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right) \;=\; & \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left[g_{ij} \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{kl}^{(s)}(1, \boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0, \mathbf{0})} \right\} + g_{ji} \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{lk}^{(s)}(1, \boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0, \mathbf{0})} \right\} \right] \\
& + \sum_{j \neq i}^{N} \sum_{l=1}^{K} \left[(1 - g_{ij}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{kl}^{(s)}(0, \boldsymbol{x}_{ij})}{\pi_{kl}^{(s)}(0, \mathbf{0})} \right\} + (1 - g_{ji}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{lk}^{(s)}(0, \boldsymbol{x}_{ji})}{\pi_{lk}^{(s)}(0, \mathbf{0})} \right\} \right] \\
= & \sum_{\boldsymbol{x} \in \mathscr{X}} \sum_{j \neq i}^{N} \sum_{l=1}^{K} g_{ij} \mathbb{I}(\boldsymbol{x}_{ij} = \boldsymbol{x}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{kl}^{(s)}(1, \boldsymbol{x})}{\pi_{kl}^{(s)}(0, \mathbf{0})} \right\} \\
& + g_{ji} \mathbb{I}(\boldsymbol{x}_{ji} = \boldsymbol{x}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{lk}^{(s)}(1, \boldsymbol{x})}{\pi_{lk}^{(s)}(0, \mathbf{0})} \right\} + (1 - g_{ij}) \mathbb{I}(\boldsymbol{x}_{ij} = \boldsymbol{x}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{kl}^{(s)}(0, \boldsymbol{x})}{\pi_{kl}^{(s)}(0, \mathbf{0})} \right\} \\
& + (1 - g_{ji}) \mathbb{I}(\boldsymbol{x}_{ji} = \boldsymbol{x}) \xi_{jl}^{(s)} \left\{ \log \frac{\pi_{lk}^{(s)}(0, \boldsymbol{x})}{\pi_{lk}^{(s)}(0, \mathbf{0})} \right\}
\end{aligned}$$

With

$$\boldsymbol{\Pi}^{(s)}\left(d, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right) \coloneqq \begin{bmatrix} \log \frac{\pi_{11}(d, \boldsymbol{x})}{\pi_{11}(d, \mathbf{0})} & \log \frac{\pi_{12}(d, \boldsymbol{x})}{\pi_{12}(d, \mathbf{0})} & \ldots & \log \frac{\pi_{1K}(d, \boldsymbol{x})}{\pi_{1K}(d, \mathbf{0})} \\ \log \frac{\pi_{21}(d, \boldsymbol{x})}{\pi_{21}(d, \mathbf{0})} & \log \frac{\pi_{22}(d, \boldsymbol{x})}{\pi_{22}(d, \mathbf{0})} & \ldots & \log \frac{\pi_{2K}(d, \boldsymbol{x})}{\pi_{2K}(d, \mathbf{0})} \\ \vdots & \vdots & \ddots & \vdots \\ \log \frac{\pi_{K1}(d, \boldsymbol{x})}{\pi_{K1}(d, \mathbf{0})} & \log \frac{\pi_{K2}(d, \boldsymbol{x})}{\pi_{K2}(d, \mathbf{0})} & \ldots & \log \frac{\pi_{KK}(d, \boldsymbol{x})}{\pi_{KK}(d, \mathbf{0})} \end{bmatrix}.$$

and the functions $\boldsymbol{\Gamma} \colon \{0, 1\} \to \mathbb{R}^{n \times n}$ and $\boldsymbol{\Lambda}_s \colon \{0, 1\} \to \mathbb{R}^{n \times n}$ $(s = 1, \ldots, p)$ such that

$$\boldsymbol{\Gamma}(d) \coloneqq \begin{cases} \boldsymbol{G} & (d = 1) \\ \boldsymbol{J} - \boldsymbol{G} & (d = 0) \end{cases}$$

and

$$\boldsymbol{\Delta}(\boldsymbol{x}) \coloneqq \boldsymbol{X_1}^{x_1} \circ (\boldsymbol{J} - \boldsymbol{X_1})^{1-x_1} \circ \cdots \circ \boldsymbol{X_p}^{x_p} \circ (\boldsymbol{J} - \boldsymbol{X_p})^{1-x_p}$$

we can write

$$\Lambda_{ik}^{(s)}\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right) = \left(\sum_{\boldsymbol{x} \in \mathscr{X}} \sum_{d \in \{0,1\}} \boldsymbol{\Gamma}(d) \circ \boldsymbol{\Delta}(\boldsymbol{x}) \, \boldsymbol{\xi}^{(s)} \, \boldsymbol{\Pi}^{(s)}\left(d, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right) + \left(\boldsymbol{\Gamma}(d) \circ \boldsymbol{\Delta}(\boldsymbol{x})\right)^{\top} \boldsymbol{\xi}^{(s)} \, \boldsymbol{\Pi}^{(s)}\left(d, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right)^{\top} \right)_{ik}$$

Extending (19) to generic covariate values $\boldsymbol{x}$ yields

$$\begin{aligned} \boldsymbol{\pi}^{(s+1)}(1, \boldsymbol{x}) = & \left\{ \left(\boldsymbol{\xi}^{(s+1)}\right)^{\top} \boldsymbol{g} \circ \boldsymbol{X_1}^{x_1} \circ (\boldsymbol{J} - \boldsymbol{X_1})^{1-x_1} \circ \cdots \circ \boldsymbol{X_p}^{x_p} \circ (\boldsymbol{J} - \boldsymbol{X_p})^{1-x_p} \left(\boldsymbol{\xi}^{(s+1)}\right) \right\} \oslash \\ & \left\{ \left(\boldsymbol{\xi}^{(s+1)}\right)^{\top} \boldsymbol{X_1}^{x_1} \circ (\boldsymbol{J} - \boldsymbol{X_1})^{1-x_1} \circ \cdots \circ \boldsymbol{X_p}^{x_p} \circ (\boldsymbol{J} - \boldsymbol{X_p})^{1-x_p} \left(\boldsymbol{\xi}^{(s+1)}\right) \right\}, \end{aligned}$$

enabling the evaluation of $\boldsymbol{\Pi}^{(s)}\left(d, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right)$ for $\boldsymbol{x} \in \mathscr{X}$. Following Dahbura et al. (2021), we can still evaluate this matrix by sparse matrix operations.

In sum, we have shown how to compute the terms $\Omega_{ik}^{(s)}\left(\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{\xi}^{(s)}\right)$ and $\Lambda_{ik}^{(s)}\left(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\xi}^{(s)}\right)$ through matrix multiplications. Plugging in these terms into (17) enables a fast computation of $\Omega_{ik}^{(s)}(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\xi}^{(s)})$, which is the computational bottleneck for evaluating the quadratic term needed to update $\boldsymbol{\xi}^{(s)}$ to $\boldsymbol{\xi}^{(s+1)}$.

A.2 STEP 2: Estimation of structural utility parameters

In the second step, we condition on the approximate block structure estimated in the first step $\widehat{\boldsymbol{z}}$ and estimate the structural payoff parameters. We compute the conditional probability of a link within types and between types

$$p_{ij}(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma; \widehat{\boldsymbol{z}}) = \begin{cases} \Lambda\left(u_{ij}(\alpha_w, \beta_w) + u_{ji}(\alpha_w, \beta_w) + 4\gamma \sum_{r \neq i,j} I_{ijr} g_{jr} g_{ir}\right) & \text{if } \widehat{z}_i = \widehat{z}_j \\ \Lambda\left(u_{ij}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + u_{ji}(\boldsymbol{\alpha}, \boldsymbol{\beta})\right) & \text{otherwise} \end{cases}$$

where $I_{ijr} = 1$ if $\widehat{z}_i = \widehat{z}_j = \widehat{z}_r$ and $I_{ijr} = 0$ otherwise; and $\Lambda(u) = e^u/(1+e^u)$ is the logistic function.

The maximum pseudolikelihood estimator (MPLE) solves the following maximization problem

$$\begin{aligned}(\widehat{\boldsymbol{\alpha}}, \widehat{\boldsymbol{\beta}}, \widehat{\gamma}) &= \arg\max_{\boldsymbol{\alpha},\boldsymbol{\beta},\gamma} \ell_{PL}(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma; \widehat{\boldsymbol{z}}) \\ &= \arg\max_{\boldsymbol{\alpha},\boldsymbol{\beta},\gamma} \sum_{i \neq j}^{n} \left[g_{ij} \log p_{ij}(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma; \widehat{\boldsymbol{z}}) + (1 - g_{ij}) \log(1 - p_{ij}(\boldsymbol{g}, \boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \gamma; \widehat{\boldsymbol{z}})) \right]\end{aligned}$$

In practice the estimator maximizes the log of the product of conditional probabilities of linking. The asymptotic theory for this estimator is in Boucher and Mourifie (2017) and Stewart and Schweinberger (2023). It can be shown that the estimator is consistent and asymptotically normal. As long as the estimator for $\boldsymbol{z}$ provides consistent estimates, the estimator for the structural parameters is well-behaved.

B Additional figures and estimates

Figure A1: Value of the lower bound of the likelihood in the first step of our estimation-

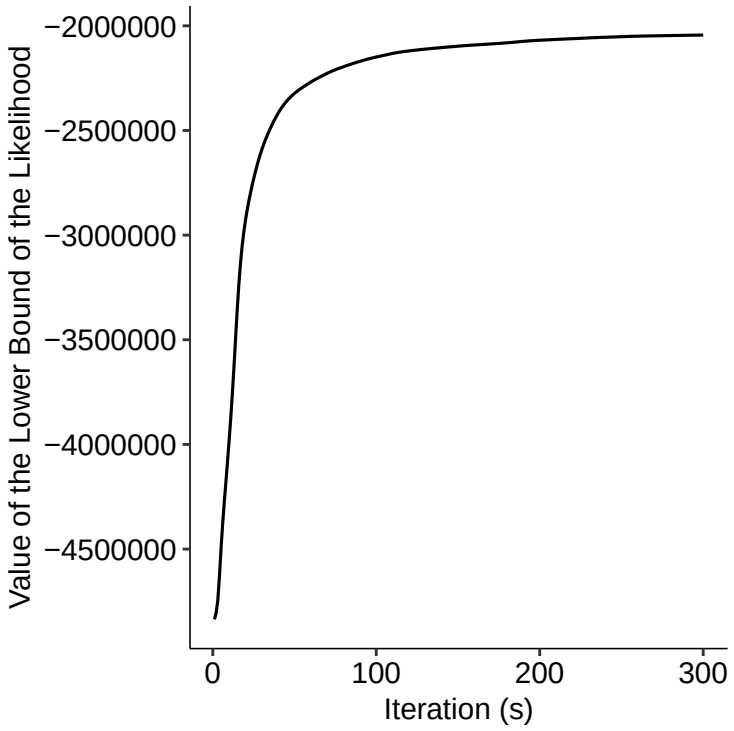


Figure A2: Comparison of k-step systemicness for the 1% of repositories in the Rust Cargo ecosystem with the highest number of dependent repositories (in-degree). The k-step systemicness measures how many repositories the average vulnerable repository renders vulnerable k steps away. The solid black line shows the average k-step systemicness when no node is protected, while the dashed and dot-dashed black lines show the average k-step systemicness when the 10 and 100 most systemic repositories are perfectly protected against vulnerabilities, respectively. The dashed and dot-dashed blue line shows the average k-step systemicness when systemicness for protected nodes are measued as the number of depdents a repository has (in-degree).