Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem

Chengwei Liu*
College of Intelligence and Computing, Tianjin University
Tianjin, China
chengwei001@e.ntu.edu.sg

Sen Chen†
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Lingling Fan
College of Cyber Science, Nankai University
Tianjin, China
linglingfan@nankai.edu.cn

Bihuan Chen
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University
Shanghai, China

Yang Liu
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore

Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University
Shanghai, China

ABSTRACT

Third-party libraries with rich functionalities facilitate the fast development of JavaScript software, leading to the explosive growth of the NPM ecosystem. However, it also brings new security threats that vulnerabilities could be introduced through dependencies from third-party libraries. In particular, the threats could be excessively amplified by transitive dependencies. Existing research only considers direct dependencies or reasoning transitive dependencies based on reachability analysis, which neglects the NPM-specific dependency resolution rules as adapted during real installation, resulting in wrongly resolved dependencies. Consequently, further fine-grained analysis, such as precise vulnerability propagation and their evolution over time in dependencies, cannot be carried out precisely at a large scale, as well as deriving ecosystem-wide solutions for vulnerabilities in dependencies.

To fill this gap, we propose a knowledge graph-based dependency resolution, which resolves the inner dependency relations of dependencies as trees (i.e., dependency trees), and investigates the security threats from vulnerabilities in dependency trees at a large scale. Specifically, we first construct a complete dependency-vulnerability knowledge graph (DVGraph) that captures the whole NPM ecosystem (over 10 million library versions and 60 million well-resolved dependency relations). Based on it, we propose a novel algorithm (DTResolver) to statically and precisely resolve dependency trees, as well as transitive vulnerability propagation paths, for each package by taking the official dependency resolution rules into account. Based on that, we carry out an ecosystem-wide empirical study on vulnerability propagation and its evolution in dependency trees. Our study unveils lots of useful findings, and we further discuss the lessons learned and solutions for different stakeholders to mitigate the vulnerability impact in NPM based on our findings. For example, we implement a dependency tree based vulnerability remediation method (DTReme) for NPM packages, and receive much better performance than the official tool (`npm audit fix`).

ACM Reference Format:
Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510142

1 INTRODUCTION

Due to the rapid growth of functionality complexity in software applications, software componentization has become an irresistible trend in software development, leading to the boosting of third-party libraries. As investigated, over 1.7 million Node.js libraries have been published on NPM [13, 14] (a node package manager) to facilitate software development. As Contrast Security [5] revealed, third-party libraries appear in a majority (79%) of today's software [76]. However, every coin has two sides. Although using libraries reduces development cost and time, these integrated libraries pose a new security threat to the software ecosystem in practice, that vulnerabilities in these libraries may expose software that depend on them under security risks constantly [52, 83–85]. For example, lodash [9], a widely-used JavaScript utility library with over 80 million downloads as dependencies per month, is identified to have severe vulnerability of prototype pollution and exposes 4.35 million projects on GitHub to the potential risk of being attacked [31].

Previous works have investigated vulnerability impact across the NPM ecosystem, while their approaches either only statically consider direct dependencies [52], or excessively analyze dependencies based on static reachability reasoning [85] which may introduce inaccurate transitive dependencies (illustrated by the motivating example of Figure 2 (b) in Section 2.1) resulting in false-positive vulnerability warnings. None of the existing approaches provide precise dependencies, especially the inner complex relations among

*Also with Nanyang Technological University.
†Sen Chen is the corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510142

dependencies of software, at a large scale, which makes the impact of their analysis weakened and limits further solutions (i.e., precise remediation) to be proposed. Although some existing SCA tools (e.g., Snyk [30] and Blackduck [1]) support NPM dependency analysis for user projects, most of them retrieve dependency trees from real installation rather than static reasoning. Besides, dependencies, as well as vulnerabilities in dependencies, are actually under dynamic change over time due to the flexibility of semantic versioning [29]. Therefore, although existing work has also investigated the impact of vulnerabilities [52, 85], it is still challenging to analyze the evolution of vulnerability propagation existing in dependencies at a large scale without static and precise dependency resolution, not to mention to derive practical solutions on preventing vulnerabilities from dynamically being introduced into dependencies.

To fill these gaps, we face the following challenges. 1) **Completeness**. NPM ecosystem is the largest platform with over 1 million published packages, which are hard to be fully analyzed. Some existing work only either analyzed with a limited number of libraries [55, 64, 77], or studied vulnerabilities in dependency trees of limited projects [39, 44, 61]. Besides, NPM allows various ways to reuse third-party libraries as dependency constraints [23], which are also hard to fully capture and resolve. 2) **Accuracy**. Existing work [49, 64, 66, 77] only conduct dependency-based analysis by identifying transitive dependencies via reachability reasoning while neglecting the NPM-specific dependency resolution rules [16], which would lead to inaccurate results. 3) **Efficiency**. Even though real installation can precisely retrieve dependency trees, installing NPM packages (i.e., `npm install`) is known to always take minutes per run [18], which is obviously not efficient enough to support large-scale studies. 4) **Dynamic updates**. NPM packages are known to have the most dependencies. Any new release of libraries in dependency trees could lead to changes in installed dependencies for installation afterwards (as the example in Figure 6), which even complicates the management of dependencies, as well as the vulnerability propagation in dependencies. Thus, it is challenging to explore the vulnerability propagation evolution over time.

In this paper, to overcome these challenges, **1)** we implement a robust dependency constraint parser to tackle the diversity of NPM dependency constraints, and based on it, we construct a complete dependency-vulnerability knowledge graph (DVGraph) to capture the dependency relations among all NPM packages (over 1.14 million libraries and 10.94 million versions), as well as over 800 known CVEs (Common Vulnerabilities and Exposures) [4] from NVD [11], which further supports the thorough analysis of vulnerability propagation. **2)** We propose an accurate DVGraph based dependency resolution algorithm (DTResolver) to calculate dependency trees[1] at any installation time. It integrates the official dependency resolution rules and DVGraph-based reasoning to simulate the process of installation (illustrated by the motivating example of Figure 2 (c) in Section 2.1). Our DTResolver is validated to have an accuracy of over 90% of resolved dependency trees being exactly the same comparing to real installation. (3) We further conduct an empirical study on vulnerability propagation in dependency trees. First, we investigate the characteristics of dependency trees brought by NPM dependency resolution (details are available at our supplementary material), based on which, we analyze the impact and features of vulnerability propagation in dependency trees, particularly the vulnerabilities from transitive dependencies. Besides, we also extend our study to time dimension to investigate the evolution of vulnerability propagation in dependency trees over time to unveil the reasons of vulnerabilities being introduced in dependency trees, as well as possible solutions.

Through our empirical study, we conclude some findings as follows. For example, **1)** We statistically prove that vulnerabilities widely exist in the dependencies of NPM packages (over one-quarter of library versions from 20% of libraries), even in the latest versions (16% of libraries). Besides, vulnerabilities from direct dependencies are widely neglected (over 30% affected library versions). **2)** Known vulnerabilities are causing a larger impact over time, with more affected packages and more vulnerabilities in dependency trees. Most vulnerabilities (93%) are introduced into dependency trees before they are discovered, and most fixing versions (87%) of them are released before they get published. Based on these timely releases, most of the vulnerable dependencies can be removed (90%) along with time (in average it takes a year), but there are still 40% of vulnerabilities unable to get thoroughly excluded. **3)** Considerable user projects contain unavoidable vulnerabilities even though we have exhausted all possible dependency trees. More findings can be found in Section 5. Additionally, since the severe situation of vulnerability propagation is complicated and requires efforts from different roles to mitigate, we also conclude actionable solutions from different stakeholders in the supply chain of third-party libraries based on our findings in Section 6.

In summary, we make the main contributions as follows.

- We design and construct a complete and precise DVGraph for the whole NPM ecosystem by leveraging a robust dependency constraint parser. The construction and maintenance pipelines take 20 person-months.
- We propose a novel algorithm (DTResolver) based on DVGraph to statically and precisely resolve the dependency trees for any installation time with high accuracy (over 90%), which is validated by around 100k representative packages.
- We conduct the first large-scale empirical study based on over 50 million resolved dependency trees (calculated on an 8-core machine for one month) to peek into the vulnerability propagation and the evolution of vulnerability propagation over time and provide useful findings.
- We provide an in-depth discussion, including lessons learned and actionable solutions, which provide useful insights to improve the security of the whole NPM ecosystem for different stakeholders, such as the proposed remediation (DTReme) that excludes more vulnerabilities than the official tool `npm audit fix`.
- We have made the relevant analytic data publicly available[2] to facilitate the relevant research on the NPM ecosystem.

Figure 1 demonstrates the overview of our work, including the dependency-vulnerability knowledge graph construction (Section 3), dependency tree resolution, vulnerable path identification, and their validations (Section 4), a large-scale empirical study (Section 5) and discussion on lessons learned and solutions, as well as possible research directions (Section 6).

[1]The resolved dependency graph when a given root package is installed.

[2]https://sites.google.com/view/npm-vulnerability-study/

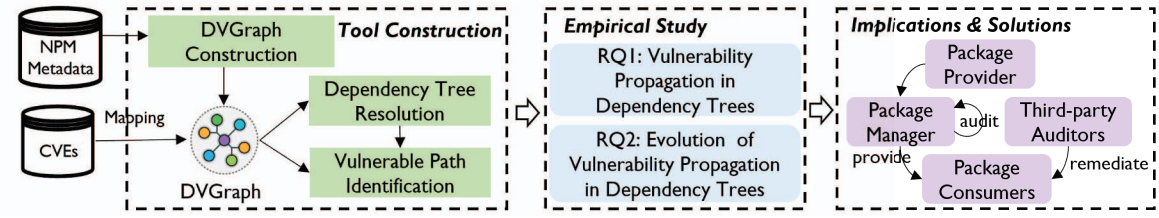
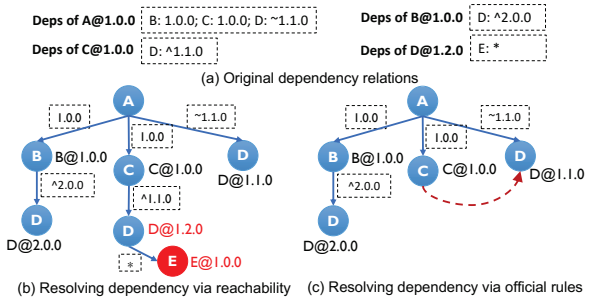


Figure 1: Overview of our work



(1) D has four versions: 1.0.0, 1.1.0, 1.2.0, 2.0.0
(2) According to node-semver, "~1.1.0 represents "<1.2.0 and >=1.1.0", "^1.1.0 represents "<2.0.0 and >=1.1.0", "*" represents any version.

Figure 2: An example of NPM dependency resolution

2 MOTIVATING EXAMPLE AND BACKGROUND

2.1 Motivating Example

Here, we present an example to illustrate why it is unreliable to conduct vulnerability propagation analysis via existing reachability analysis. Figure 2 presents an example, where each package is represented in the format of library@version. Figure 2 (a) presents the original dependency relations of packages in the format of library:constraint. Figure 2 (b) and Figure 2 (c) present the resolved dependency of A@1.0.0 via reachability reasoning (i.e., dependency reach in [85]) and NPM official rules, respectively. Specifically, when resolving the dependency of C@1.0.0, reachability analysis selects D@1.2.0 because 1.2.0 is the highest satisfying version of dependency constraint ^1.1.0 from C@1.0.0 to D, and package E@1.0.0 is also selected. However, this may lead to inaccurate results. NPM follows its own principle to resolve dependencies during installation [16]. For example, NPM takes real installation context into account when resolving dependency trees (e.g., allowing existing versions to be reused), while resolving transitive dependency via reachability fails to involve such rules. As presented in Figure 2 (c), since existing D@1.1.0 satisfies ^1.1.0, it will directly reuse D@1.1.0 instead of resolving a new one. Thus, D@1.2.0 and E@1.0.0 are wrongly resolved by reachability analysis in Figure 2 (b).

2.2 Background

We briefly introduce several concepts related to dependency management and vulnerabilities in the NPM ecosystem.

NPM Package Metadata. NPM [13] is the official package manager for Node.js and provides a public NPM registry to maintain

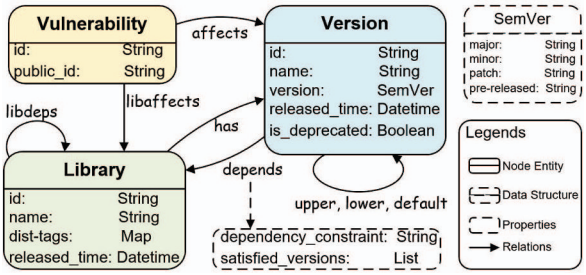


Figure 3: Schema of NPM dependency-vulnerability graph

information of all published libraries (i.e., NPM package metadata [22]). Such metadata describes all information of a given library and its released versions, and the dependency relation of each version, which is useful for indexing libraries and resolving dependencies when installing a library.

Dependency Constraints. Dependencies of package p are specified as a list of key-value pairs in metadata, where key represents a library lib_a and value represents the allowed version range (i.e., constraint) that p should follow when selecting the version of library lib_a during installation. There are different types of constraints in NPM [21], such as Version and Range, Tag, URLs (i.e., Git urls and Remote links), and local paths (Directory and File).

Semantic Versioning. It has been proposed as a solution of version control when maintaining package dependencies. It defines version numbers presented in the format of Major.Minor.Patch. When a new version is released, Major increases when incompatible API changes have been taken out, Minor raises when functionality changes are still backward compatible, and Patch grows when backward compatible bug fixes have been made.

Vulnerabilities in NPM Packages. Vulnerabilities in NPM packages are included in the CVE reports. Each CVE is published with detailed information about vulnerabilities and their references. Specifically, the affecting libraries and the corresponding versions are always described in free text descriptions, which raises more efforts to retrieve and map CVEs with exact affecting library versions.

3 DVGRAPH CONSTRUCTION

To support large scale dependency-based vulnerability analysis with high accuracy and efficiency, we design and implement a set of infrastructures to construct and maintain a complete and precise dependency-vulnerability graph (DVGraph).

DVGraph Definition and Schema. The NPM *DVGraph* is designed as a directed knowledge graph with labeled vertices and directed edges: $G = (N, E)$, where N represents all node entities with different types in G, i.e., Library (*Lib*), Version (*Ver*), and

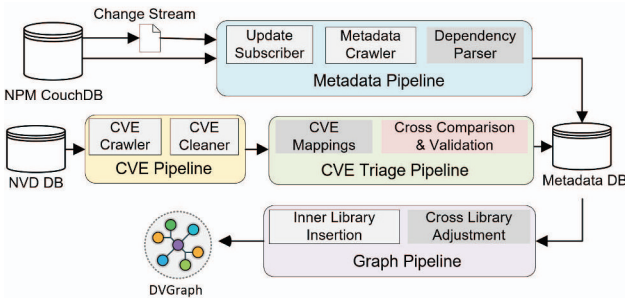


Figure 4: Automated data processing framework

Vulnerability (*Vul*). E represents the relations between nodes, 8 types in total, including inner-library relations (i.e., *has*, *upper*, and *lower*), cross-library relations (i.e., *depends*, *default*, and *libdeps*), and vulnerability-related relations (i.e., *affects* and *libaffects*). Figure 3 shows the detailed schema of each type of nodes and relations. Specifically, we present relations in the format of "<Node>−<*relation*>⟶<Node>" as follows: ❶ Inner-library relations: "Lib_1−*has*⟶ Ver_1" denotes Lib_1 has a released version Ver_1, "Ver_1−*upper/lower*⟶ Ver_2" denotes the next semantically upper/lower version of Ver_1 is Ver_2. ❷ Cross-library relations: "Ver_1−*depends*⟶ Lib_2" denotes Ver_1 directly depends on Lib_2. Specifically, for each *depends*, we denote "Ver_1−*default*⟶ Ver_2", where Ver_2 is the latest version of Lib_2 that satisfies the *dependency_constraint* of *depends* , because NPM takes the latest satisfied version when resolving dependency constraint to specific versions by default. Moreover, "Lib_1−*libdeps*⟶ Lib_2" denotes that there exists at least one version of Lib_1 depending on Lib_2. ❸ Vulnerability-related relations: Since vulnerabilities usually exist in multiple versions, "Vul_1−*affects*⟶ Ver_1" and "Vul_1−*libaffects*⟶ Lib_1" denote that Vul_1 exists in Ver_1 and exists in at least one version of Lib_1, respectively.

DVGraph Construction and Maintenance. The DVGraph is constructed as a knowledge graph in Neo4j [12]. NPM resolves dependency constraints with the highest satisfying version by default, which claims a high demand for real-time updates of dependency data. Therefore, we conduct an automated data processing framework for long-term maintenance as shown in Figure 4, including: (1) Metadata Pipeline, subscribes, daily collects, cleans, and processes new coming NPM package metadata, and preserves them in our metadata database. (2) CVE Pipeline, collects CVE feeds [27] from the NVD database. Since some information in CVE feeds are usually in plain text, a CVE cleaner is designed to filter the languages and identify the affected libraries, as well as affecting version ranges, and save them as the initial results. (3) CVE Triage Pipeline, is a semi-automated pipeline. It helps experienced security analysts process, label and confirm the newly crawled CVE data with corresponding affected libraries and versions. (4) Graph Pipeline, parses the new coming metadata and mapped CVE data, calculates the operations (i.e., adding, altering, and deleting nodes and edges) to be done on DVGraph, and finally executes them.

Specifically, there are two challenges during DVGraph update.

- **Dependency Parser**: The diversity of NPM dependency constraints makes it complex to resolve proper versions. Wrongly handled dependency relations can cause deviation when reasoning transitive dependencies, and none of the existing work has taken all major types into account. To this end, we propose and develop **a robust dependency constraint parser** based on *node-semver* [28]. It handles not only semver version ranges but also version tags, Git URLs, and remote links [21].
- **CVE Mappings**: Even though the CVE pipeline crawls and processes CVE data and automatically recognizes affected libraries and version ranges, there still could be mislabeling since they are usually given in free-text descriptions [58, 59]. Therefore, we implement the CVE Triage Pipeline as shown in Figure 4 and have devoted four experienced analysts to check the CVE mappings of affected library and version ranges, some existing famous vulnerability databases [32] [34] are also involved as references. After confirmation, all affected versions can be sorted out by the constraint parser.

Remarks: It takes 3 months for 3 co-authors and experienced software engineers to implement and test, and another 4 security analysts to conduct daily validation of CVE mappings.

Table 1: Graph statistics

Elements	#Instances	Elements	#Instances
Lib	1,147,558	*has*	10,939,334
Ver	10,939,334	*upper*	9,804,406
Vul	815	*lower*	9,804,406
depends	62,232,906	*affects*	23,217
default	61,940,009	*libaffects*	830
libdeps	4,216,742	Graph size	15.15GB

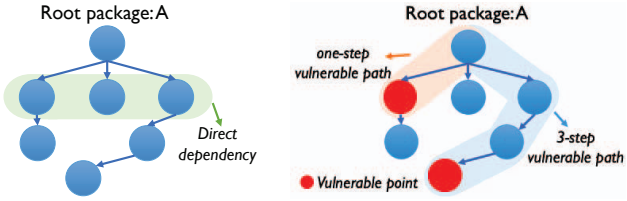
DVGraph Statistics. To carry out large-scale studies on NPM vulnerability propagation and evolution, we take a snapshot of DVGraph by the end of 2020 (Most of CVEs before 2020 are finalized) to conduct further analysis. Table 1 shows the basic statistics of the snapshot. 1,147,558 libraries and 10,939,334 versions have been captured in the DVGraph, among which 62,232,906 direct dependencies (*depends*) are captured in DVGraph. Besides, 815 CVEs have been included in DVGraph, generating 830 *libaffects* and 23,217 *affects* from these CVEs to 624 libraries and 14,651 versions, respectively. Overall, the storage size of the DVGraph snapshot is over 15GB.

DVGraph and CVE Mapping Validations. To roughly validate the coverage of DVGraph, we take a snapshot of the metadata database to compare with it. We find that DVGraph covers 100% of libraries and 99.96% of versions of the metadata database (the rest are unpublished [25]). Besides, only 0.36% of direct dependencies are not captured in DVGraph because the corresponding dependency libraries are missing in the NPM registry. Furthermore, we also find 0.47% of *depends* cannot be resolved to any satisfying version due to no satisfying version or invalid dependency constraints.

Remarks: As for the validation of CVE mappings, besides 4 security analysts have validated all new-coming CVEs every day, the co-authors still manually check the mappings between all CVEs and the affected library versions after we conduct the snapshot. In particular, since the publish time of CVEs differs from several sources [4, 11, 32], we take the earliest ones.

4 DEPENDENCY TREE RESOLUTION AND VULNERABLE PATH IDENTIFICATION

To facilitate large-scale studies, we propose a novel methodology on dependency tree resolution to statically resolve the dependency



(a) Dependency tree of package A (b) Vulnerable Paths

Figure 5: Examples of dependency tree and vulnerable paths (each node represents a package with an exact version)

trees of packages based on the complete and well-maintained DVGraph. It not only resolves dependency trees precisely but also preserves the high efficiency of static analysis, which enables most of SCA [33] scanning tasks to be carried out without real installation, e.g., license violation detection, untrustworthy dependency detection, etc. Therefore, identifying vulnerabilities and corresponding vulnerable paths that vulnerable packages propagate to affect the root package can also be accurately and efficiently captured.

4.1 Dependency Tree Definition

Third-party libraries are usually installed along with the installation of user projects as dependencies, during which NPM follows its own dependency resolution rules and resolves dependency constraints into specific library versions recursively. These resolved library versions and dependency relations among them form a directed dependency graph.

Precisely, we denote the graph as **dependency tree**, $DT_{root} = (Ver, Dep)$, $root$ denotes the root package (could be user project) that is to be installed, and Ver represents the set of library versions that are resolved during the real installation, and Dep is the set of dependency relations among Ver. To distinguish with *depends* in DVGraph, we further define $v_i \xrightarrow{dep} v_j$ as the specific dependency relation between library version v_i and v_j in DT. Figure 5a shows the dependency tree DT_A of package A starting from package A (the root package) and connecting all resolved library versions with dependency relations. The $Vers$ that are directly depended on by the root package A are denoted as **direct dependencies**, and the others that are not directly depended on by A are denoted as **indirect (transitive) dependencies**, as presented in Figure 5a.

4.2 Dependency Tree Resolution

Lots of work [49, 64, 66, 77] has been carried out to investigate transitive dependencies in the NPM ecosystem. However, none of them have taken into consideration the platform-specific dependency resolution rules [6], which could result in inaccurate dependencies being resolved (illustrated by the example shown in Figure 2). To fill this gap, we aim to **statically** resolve dependency trees that are consistent with what NPM dynamically resolves and installs during real installation, so that we can identify vulnerabilities and vulnerable paths in dependency trees precisely and efficiently without real installation.

Besides the reachability analysis, there exist other tools [15] scanning vulnerabilities by examining dependencies after real installation. However, such dynamic approaches always take a much longer time than static approaches and are not efficient enough for large-scale analysis. Besides, during installation, NPM manipulates dependencies based on physical trees (connecting $Vers$ based on physical location of installation), which makes the inner dependency relations in logical trees (connecting $Vers$ based on dependency relations) implicit. Therefore, to improve the accuracy and meanwhile reserve the efficiency, we propose a DVGraph-based dependency resolution algorithm (DTResolver) to statically calculate the implicit logical dependency trees without installation.

Specifically, as presented in Algorithm 1, we simulate the folder allocation process [6] during real installation, and recursively resolve dependencies for each selected library versions (lines 5~27). Q denotes a queue to control the recursion process, and Dir represents a virtual and empty directory (as well as an empty physical tree, line 1). For each visited library version in Q, we iterate its dependencies alphabetically (line 8). For each dependency relation, the latest satisfied version will be selected when resolving a certain dependency (line 15). However, such resolving is also influenced by the physical tree. If there is an installed version v_i for the required library that is on a higher position in the physical tree, the installed one will be reused instead of resolving a new version (lines 11~13). Otherwise, a new latest satisfied version will be resolved and installed (lines 15~27). Note that, we denote $install\_path$ to present install position in the physical tree, and when $path\_1 \sqsubseteq path\_2$, it means $path\_2$ is inside $path\_1$. In other words, $path\_1$ is on higher position of $path\_2$ (lines 11, 20).

Meanwhile, we also maintain the individual dependency relations between packages during the simulation process (lines 12, 25). Therefore, we can further recover the logical tree with the physical tree structure and those dependency relations. Besides, we also include lots of individual version selection rules (e.g., priorities on deprecated versions [19] and latest tags [36]) that NPM follows in our algorithm to the best of our knowledge.

Additionally, we extend our algorithm to **time dimension** by adding filters on release time when selecting satisfying versions (comments on line 15), and this empowers the calculation of dependency trees at any previous time from release time. Therefore, more time-based analyses could be carried out.

4.3 Vulnerable Path Identification

Since all known vulnerabilities (CVEs) and their affecting libraries and versions are already well mapped in DVGraph, the resolved dependency trees can further provide the capability to extract dependency paths. Dependency paths connect $Vers$ in DT in series via $Deps$ among them, each path P is a subset of DT with orders, and can be denoted as $P = (PN, PE)$, $P \sqsubseteq DT$, where PN is an ordered list of $Vers$ that can be connected via $Deps$ in the given order, and PE is the ordered list of $Deps$ that connect $Vers$, they satisfy Equation 1 and 2, respectively.

$$\forall\, i.\ i \in [0, k-1) \wedge v_i \in PN \vdash \exists\, d.(d \in Dep \wedge d_{src} = v_i \wedge d_{dst} = v_{i+1}) \quad (1)$$

$$\forall\, e_i.(e_i \in PE) \vdash e_{i\_src} = v_i \wedge e_{i\_dst} = v_{i+1} \quad (2)$$

$$P|_{dst}^{src} = \{P_i \sqsubseteq DT : PN_{i\_0} = v_{src} \wedge PN_{i\_k} = v_{dst})\} \quad (3)$$

Algorithm 1: Dependency Tree Resolution

Input: G: DVGraph, r: given root package, // t: given time
Output: DT_r: Resolved dependency tree of r

$Dir \leftarrow$ new InstallDirectory()
$root\_path \leftarrow \emptyset, Q \leftarrow \emptyset, Deps \leftarrow \emptyset$
Dir.install(r, $root\_path$)
Q.push(r)
// 1. Traverse all resolved dependency nodes by BFS, and simulate real installation to create folders for packages
while $Q \neq \emptyset$ **do**
  $lv \leftarrow Q$.pop()
  $deps \leftarrow \{e \in G : e_{src} = lv \wedge e.type = depends\}$
  foreach $depend \in deps$ **do**
    $vers \leftarrow depend.satisfied\_versions$
    deplib $\leftarrow depend_{dst}$
    if $\exists\ v_i.\ v_i \in Dir \cap vers \wedge v_i.dir\_path \sqsubseteq lv.dir\_path$ **then**
      $r \leftarrow$ (**CREATE** $lv \xrightarrow{dep} v_i$)
      $Deps$.push(r)
    else
      $selected \leftarrow v_i.v_i \in vers \wedge (\forall v_j.v_j \in vers \wedge i \neq j \wedge v_i > v_j)$ // $\wedge v_i.released\_time < t$
      if $Dir \cap vers = \emptyset$ **then**
        $install\_path \leftarrow root\_path$
      else
        foreach $subpath \sqsubseteq lv.dir\_path$ **do**
          if $\neg\exists n.n \in subpath \wedge (deplib - has \rightarrow n)$ **then**
            $install\_path \leftarrow subpath$
            break
      Dir.install($selected$, $install\_path$)
      $r \leftarrow$ (**CREATE** $lv \xrightarrow{dep} selected$)
      $Deps$.push(r)
      Q.push($selected$)
// 2. Recover a dependency tree from install directory and CREATED Deps relations
$Ver_r \leftarrow \{lv : lv \in Dir\}$
$Dep_r \leftarrow Deps$
$DT_{root} \leftarrow <Ver_r, Dep_r>$
return DT_r

where k denotes the length of PN, and e_i denotes the dependency relation between v_i and v_{i+1}. Specifically, there could be multiple paths to one target node in a dependency tree. Considering the cross dependencies, we denote the set of dependency paths from library version v_{src} to library version v_{dst} as $P|^{src}_{dst}$, as defined in Equation 3. Specifically, we denote the set of dependency paths from root package to library version v_{dst} as $P|_{dst}$.

Based on the resolved dependency trees and paths for the installed packages, the algorithm also facilitates the identification of vulnerable packages in dependencies (i.e., **vulnerable point**) and the paths that these vulnerable packages propagate on to affect the root package (i.e., **vulnerable path**), and we also denote them as Equation 4 and 5, respectively. AP_A denotes the set of $Vers$ in DT_A that are affected by known vulnerabilities, VP denotes the set of P_i whose last node P_{i_k-1} is affection point, Vul denotes all vulnerabilities we have maintained in DVGraph. Moreover, we define **K-step vulnerable path** $VP_{S=k}$ as vulnerable paths that contain k dependency relations, as defined in Equation 6.

$$AP_A = \{v \in Ver_A : \exists n_{vul}.(n_{vul} \in Vul \wedge (n_{vul} - affects \rightarrow v))\} \quad (4)$$

$$VP_A = \{P|_v \sqsubseteq DT_A : (P|_v)_{k-1} \in AP_A\} \quad (5)$$

$$VP_{S=k} = \{P|_v \in VP_A : |(PE|_v)| = k\} \quad (6)$$

Examples of vulnerable points and paths are given in Figure 5b. Thus, we implement a vulnerable path extractor by reverse Depth First Search (DFS) to exhaustively find dependency relations from vulnerable points to the root node in a dependency tree.

4.4 Validation of Dependency Tree Resolution

We validate the DTResolver by comparing the dependency trees resolved by DTResolver with the real installed ones. Moreover, we take npm-remote-ls [24] as a baseline method when comparing, which is a widely-used public API to get dependency trees without real installation in practice, and it exactly follows the dependency reach to derive dependency trees.

Data Selection Our validation is based on the data collected by two criteria: (1) Popularity, for each popularity metrics (i.e., most stars, most forks, most downloaded in the past, past 3 years, and last year), we select the top 2,000 libraries respectively. (2) Centrality, for each centrality metric (i.e., most in and out degree), we also select the top 2,000 libraries and top 20K versions. respectively. For libraries, we take the highest patch version for each minor version. Finally, 103,609 versions from 15,673 libraries are sorted out.

Experiment. Based on the collected data, we first collect all installation dependency trees (Install Tree) for each version from real installation (npm-install [17] and npm-ls [20]), 82,415 of these versions are successfully collected after excluding those with installation errors. We also collect the dependency trees (Remote Tree) from npm-remote-ls. Moreover, to compare with real installation results, we update the graph after all Install Trees are well collected, so that all packages in the Install Trees are updated into the graph. Based on it, we further compute the dependency trees (Graph Tree) for all versions with their corresponding installation times.

Evaluation of DTResolver. According to the result, **90.58%** of Graph Trees exactly match the Install Trees after ignoring incalculable cases, e.g., having bundled dependencies [3] and containing dependencies with no released time. While only 53.33% of Remote Trees exactly match the Install Trees, which is because npm-remote-ls have missed some official resolution rules (e.g., priority selection on not deprecated versions). Besides, we further identified two major reasons for mismatched dependency trees: 1) Dependencies are deduplicated [7] in the output of npm ls, which omits some packages and dependency relations to simplify the tree view. 2) Dependencies may not be fully installed due to environmental issues (e.g., some packages may not be installed when the required OS support is missing). Besides, missing library versions (i.e., not in the NPM registry or crawling failure) also cause some missing packages in the dependency trees.

Evaluation of Vulnerability Detection and Vulnerable Path Identification. Besides the evaluation of DTResolver, we also extend to compare the detected vulnerabilities and vulnerable paths. Since the Install Tree retrieved from real installation may be incomplete (e.g., some packages in dependencies are not installed due to environment issues), we evaluate the accuracy of vulnerability detection by the recall of the identified vulnerabilities and vulnerable paths in Graph Tree and Remote Tree. We find that both

DTResolver (98.1%) and npm-remote-ls (97.7%) have similarly high coverage on detecting vulnerable components but vary on identifying vulnerable paths (92.60% vs. 78.31%). This is probably because most dependency constraints are resolved to the highest satisfied version, and dependency reach also follows this rule, therefore, most vulnerable packages can still be identified. However, resolving dependencies via dependency reach neglects the NPM specific resolution rules, which compromises the accuracy on identifying dependency paths.

The results not only prove the quality of the DVGraph and the accuracy of DTResolver, but also the accuracy of vulnerability detection and vulnerable path identification. We take more evaluation details and case analysis on our website (https://sites.google.com/view/npm-vulnerability-study/).

5 LARGE-SCALE EMPIRICAL STUDY

The dependency reachability reasoning adapted in existing work makes it difficult to carry out more fine-grained analysis on dependencies, such as deducing the dynamic changes of vulnerability in dependency trees, due to the neglecting of inner dependency relations (i.e., structure) in dependency trees as discussed in Section 4.2. However, such analysis is vital to unveil the reasons and characteristics of vulnerabilities being introduced as dependencies to support precise remediation for dependency trees and even solutions to mitigate the entire NPM ecosystem. Therefore, our proposed DTResolver is vital, and based on this, we further carry out our study on vulnerability propagation and evolution in the context of dependency trees from these two research questions:

- **RQ1**: (Vulnerability Propagation via Dependency Trees) How do vulnerabilities affect the NPM ecosystem? How do vulnerabilities propagate to affect root packages via dependency tree?
- **RQ2**: (Vulnerability Propagation Evolution in Dependency Trees) How do vulnerability propagation evolves in dependency trees? How do dependency tree changes influence the evolution of vulnerability propagation?

5.1 RQ1: Vulnerability Propagation via Dependency Trees

The goal of this section is to investigate how does NPM dependency resolution influence the vulnerability propagation via dependency trees from two aspects: 1) the propagation of vulnerability via dependency trees and 2) the influence on vulnerability propagation brought by NPM dependency resolution.

5.1.1 ***How many packages are affected by existing known vulnerabilities in the NPM ecosystem?*** Existing studies [52, 85] have unveiled that vulnerabilities in third-party libraries can widely affect the NPM ecosystem via dependencies, while their neglecting on NPM specific dependency resolution rules may lead to inaccurate dependencies (*cf.* Figure 2), resulting in biases in conclusions. Therefore, we re-evaluate the vulnerability impact by computing dependency trees for all packages in the NPM ecosystem and analyzing vulnerability propagation for each of them.

As presented in Table 1, we have captured 815 known vulnerabilities in DVGraph, which exist in 14,651 versions from 624 libraries. The amount of these library versions (directly affected) are relatively small, comparing to the mass of the NPM ecosystem. However, based on the dependency trees we resolved, we find that an astonishing portion (i.e., 24.78%, 2,711,222) of versions, from 19.96% (229,037/1,147,558) libraries, are transitively affected by 416 CVEs, which are introduced from versions of 294 vulnerable libraries.

Besides, since users are always recommended to take the latest version of libraries to get rid of vulnerability, we further analyze the vulnerability propagation in the latest versions of all libraries (1,147,558), and we find that the latest versions of 185,598 libraries (16.17%) are still transitively affected. This finding reveals that latest versions of third-party libraries are also under potential risk of being affected by known vulnerabilities via dependencies.

Moreover, we further notice a bad practice that the latest version of 35.03% (103/294) of vulnerable libraries that have dependent packages are vulnerable. Since NPM usually resolves dependency constraints as the highest satisfying versions, these vulnerable latest versions have a much higher chance of being depended on by other packages than old versions, leading to much higher possibility of distributing indirect vulnerability propagation.

Finding-1: ① It is statistically proved that vulnerabilities are widely existing in dependencies of NPM packages (one-quarter versions of 19.96% libraries across the ecosystem). ② Latest versions of third-party libraries (16.17%) are still under potential risks of being affected by vulnerabilities via dependencies. ③ A considerable portion of vulnerable libs (over 100) that are used by others, still have vulnerable latest versions.

5.1.2 ***How do vulnerabilities propagate to affect root packages via dependency tree?*** Based on the dependency trees (over 10 million) we resolved, we also extract the vulnerable points and corresponding vulnerable paths to investigate how do vulnerabilities propagate to affect root packages via dependency trees.

For vulnerable points, we notice that there is clear centrality that some influential vulnerabilities transitively affect a significant portion of library versions in the NPM ecosystem. Particularly, we find 25 CVEs have affected over 10k libraries or 100k versions (1% of the entire ecosystem), which might be utilized to threaten the NPM ecosystem. The top 10 of them are presented in Table 2.

Table 2: Top 10 CVEs that affect most versions

Public ID	Source Lib.	#Affected Ver.	#Affected Lib.
CVE-2019-10747	**set-value**	948,208	73,947
CVE-2019-10744	**lodash**	867,148	79,459
CVE-2018-16487	lodash	819,360	77,433
CVE-2018-3721	lodash	790,100	75,817
CVE-2018-3728	hoek	741,754	62,227
CVE-2019-1010266	lodash	712,971	70,956
CVE-2018-1000620	cryptiles	601,414	52,334
CVE-2018-20834	tar	592,691	48,356
CVE-2017-16137	debug	509,455	38,626
CVE-2016-10540	minimatch	388,126	41,423

To have a more intuitive view of vulnerability propagation, we extract vulnerable paths from these vulnerable points to corresponding root packages. Note that we ignore library versions (0.5%) which have over 1k vulnerable paths, since it takes too much time to compute all paths exhaustively. Finally, we identified **88,192,572 vulnerable paths**. On average, each vulnerable version has 3.97

vulnerable points in its dependency tree, which generate 32.53 vulnerable paths. Besides, nearly 90% of vulnerable paths are longer than 3 steps. This means that each vulnerable point in dependency trees may have multiple complex paths to affect the root packages, and these results prove that remediation on single dependency relation or even single vulnerable path may not be enough to exclude corresponding vulnerabilities completely.

Besides, it is surprising that there are still 33.33% vulnerable versions (903,569) having **one-step vulnerable paths** (refer to Figure 5b), and 12.04% of them (326,404) only have such paths. Since direct dependencies are visible to developers and maintainers as configured in package.json [23], one-step vulnerable path should be easily identified and handled if developers and maintainers are sensitive to vulnerabilities in dependencies. Moreover, we further notice that one-step vulnerable paths exist in dependency trees of 33.42% (62,022/185,598) of vulnerable latest versions, which means even for the latest versions, developers and maintainers have not paid enough attention to security in dependencies. These findings indicate a universal lack of attention on vulnerabilities from dependencies, even for vulnerabilities from direct dependencies.

As for multi-step vulnerable paths, the vulnerable paths of 49.57% of vulnerable versions (1,344,020) propagate and affect root package via only one direct dependency. Vulnerable paths of 78.94% of them (2,140,239) go through no more than 3 direct dependencies. This indicates that most vulnerable paths are centralized to propagate and affect root packages via limited direct dependencies, and it proves that controlling direct dependencies precisely may be an effective solution to cut off most vulnerable paths.

Finding-2: ① There are centrality that some influential known CVEs widely exist in the dependency trees of a significant portion of packages. ② Packages are usually affected by multiple vulnerable points, and each vulnerable point affects root packages via multiple vulnerable paths (averagely, one vulnerable points introduce 8 vulnerable paths). ③ Vulnerabilities still widely exist in direct dependencies of affected library versions (over 30%), even for the latest versions. ④ There is also centrality on vulnerable paths that most of the vulnerable paths go through limited direct dependencies, which could be utilized to cut off vulnerable paths.

5.2 RQ2: Vulnerability Propagation Evolution in Dependency Trees

Since dependency trees installed by default could change along with the release of new version of any library in the tree, it is highly possible that the status of root packages being affected by vulnerability via dependencies also changes over time. An example of dependency tree changes (DTCs) that introduces vulnerability is depicted in Figure 6. A@1.0.0 has experienced two DTCs when B and C release new versions, two vulnerable points (B@1.0.1 and D@1.1.0) are introduced into the dependency tree of A@1.0.0. Note that the dependency trees we analyze are the ones to be installed by default, instead of the outdated dependencies in runtime environment.

To investigate the evolution of vulnerability propagation, therefore, find out reasons for the widespread of vulnerability propagation and further derive actionable solutions, we resolve dependency

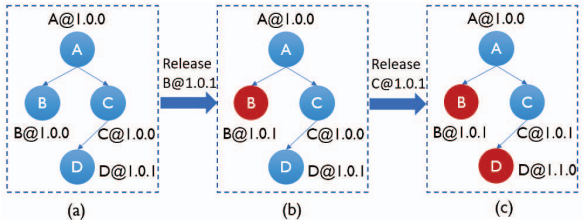
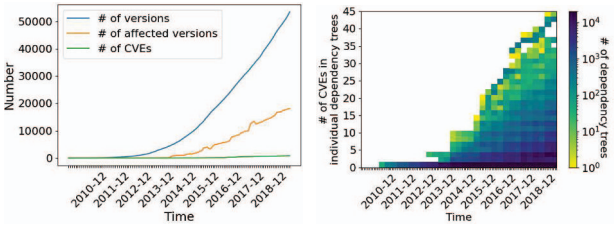


Figure 6: An example of vulnerability propagation evolution via dependency tree changes (DTCs)



(a) Evolution of library versions and CVEs **(b) Evolution of CVE density in dependency trees**

Figure 7: Evolution of known CVE propagation

trees and analyze corresponding vulnerable paths over time to investigate: 1) the overall evolution of known vulnerability propagation in dependency trees; 2) the lifecycle of vulnerabilities in dependency trees; 3) a possible solution to mitigate vulnerability affection in dependency trees based on our findings.

Due to the exponential increase of dependency trees over time, we conduct our analysis on the validation dataset in Section 4.4. Overall, 53,541 versions are selected after excluding versions with unusual dependency trees by Quartile Variation [40] and those that have dependencies with unknown release times, and we further calculate their dependency trees from release time to the latest. Finally, we obtain 10,906,781 dependency trees in total.

5.2.1 ***How does known vulnerability propagation evolve over time?*** To investigate the historical vulnerability propagation in dependency trees, we first take snapshots of dependency trees at the end of each month for all the 53,541 versions. Note that here we only measure the propagation of known vulnerabilities from their publish time.

As presented in Figure 7a, library versions and known CVEs grow rapidly, which is consistent with existing work [52, 85]. Besides, we further investigate known vulnerabilities in dependency trees to analyze the breadth of known vulnerability propagation. Specifically, we find 19.27% of these versions (10,320) were affected by known vulnerabilities at release time, while 33.86% of versions (18,127) are affected by known vulnerabilities at the latest time. This indicates that more versions are getting affected by known vulnerabilities from dependency trees over time.

Besides, we analyze the changes of the number of CVEs in individual dependency trees over time. As shown in Figure 7b, the number of CVEs in dependency trees increases rapidly over time. Specifically, we identify 69.76% of these ever affected versions (14,356) have more known vulnerabilities in dependency trees at the time of snapshot comparing to release time. In contrast, only 7.4% of them (1,524) are just the opposite. Our findings indicate that, along with

the discovery of known vulnerabilities and DTCs over time, the impact of known vulnerabilities is getting larger that known vulnerabilities are not only affecting more library versions (breadth), but also affecting each version via multiple vulnerable points (depth).

Finding-3: Known vulnerabilities are causing a larger impact across the NPM ecosystem over time. They are not only affecting more library versions but also affecting them with more vulnerable points in dependency trees.

5.2.2 ***How long do vulnerabilities live in dependency trees?*** Vulnerabilities in dependency trees can cause enormous impact not only after published. To investigate the evolution of vulnerability propagation (i.e., lifecycle of vulnerabilities) in dependency trees, we analyze how vulnerability changes in dependency trees.

Therefore, we take the time when the root package was affected by vulnerabilities as the initial time, and further define the living time of each CVE in dependency trees as the time interval between **CVE introduction** (the first DTC that brings this CVE to the dependency tree) and **CVE elimination** (the last DTC that removes this CVE from the dependency tree).

We have identified 213 CVEs from the 10 million dependency trees, and quantified 243,448 individual CVE introductions related to these CVEs. Besides, to look into the living time of these CVEs in dependency trees, we have further quantified corresponding CVE eliminations for these introduced CVEs. As a result, 60.05% of them (146,192) are removed from the corresponding dependency trees afterward, with an average living time of 371 days, while the rest of them (39.95%) still remain in the latest dependency trees. Specially, we find that 87.69% of CVE eliminations (128,190) happen before these CVEs are published, while only 7.4% of CVE introductions happen after CVE publish time.

These findings reveal that most CVEs (around 93%) are introduced to dependency trees before they are discovered, and half of the introduced CVEs (60%) in dependency trees are removed via DTCs. Besides, 88% of such CVE eliminations happen before CVE publication since most CVEs only get published after the maintainers have fixed them. However, the living time of these removed CVEs is still longer than a year, and there is still 40% of CVEs in dependency trees not removed. This is probably because it indeed requires a quite long time to handle CVEs (e.g., identifying security bugs, fixing these bugs, publishing CVEs). However, the remaining CVEs prove that there still exist CVEs that can not be removed by DTCs automatically, and lacks mature mechanisms to warn users of the CVEs in dependencies of their projects efficiently.

Finding-4: ① Most of the CVEs (93%) have already been introduced to dependency trees before they were discovered, and the fixed versions of these CVEs (87%) were also mostly released before CVE publish. ② Only 60% of CVEs in dependency trees are removed automatically by DTCs, and even so, it still takes over one year for each CVE to get removed.

5.2.3 ***Why are there still a considerable portion of CVEs not removed?*** Actually, most of the CVEs can be removed by DTCs over time, and the remained CVEs are the ones we should try to mitigate. To investigate the possible reasons of the remaining 40% of CVEs and further conclude applicable solutions, we extract all vulnerable paths introduced by these CVEs from dependency trees and classify them into **Not Removed Paths (NRP)** (i.e., those remained in latest dependency trees) and **Removed Paths (RP)** (i.e., those removed by DTCs). Therefore, 318,652 NRPs and 1,669,258 RPs have been identified. According to our previous findings, the latest versions being vulnerable are probably the common features of remained vulnerable points, and the root cause of CVE introduction and elimination is the change of dependency trees, which requires two preconditions: 1) nodes in the dependency tree have new versions released; 2) the newly released version satisfies the corresponding dependency constraint. Therefore, we think there are probably the two main reasons that may break preconditions and block CVE eliminations: Outdated Maintenance (i.e., no clean version released) and Unsuitable Dependency Constraint (i.e., the released clean version does not satisfy the dependency constraint).

We also select two typical cases to analyze how they block CVE eliminations: 1) Vulnerable Latest Version, the highest satisfying version of vulnerable point is vulnerable; (2) Fixed-Ver. D.C., the dependency constraints are fixed versions instead of ranges. We measure both of them in NRPs and RPs. The result shows that 1) Compared to RPs, NRPs contain more Fixed-Ver. D.C. (53.10% vs. 25.67%), and have more Fixed-Ver. D.C. per path (0.87 vs. 0.43). 2) Vulnerable points in 61.54% of NRPs are Vulnerable Latest Versions, while only 16.12% of RPs are in such cases. These findings prove that bad practices such as Outdated Maintenance (provider) and Unsuitable Dependency Constraint (consumer) are possibly the reasons that postpone or even block the automatic removal of CVEs by DTCs over time, and more effective and actionable instructions and solutions should be further derived against them.

Finding-5: Outdated Maintenance (provider) and Unsuitable Dependency Constraint (consumer) are the main reasons that hinder the automated vulnerability removal in dependency trees over time. More countermeasures and solutions should be carried out to avoid, monitor, or even fix these bad practices.

5.2.4 ***Example of remediation by avoiding vulnerability introduction*** *(DTReme).* Outdated Maintenance and Unsuitable Dependency Constraint are the main reasons that hinder CVEs from being automatically removed, which requires all stakeholders' efforts to exclude. However, we can remediate vulnerabilities from another direction by preventing CVE introductions. Therefore, we further propose and implement an application to provide a novel and more precise remediation (DTReme) for dependency trees, based on the DTResolver we presented in Section 4.2.

Theoretically, vulnerable paths could be introduced or removed by DTCs when new versions of libraries in the middle of paths are released. Therefore, we could use forward checking [60] and backtracking [54] to explore all possible solutions for single dependency path and avoid resolving vulnerable versions, therefore, avoiding the introduction of vulnerabilities. However, since dependency paths are not independent and could be influenced by other existing nodes in dependencies, fixing single vulnerable path may not be able to remove the vulnerability thoroughly (i.e., these remained vulnerable points in Section 5.2.3). Therefore, we combine NPM dependency resolution and strategies of forward vulnerability

Table 3: Comparison of remediation effects between npm audit fix and our remediation

# of vulnerable points in Dependency Trees	# of projects
DefDep = 0	198
DefDep = AuditDep = RemeDep >0	86 (15)
DefDep >AuditDep = RemeDep	69 (1)
DefDep >= AuditDep >RemeDep	77
DefDep >= RemeDep >AuditDep	30

checking and backtracking to resolve clean dependency trees, thus, provides remediation for entire dependency trees.

Notably, we add 1) forward vulnerability checking, when resolving versions for new coming dependencies (line 13 and 17 in Algorithm 1), only resolve clean versions for every dependency relation; 2) backward installed package tracking, once no clean version could be resolved, roll back to the resolution for parent node and find alternative versions to avoid cases like no clean version. Therefore, we can traverse all possible solutions exhausitively and find possible clean dependency trees, and a new package-lock.json file can be generated for the entire dependency tree as the remediation solution. Note that the integrity issue of lock file is handled by ssri [26].

To prove the effectiveness, we evaluate DTReme with popular JavaScript repositories from Github by comparing the remediation result with npm audit fix, the official dependency auditing tool. We first collect top 1K most stared repositories from Github [8], after excluding unsuitable projects (115 have no package.json, 239 use yarn, 27 have no dependencies, 159 have dependencies that are not published in NPM registry), we obtain 460 projects as experiment objects to compare with npm audit fix. Next, we collect 3 types of dependencies, default dependencies (DefDep), dependencies after audit fix (AuditDep), and dependencies after remediation (RemeDep), as well as vulnerable points in these dependency trees to compare remediation effects. The results are presented in Table 3.

Overall, our DTReme handles more vulnerabilities than npm audit fix. Among the 262 projects that have vulnerabilities in their dependencies, the performance of our DTReme is better than npm audit fix in 77 projects (i.e., the deep gray cell), while only 30 projects (i.e., the light gray cell) are opposite. However, these 30 cases are because that sometimes npm audit fix remediates vulnerabilities by violating direct dependency constraints [2], and our remediation follows user-defined dependency constraint strictly. Besides, among the 155 projects that DTReme and npm audit fix have the same performance, DTReme reduces more vulnerable paths introduced by these vulnerable points in 16 projects.

Finding-6: These results prove that DTReme has better performance on remediation than npm audit fix. Back tracing the vulnerable paths to the status before vulnerabilities are introduced is an effective way to exclude more vulnerabilities in dependency trees. Besides, these results also prove that there are noticeable vulnerabilities unavoidable even though we have exhausted all possible dependency paths, and mitigating such vulnerabilities requires all stockholders to be responsible for their parts and working together.

6 DISCUSSION

6.1 Lessons Learned by Our Study

We discuss the actionable solutions from different stakeholders to mitigate the severe situation.

For Package Providers. Outdated maintenance is one of the major reasons for CVEs remaining in dependency trees shown in Section 5.2.3. Thus, we conclude some tips: ① releasing patch versions soon when vulnerabilities are found, especially for those major versions that are not latest by still widely used; ② deprecating or unpublishing the vulnerable versions from the NPM registry; ③ being more responsible to maintain at least one satisfying clean version for most commonly used dependency constraints, especially when moving to the next major versions; ④ frequently checking the dependencies of their own packages with additional tools (e.g., third-party auditors), and remediating vulnerable dependencies in time, in case they propagate to transitively affect downstream users.

For Package Consumers. We recommend managing dependency trees with a compromised strategy, using dependency lock with periodically updating dependency trees to include new features and vulnerability fixes that are still compatible. Such a strategy could trade off the conflict with limited risks of containing known CVEs and reduced compatibility issues. Besides, there is a noticeable lack of attention on vulnerabilities in dependencies. A significant portion of the transitively affected packages contain one-step vulnerable paths as shown in Section 5.1.2. We strongly call on the attention of users on vulnerabilities in dependencies, especially in direct dependencies, and more analysis tools (i.e., third-party auditors) should also be applied during software development and maintenance.

For Third-party Auditors. Most existing software component analysis (SCA) tools are heavy (i.e., require real installation or lock file). Lighter static tools (e.g., DTReme) can be further included (e.g., in IDEs) and help developers examine their dependencies with much higher frequency. Besides, there are more directions based on our findings that could improve the security for users' projects. ① More fine-grained remediation. Apart from patching on vulnerable codes, currently, version-based remediations are manipulating either vulnerable points (npm audit fix) or direct dependencies only (e.g., snyk's remediation [35]), while there are still lots of CVEs in dependency trees unremediated. As presented by DTReme, simply excluding vulnerable dependencies from path level is efficient to remediate much more vulnerabilities than npm audit fix, and more recommendations, e.g., even replacing libraries with similar functionalities, could be further investigated. ② More accurate reachability analysis. Package-level detection is not accurate and could introduce false positives [56]. Although it is a difficult task [63, 67, 73], reachability analysis based on call graph [68] can precisely filter out if these vulnerable codes are really called.

6.2 Limitations and Threats to Validity

First, the vulnerabilities in dependencies may never actually affect root packages since these vulnerable functions may never be reached. This can only be further tackled by analyzing vulnerable function call paths based on dependency trees and call graphs. We leave this as our future work. Second, the mapping of CVEs and

library versions is labeled manually, which may cause data mislabeling, and the co-authors have cross-validated the data with existing CVEs to mitigate such threats. Third, we can not distinguish installations that contain missing dependencies, which could make the ground truth inaccurate, we only take packages in dependencies that are successfully installed as ground truth in validation. Fourth, we ignore versions with over 1k vulnerable paths when analyzing vulnerability propagation due to excessively high computation costs. Overall, such versions only account for 2.01%, which can only cause limited bias to our results.

7 RELATED WORK

Vulnerability Analysis via Dependency. Lauinger et al. [65] checked 133k websites and found 37% websites use at least one JavaScript library with a known vulnerability. Pfretzschner et al. [70] discussed four typical dependency-based attacks. Ohm et al [69] investigated the security attacks via malicious packages from supply chain. Prana et al [71] observed the vulnerabilities from dependencies of selected projects in Java, Python and Ruby by Veracode [10]. Alfade et al [39] measured the threats of vulnerabilities by their lifecycle. Zerouali et al. [79] reported that the presence of outdated NPM packages increases the risk of potential vulnerabilities. Gkortzis et al [57] found a strong correlation between a higher number of dependencies and vulnerabilities. Javanjafari et al [61] investigated the dependency smells that could cause negative impact in JavaScript projects. Decan et al. [52] conducted an empirical study by leveraging the direct dependencies of JavaScript libraries. Their impact analysis is conducted only with direct dependents which are upstream traced with only one step, and did not consider the dependencies as integrity to analyze the vulnerability impact via transitive dependencies. Zerouali et al. [80] investigated vulnerability impact via transitive dependencies, but they reasoned such impact via dependency reach, and they are more focusing on comparison between packages and projects on general properties, i.e., dependency level. Zimmermann et al. [85] found the individual packages could impact large parts of the NPM ecosystem, and they analyzed from the perspective of maintainer accounts that could be used to inject malicious code. However, their dependencies are also derived from dependency reachability reasoning. Most existing work only conducted dependency-based vulnerability impact analysis on the reachability of library dependencies or limited transitive dependency steps, while our study is conducted based on dependency trees with high accuracy at a large scale.

Some work focused on improving security based on dependency. Cox et al. [47] proposed a metric-based method to decide if the dependencies should be updated. Van et al. [75] proposed NodeSentry to identify the vulnerable libraries by using rules for secure integration of JavaScript libraries.

In summary, these work focused on mitigating vulnerabilities based on dependencies that are already installed, while our approach focuses on identifying vulnerabilities and vulnerable path before installation. Besides, we can also include recommendations on possible remediation for identified vulnerabilities via DTReme.

Ecosystem Analysis. In most cases, researchers analyzed the dependency relations in various languages to understand the ecosystem status and dependency evolution. Wittern et al. [77] investigated the NPM ecosystem from several aspects (e.g., library dependency, download metrics). Decan et al. [49, 50, 53] conducted comparison studies of different ecosystems, and they [48] also recommend recommend semantic versions by the wisdom of the crowd. Similarly, Kikas et al. [62] analyzed the dependency network and evolution of three ecosystems (i.e., NPM, Ruby, and Rust). Besides, some existing empirical studies further analyzed different aspects of the ecosystem from various entry points. [66] used topological data analysis to investigate the NPM ecosystem. [38, 42, 45, 64] investigated the prevalence and impact of trivial packages. [44, 51, 74, 78, 81] investigate the technical lags of adopting updates in several ecosystems. [82] compares different metrics on popularity, and [55] predicts the popularity change based on dependency supply chain. [37] compares the dependency resolution of different package managers. [46] focus on packages with the same day release. [72] investigates the problem of license violation. [41] inspects the packages that are usually co-occur dependencies. [43] investigated the slow patching process within the NPM ecosystem.

Compared with our study, most of the existing work on ecosystem analysis only focused on a limited group of study subjects and more focused on general statistics of entire ecosystems without precisely considering the accurate dependency reachability. Instead, we have conducted a large-scale study on a full set of packages in the NPM ecosystem, and have analyzed on both ecosystem and package (version) levels with accurate dependency trees considered.

8 CONCLUSION

In this paper, we carry out a large-scale empirical study on the vulnerability propagation and propagation evolution by leveraging a complete and precise DVGraph, and a novel algorithm that we firstly propose to statically and precisely resolves accurate dependency trees DVResolver at any time for each package. Our study unveils many useful findings on the NPM ecosystem. Based on it, we propose DVReme as an example to mitigate vulnerability impact, and we also highlight some implications to shed light on the severe security threats in NPM ecosystem and further invoke actionable solutions for different stakeholders to mitigate such security risks.

ACKNOWLEDGMENTS

This research was partially supported by the National Natural Science Foundation of China (Grant No. 62102284), the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRFI06-2020-0022-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2021. *BlackDuck.* https://www.blackducksoftware.com/
[2] 2021. *[BUG] "npm audit fix" replaces direct dependencies without considering semantic version compatible.* https://github.com/npm/cli/issues/2478
[3] 2021. *bundledDependencies.* https://docs.npmjs.com/files/package.json#bundleddependencies
[4] 2021. *Common Vulnerabilities and Exposures.* https://cve.mitre.org/
[5] 2021. *Contrast Security.* https://www.contrastsecurity.com/
[6] 2021. *Dependency Resolution Rules.* http://npm.github.io/npm-like-im-5/npm3/dependency-resolution.html
[7] 2021. *How MPM3 Works.* https://npm.github.io/how-npm-works-docs/npm3/how-npm3-works.html
[8] 2021. *Introducing new ways to keep your code secure.* https://github.blog/2019-05-23-introducing-new-ways-to-keep-your-code-secure
[9] 2021. *Ladash vulnerability.* https://nvd.nist.gov/vuln/detail/CVE-2018-16487
[10] 2021. *Manage Your Entire Application Security Program in a Single Platform.* https://www.veracode.com/
[11] 2021. *National Vulnerability Database (NVD).* https://nvd.nist.gov/
[12] 2021. *Neo4j.* https://neo4j.com/
[13] 2021. *NPM - Build amazing things.* https://www.npmjs.com/
[14] 2021. *npm - Libraries.io.* https://libraries.io/npm
[15] 2021. *npm-audit.* https://docs.npmjs.com/cli/v6/commands/npm-audit
[16] 2021. *NPM-install.* https://docs.npmjs.com/cli/install#algorithm
[17] 2021. *npm-install.* https://docs.npmjs.com/cli/v6/commands/npm-install
[18] 2021. *NPM INSTALL DRIVES YOU CRAZY? YARN AND CHILL!* https://geeklearning.io/npm-install-drives-you-crazy-yarn-and-chill/
[19] 2021. *NPM install got different versions according to node-semver.* https://stackoverflow.com/questions/60350847/npm-install-got-different-versions-according-to-node-semver
[20] 2021. *npm-ls.* https://docs.npmjs.com/cli/ls
[21] 2021. *npm-package-arg.* https://www.npmjs.com/package/npm-package-arg
[22] 2021. *NPM Package Metadata.* https://github.com/npm/registry/blob/master/docs/responses/package-metadata.md
[23] 2021. *npm-package.json.* https://docs.npmjs.com/files/package.json
[24] 2021. *npm-remote-ls.* https://www.npmjs.com/package/npm-remote-ls
[25] 2021. *npm-unpublish.* https://docs.npmjs.com/cli/unpublish
[26] 2021. *npm/ssri.* https://github.com/npm/ssri
[27] 2021. *NVD Data Feeds.* https://nvd.nist.gov/vuln/data-feeds/
[28] 2021. *Semantic Versioning.* https://docs.npmjs.com/about-semantic-versioning
[29] 2021. *Semantic Versioning 2.0.0.* https://semver.org/
[30] 2021. *Snyk.* https://snyk.io/
[31] 2021. *Snyk research team discovers severe prototype pollution security vulnerabilities affecting all versions of lodash.* https://snyk.io/blog/snyk-research-team-discovers-severe-prototype-pollution-security-vulnerabilities-affecting-all-versions-of-lodash/
[32] 2021. *Snyk Vulnerability Database.* https://snyk.io/vuln
[33] 2021. *Software Composition Analysis (SCA).* https://resources.whitesourcesoftware.com/blog-whitesource/sca-software-composition-analysis
[34] 2021. *SourceClear Vulnerability Database.* https://www.sourceclear.com/vulnerability-database
[35] 2021. *Upgrading package versions to fix.* https://support.snyk.io/hc/en-us/articles/360005993658-Upgrading-package-versions-to-fix
[36] 2021. *Why does node semver use "latest" in dist-tags as max version for satisfied versions?* https://stackoverflow.com/questions/60335207/
[37] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency solving is still hard, but we are getting better at it. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 547–551.
[38] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? An empirical case study on NPM. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 385–395.
[39] Mahmoud Alfadel, Diego Elias Costa, Mouafak Mokhallalati, Emad Shihab, and Bram Adams. 2020. On the Threat of npm Vulnerable Dependencies in Node. js Applications. *arXiv preprint arXiv:2009.09019* (2020).
[40] Douglas G Bonett. 2006. Confidence interval for a coefficient of quartile variation. *Computational statistics & data analysis* (2006), 2953–2957.
[41] Kyriakos C Chatzidimitriou, Michail D Papamichail, Themistoklis Diamantopoulos, Napoleon-Christos I Oikonomou, and Andreas L Symeonidis. 2019. npm Packages as Ingredients: A Recipe-based Approach.. In *ICSOFT*. 544–551.
[42] Xiaowei Chen, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Xin Xia. 2021. Helping or not helping? Why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering* 26, 2 (2021), 1–24.
[43] Bodin Chinthanet, Raula Gaikovina Kula, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2019. On The Lag of Library Vulnerability Updates: An Investigation into the Repackage and Delivery of Security Fixes Within The NPM JavaScript Ecosystem. *arXiv preprint arXiv:1907.03407* (2019).
[44] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.
[45] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2021. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering* (2021).
[46] Filipe R Cogo, Gustavo A Oliva, Cor-Paul Bezemer, and Ahmed E Hassan. 2021. An empirical study of same-day releases of popular packages in the npm ecosystem. *Empirical Software Engineering* 26, 5 (2021), 1–42.
[47] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 109–118.
[48] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* (2019).
[49] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proceedings of the 10th European Conference on Software Architecture Workshops*. ACM.
[50] A. Decan, T. Mens, and M. Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2–12. https://doi.org/10.1109/SANER.2017.7884604
[51] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 404–414.
[52] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the NPM package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.
[53] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.
[54] Rina Dechter and Daniel Frost. 1998. Backtracking Algorithms for Constraint Satisfaction Problemsâ€"a Tutorial Survey. *Information-and Computer Science Technical Report* 56 (1998).
[55] Tapajit Dey and Audris Mockus. 2018. Are software dependency supply chain metrics useful in predicting change of popularity of npm packages?. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 66–69.
[56] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 559–563. https://doi.org/10.1109/ICSME.2018.00067
[57] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.
[58] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Xiaohong Li. 2021. Detecting and Augmenting Missing Key Aspects in Vulnerability Descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2021).
[59] Hao Guo, Zhenchang Xing, Sen Chen, Xiaohong Li, Yude Bai, and Hu Zhang. 2021. Key aspects augmentation of vulnerability description based on multiple security databases. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1020–1025.
[60] Robert M Haralick and Gordon L Elliott. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence* 14, 3 (1980), 263–313.
[61] Abbas Javanjafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* (2021).
[62] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE press, 102–112.
[63] Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 83–93.
[64] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2017. On the Impact of Micro-Packages: An Empirical Study of the NPM JavaScript Ecosystem. *arXiv preprint arXiv:1709.04638* (2017).
[65] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2018. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. *arXiv preprint arXiv:1811.00918* (2018).
[66] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. 2017. Extracting insights from the topology of the JavaScript package ecosystem. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 298–307.

[67] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[68] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node. js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 29–41.

[69] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*. 1–6.

[70] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-Based Attacks on Node.Js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (Reggio Calabria, Italy) *(ARES '17)*. Association for Computing Machinery, New York, NY, USA, Article 68, 6 pages. https://doi.org/10.1145/3098954.3120928

[71] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* 26, 4 (2021), 1–34.

[72] Shi Qiu, Daniel M German, and Katsuro Inoue. 2021. Empirical Study on Dependency-related License Violation in the JavaScript Package Ecosystem. *Journal of Information Processing* 29 (2021), 296–304.

[73] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[74] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical Lag of Dependencies in Major Package Managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 228–237.

[75] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. 2019. A server-side JavaScript security architecture for secure integration of third-party libraries. *Security and Communication Networks* 2019 (2019).

[76] James Williams and Anand Dabirsiaghi. 2012. The unfortunate reality of insecure libraries. Aspect Security. *Inc., March* (2012).

[77] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 351–361.

[78] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.

[79] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the impact of outdated and vulnerable JavaScript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 619–623.

[80] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2021. On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Networks. *arXiv preprint arXiv:2106.06747* (2021).

[81] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. 2019. A formal framework for measuring technical lag in component repositories and its application to npm. *Journal of Software: Evolution and Process* 31, 8 (2019), e2157.

[82] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the diversity of software package popularity metrics: An empirical study of npm. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 589–593.

[83] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1695–1707.

[84] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated third-party library detection for Android applications: Are we there yet?. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 919–930.

[85] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.