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

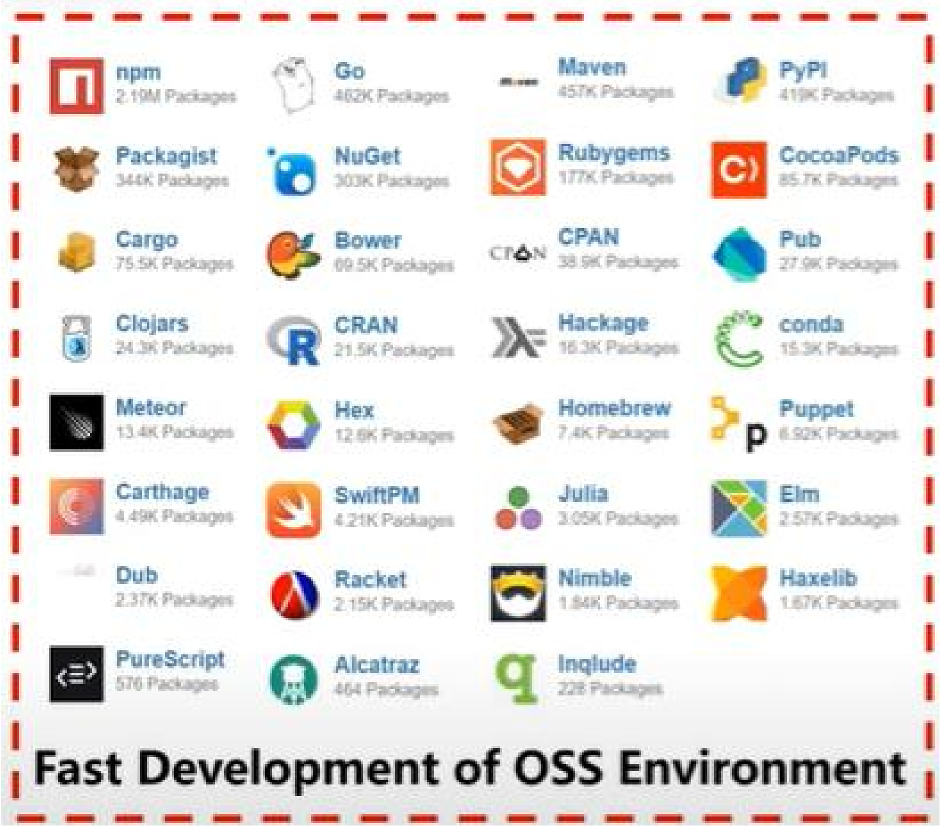
Chengwei Liu*
College of Intelligence and Computing, Tianjin University
Tianjin, China
chengwei001@e.ntu.ed

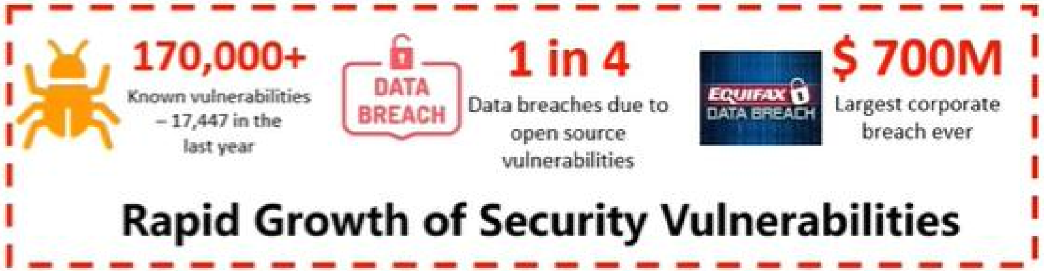
Sen Chen†
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

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

ICSE 2022

Background





The vulunerability impact could be excessively amplified by dependencies, and demystifying such impact and remediating it is urgent.

Related Work

Existing research

only considers direct dependencies [52] or

reasoning transitive dependencies based on reachability analysis [85]

which neglects the NPM-specific dependency resolution,resulting in wrongly resolved dependencies.

Existing approaches can't provide precise dependencies

only retrieve dependency trees from real installation rather than static reasoning.[30] [1]

This Work has:

1. Completeness. DVGraph covers 100% of libraries and 99.96% of versions of the metadata database
2. Accuracy. Considers NPM-specific dependency resolution rules
3. Efficiency. static
4. Dynamic updates. time dimension

[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.

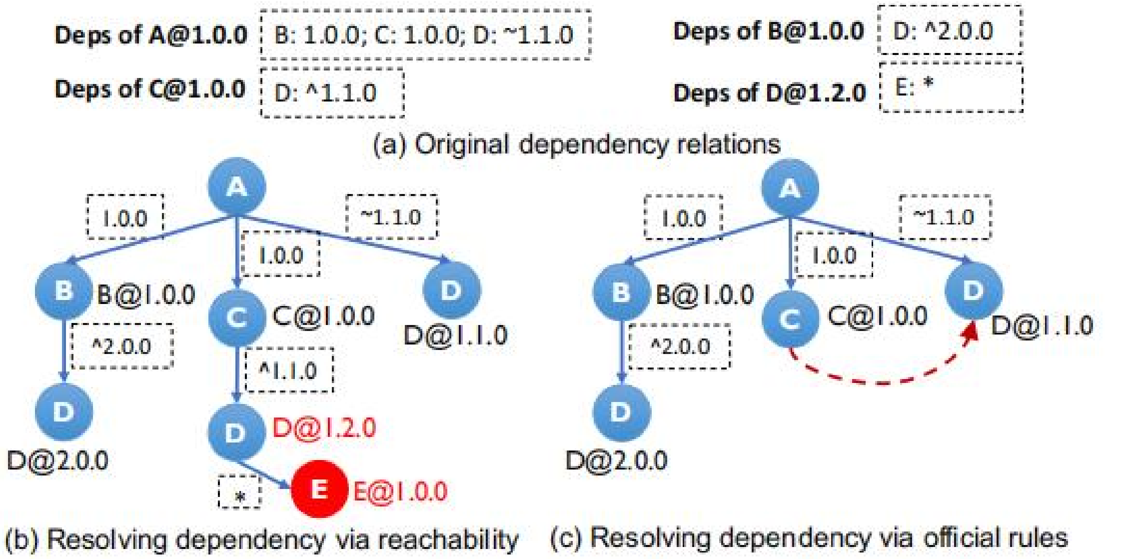
[85] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel.2019. Small world with high risks: A study of security threats in the npm ecosystem. In 28th USENIX Security Symposium (USENIX Security 19). 995–1010.

[30] 2021. Snyk. https://snyk.io/

[1] 2021. BlackDuck. https://www.blackducksoftware.com/

Motivation example

Why it is unreliable to conduct vulnerability propagation analysis via existing reachability analysis?



(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", ˆ1.1.0 represents "<2.0.0 and >=1.1.0", "*" represents any version.

Figure 2: An example of NPM dependency resolution

It is highly possible that the status of root packages being affected by vulnerability via dependencies also changes over time.

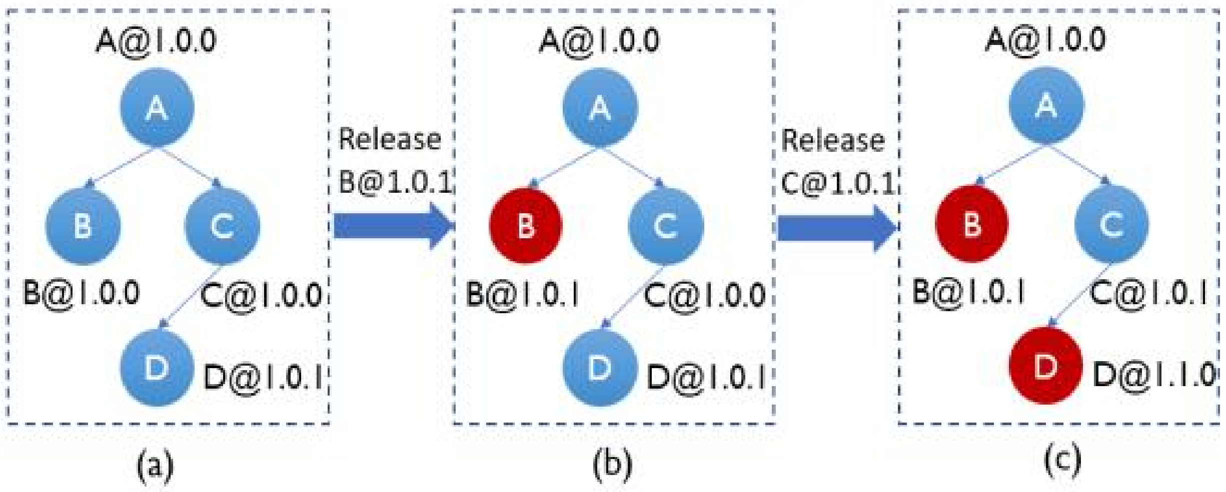


Figure 6: An example of vulnerability propagation evolution via dependency tree changes (DTCs)

Overview

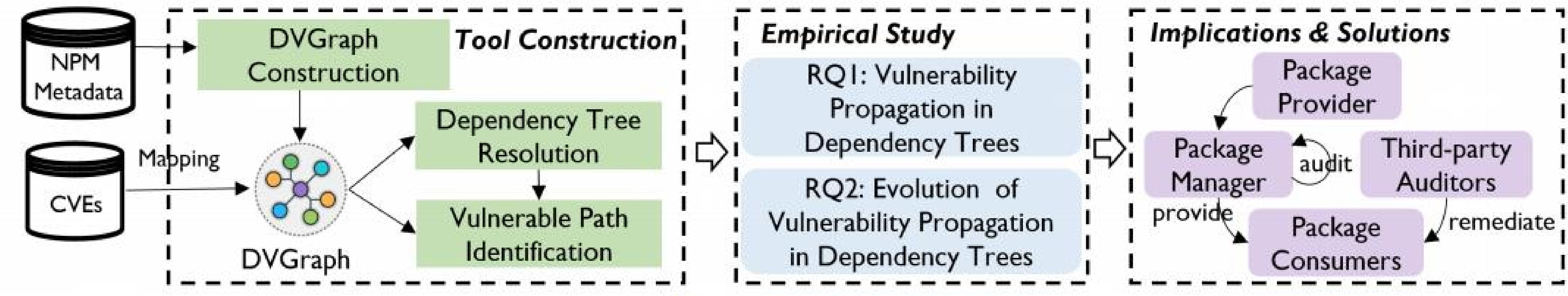


Figure 1: Overview of our work

1) via dependency constraint parser to construct a complete dependency-vulnerability knowledge graph (DVGraph) (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]

2) propose an accurate DVGraph based dependency resolution algorithm (DTResolver) to calculate dependency trees at any installation time. over 90% of resolved dependency trees being exactly the same comparing to real installation.

3) conduct an empirical study. unveil the reasons of vulnerabilities being introduced in dependency trees, as well as possible solutions.

DVGraph

- DVGraph Schema

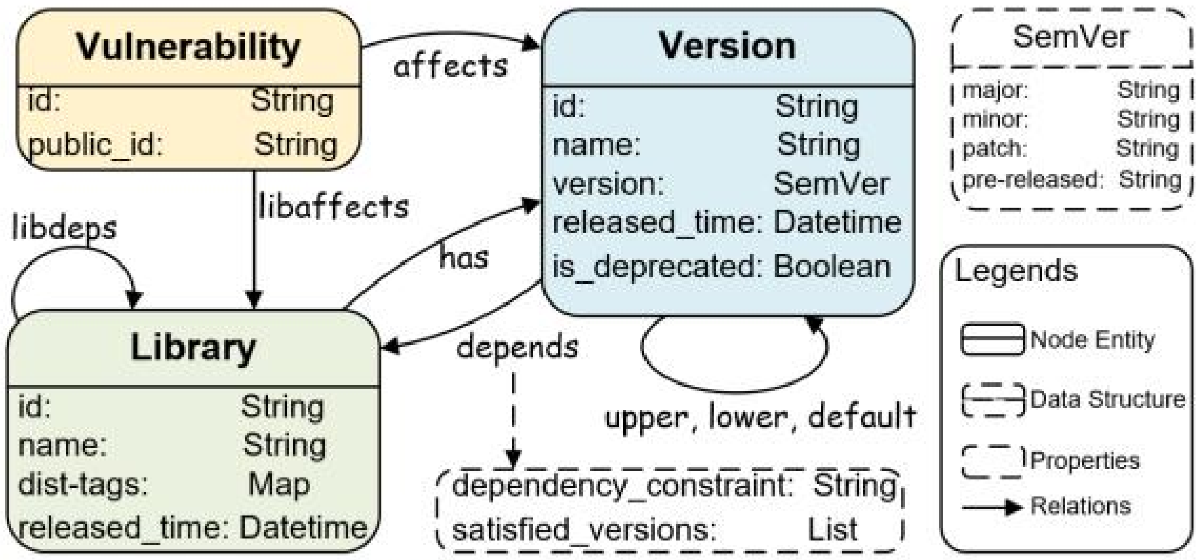


Figure 3: Schema of NPM dependency-vulnerability graph

- Graph statistics

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

Table 1: Definitions of node entities and relations

Criteria	Descriptions
Lib	Library entity contains properties (e.g., lib_id, lib_name, dist-tags).
Ver	Version entity contains properties (e.g. ver_id, released_time, is deprecated).
Vul	Vulnerability entity contains properties (e.g., vul_id and public_id).
has	$Lib_1 \rightarrow Ver_1$ presents library Lib_1 has a released version Ver_1.
upper	$Ver_1 \rightarrow Ver_2$ presents next semantically higher released version of Ver_1 is Ver_2.
lower	$Ver_2 \rightarrow Ver_1$ presents previous semantically lower released version of Ver_2 is Ver_1.
depends	$Ver_1 \rightarrow Lib_1$ presents version Ver_1 has a dependency on library Lib_1, which contains properties such as dependency_constraint and satisfied_versions.
default	$Ver_1 \rightarrow Ver_2$, assuming $Lib_1 has Ver_2$, it presents version Ver_1 *depends* on library Lib_1, and currently Ver_2 is the semantically highest released version of library Lib_1.
libdeps	$Lib_1 \rightarrow Lib_2$ presents $\exists V_L \in L_1$, V_L *depends* on L_2.
affects	**Vul** $\rightarrow$ **V**, presents that Vulnerability **Vul** directly affects version **V**.
libaffects	**Vul** $\rightarrow$ **L**, presents that $\exists$ $V_L \in L_1$, **Vul** *affects* V_L.

DVGraph

- DVGraph ConstruCtion Piplines

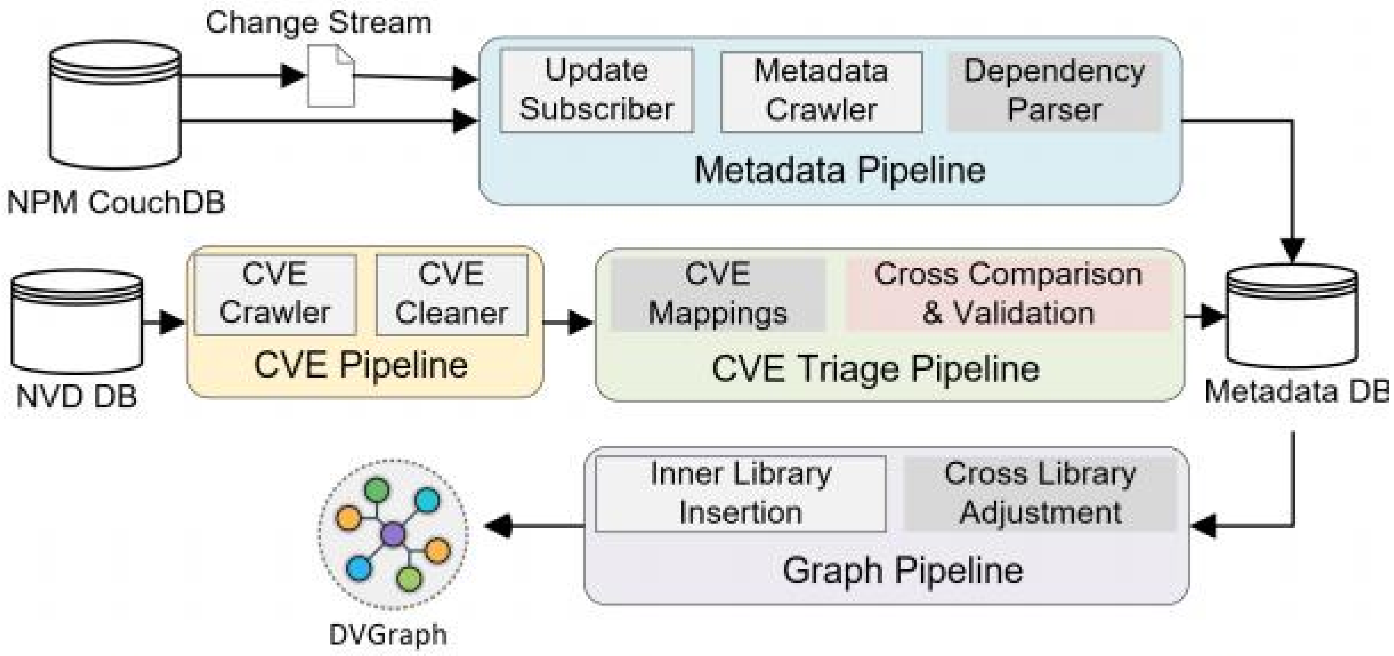


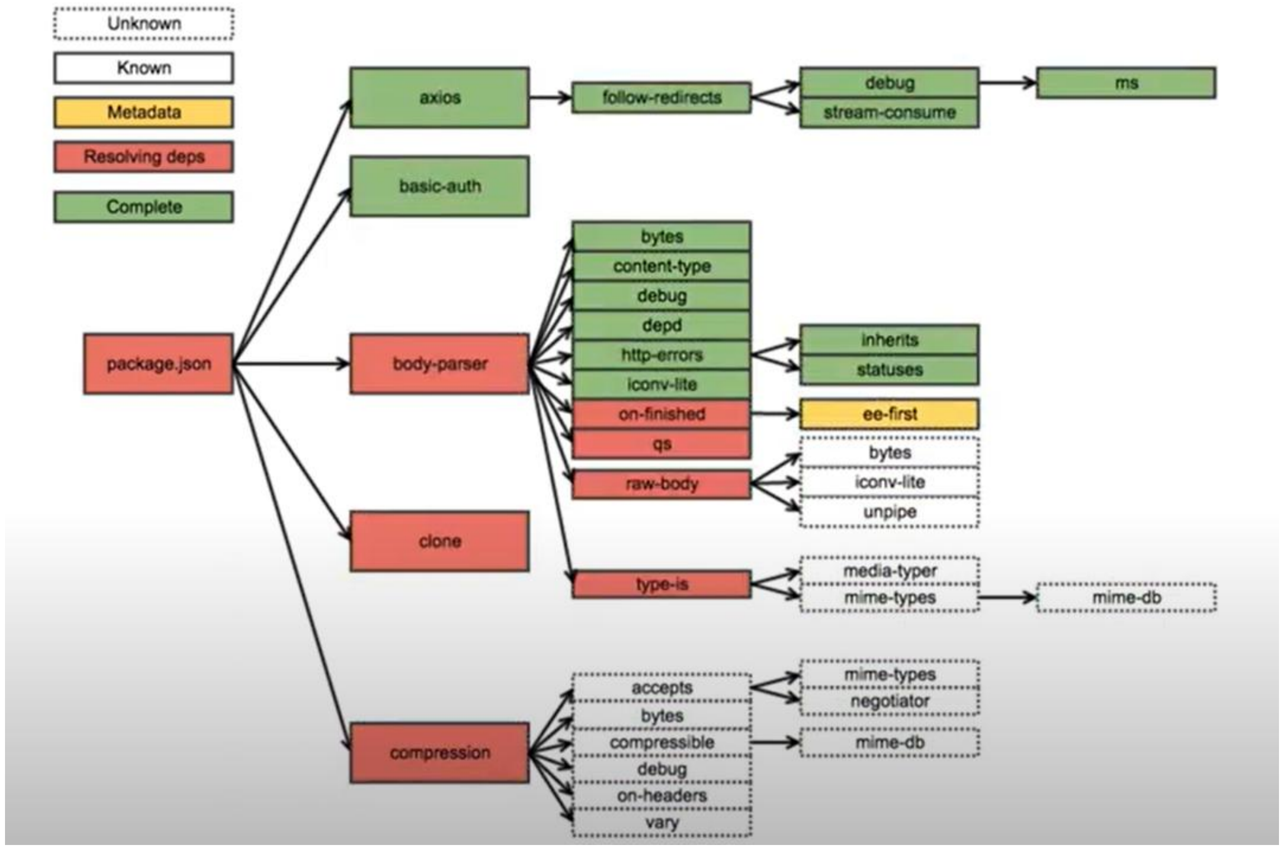
Figure 4: Automated data processing framework

Main Challenges:

1. Dependency Parser
2. CVE Mappings
3. DVGraph Updates

DTResolver

Dependency Tree Resolution



Key Rules

1. Recursively resolving dependencies by BFS
2. Allocating folders(Logical Tree and Physical Tree)
3. Preference on non-deprecated versions
4. extend time dimension by adding filters on release time
5. ...

DTResolve are also extended to resolve dependency trees that should be installed at any given installtion time.

DTResolver

Algorithm

Algorithm 1: Dependency Tree Resolution

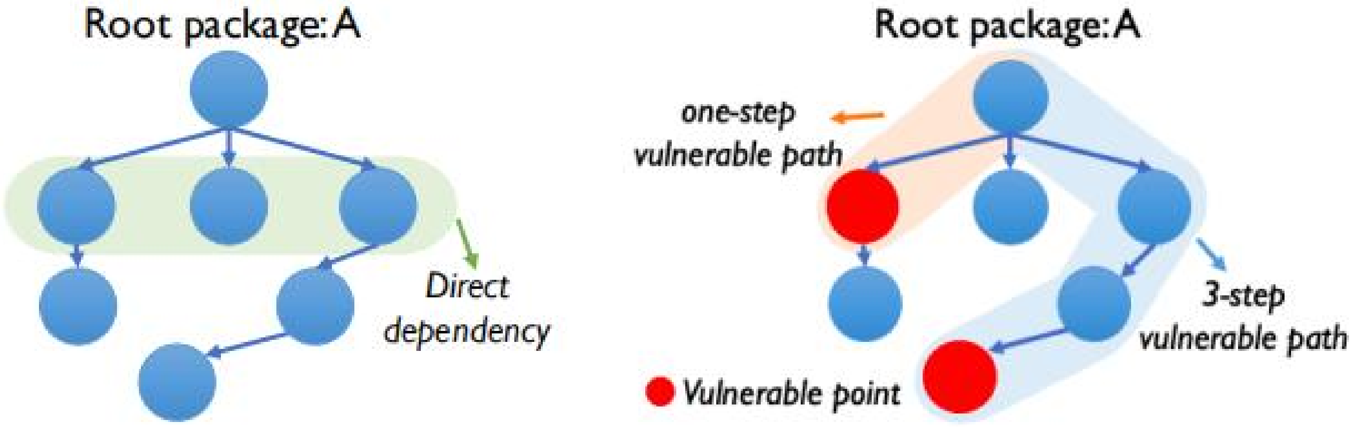
```
Input: G: DVGraph, r: given root package,                // t: given time
Output: DT_r: Resolved dependency tree of r
Dir ← new InstallDirectory()
root_path ← ∅, Q ← ∅, Deps ← ∅
Dir.install(r, root_path)
Q.push(r)
// 1. Traverse all resolved dependency nodes by BFS, and simulate
   real installation to create folders for packages
while Q ≠ ∅ do
    lv ← Q.pop()
    deps ← {e ∈ G : e_src = lv ∧ e.type = depends}
    foreach depend ∈ deps do
        vers ← depend.satisfied_versions
        deplib ← depend_dst
        if ∃ v_i. v_i ∈ Dir ∩ vers ∧ v_i.dir_path ⊑ lv.dir_path then
            r ← (CREATE lv -dep-> v_i)
            Deps.push(r)
        else
            selected ← v_i.v_i ∈ vers ∧ (∀v_j.v_j ∈ vers ∧ i ≠ j ∧ v_i > v_j)
              // ∧v_i.released_time < t

            if Dir ∩ vers = ∅ then
                install_path ← root_path
            else
                foreach subpath ⊑ lv.dir_path do
                    if ¬∃n.n ∈ subpath ∧ (deplib − has → n) then
                        install_path ← subpath
                        break
            Dir.install(selected, install_path)
            r ← (CREATE lv -dep-> selected)
            Deps.push(r)
            Q.push(selected)
// 2. Recover a dependency tree from install directory and CREATED
   Deps relations
Ver_r ← {lv : lv ∈ Dir}
Dep_r ← Deps
DT_root ← <Ver_r, Dep_r>
return DT_r
```

DTResolver

Vulnerable Path Identification

a vulnerable path extractor by reverse Depth First Search (DFS)



(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)

Validation

Evaluation of DTResolver

Table 2: Library selection criteria for graph valiadation

Criteria	Descriptions	#Instances
Most Fork	most forked JavaScript projects from GitHub	Top 2K Libraries
Most Star	JavaScript projects that have the most stars from GitHub	Top 2K Libraries
Most Downloaded in the past	packages that have most downloads in the past	Top 2K Libraries
Most Downloaded in the last three years	packages that have most downloads from 2017 to 2019	Top 2K Libraries
Most Downloaded in the last year	packages that have most downloads in 2019	Top 2K Libraries
Most Dependencies Libraries	libraries that have the most direct dependencies	Top 2K Libraries
Most Dependents Libraries	libraries that have been mostly depended on	Top 2K Libraries
Most Dependencies Versions	versions that have the most direct dependencies	Top 20K Versions
Most Dependents Versions	versions that have been mostly depended on	Top 20K Versions

103,609 versions(almost 1% of the entire NPM ecosystem) from 15673 libraries are sorted out.

90.58% of Graph Trees are exactly the same with Install Tree.
In comparison, only 53.33% of Rmote Trees are same with Install Tree.[24] (**npm-remote-ls**)

There are 2 main reasons That cause the mismatch of differences between **InstallTree** and **GraphTree**.
1、 Installtion may not be complete
2、 Dependency tree from **npm ls** are deduped

[24] 2021. npm-remote-ls. https://www.npmjs.com/package/npm-remote-ls

Validation

Evaluation of Vulnerability Detection and Vulnerable Path Identification.

31,913 library versions from our test set contains at least one vulnerable dependency， 208,129 vulnerable points in total.

DTResolver and **npm-remote-ls** have high coverage on these identified vulnerable points (98.1% v.s. 97.7%)

324,718 individual vulnerable paths are derived from these vulnerable points

300,691 of them are identified by **DTResolver (92.60%)**,but only 254,298 vulnerable paths of them are identified by **npm-remote-ls (78.31%)**

Empirical Study

RQ1: (Vulnerability Propagation via Dependency Trees)

(Dependency Trees of all 10M library versions)

RQ1.1 How many packages are affected by existing known vulnerabilities in the NPM ecosystem?
RQ1.2 How do vulnerabilities propagate to affect root packages via dependency tree?

RQ2: (Vulnerability Propagation Evolution in Dependency Trees)

(Dependency Trees Changes(DTCs) from release to current for 50K library versions from validation set,10.9 dep trees in total)

RQ2.1 How does known vulnerability propagation evolve over time?
RQ2.2 How long do vulnerabilities live in dependency trees?
RQ2.3 Why are there still a considerable portion of CVEs not removed?
RQ2.4 Example of remediation by avoiding vulnerability introduction (DTReme).

Empirical Study

RQ1: (Vulnerability Propagation via Dependency Trees)

RQ1.1 How many packages are affected by existing known vulnerabilities in the NPM ecosystem?

1、 Vulnerabilities are widely existing in dependencies of NPM packages as statically proved

- one-quarter versions of 19.96% libraries across the ecosystem.
- the latests versions of 16% libraries.

RQ1.2 How do vulnerabilities propagate to affect root packages via dependency tree?

2、 vulnerabilities from direct dependencies are widely neglected (over 30% affected library versions)

most of the vulnerable paths go through limited direct dependencies, which could be utilized to cut off vulnerable paths.

3、 Averagely, one vulnerable points introduce 8 vulnerable paths

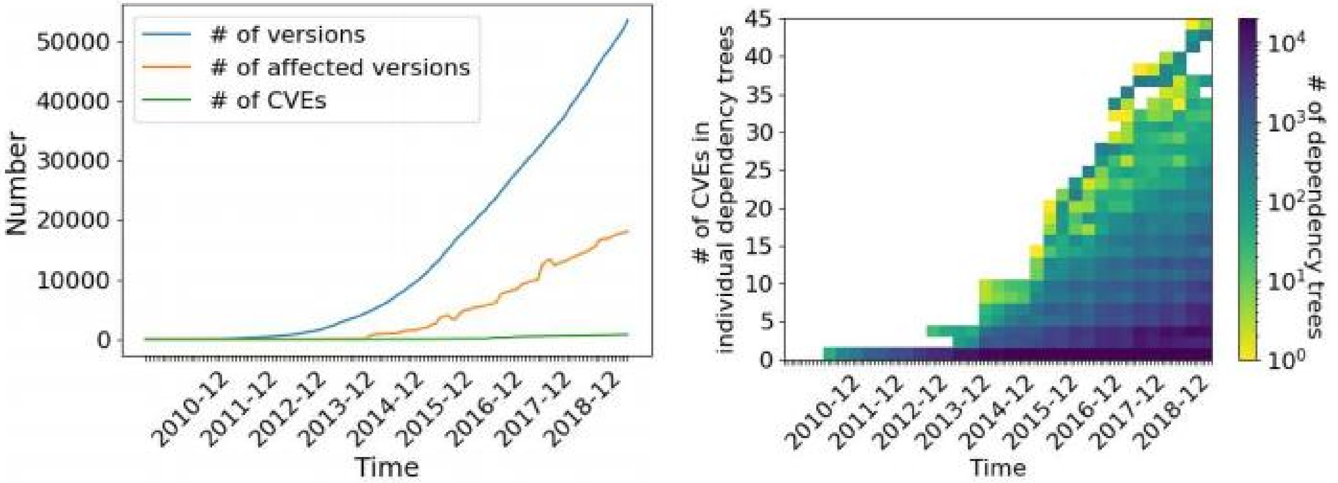
Empirical Study

RQ2: (Vulnerability Propagation Evolution in Dependency Trees)

RQ2.1 How does known vulnerability propagation evolve over time?
4、 Known vulnerabilities are causing a larger impact across the NPM ecosystem over time.

RQ2.2 How long do vulnerabilities live in dependency trees?
5、 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.

6、 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.



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

Figure 7: Evolution of known CVE propagation

Empirical Study

RQ2: (Vulnerability Propagation Evolution in Dependency Trees)

RQ2.3 Why are there still a considerable portion of CVEs not removed?

7、 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.

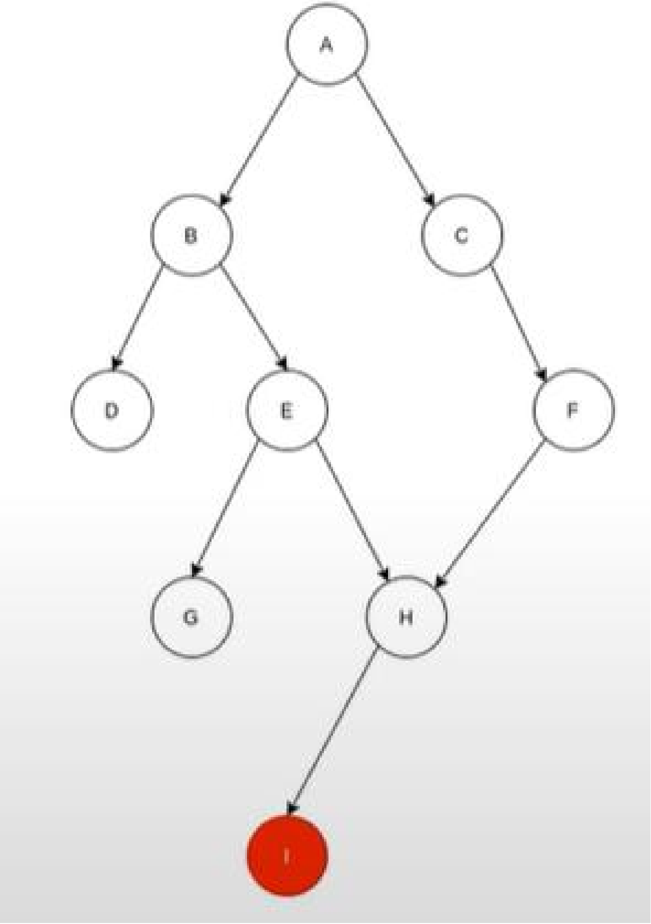
8、 Outdated Maintenance (provider) and Unsuitable Dependency Constraint (consumer) are the main reasons that hinder the automated vulnerability removal in dependency trees over time.

RQ2.4 Example of remediation by avoiding vulnerability introduction (DTReme).

9、 Considerable user projects contain unavoidable vulunerabilities even though we have exhausted all possible dependency trees(ref. DTReme results).

DTReme

Dependency Tree Remediation



Existing remediation:

For vulnerable node I in the dependency tree, remediate I by upgrading or downgrading B and C to avoid introducing vulnerable I.

DTReme

Generally,we exhaustively iterate possible alternatives for each vulnerable paths by:
(1) UpDown: Forward vulnerability checking
(2) BottomUp:Backward installed package tracking
until resolve to clean tree

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

Conclusion

1、 Consider time dimension
2、 As our next work’s Related work.

THANKS & QUESTIONS

Paper: https://arxiv.org/abs/2201.03981
Websites: https://sites.google.com/view/npm-vulnerability-study/

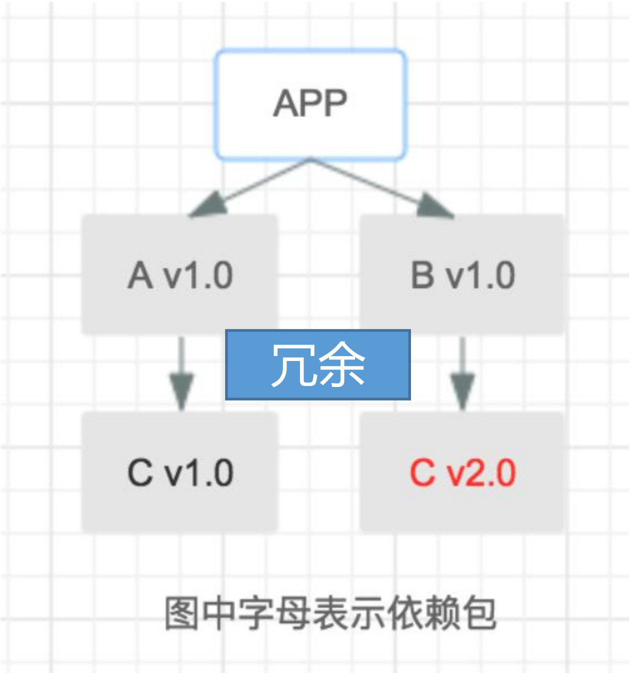
Presented by LinLi

附录 Npm模块安装机制

依赖树表面的逻辑结构与依赖树真实的物理结构不同

tree -d命令以树状图的方式列出一个项目下所有依赖的物理结构
npm ls命令以树状图的方式列出一个项目下所有依赖的逻辑结构

npm2下的模块安装机制 npm2安装多级的依赖模块采用嵌套的安装方式



附录 Npm模块安装机制

npm3下的模块安装机制：
1.在安装某个二级模块时，若发现第一层级还没有相同名称的模块，便把这第二层级的模块放在第一层级
2.在安装某个二级模块时，若发现第一层级有相同名称，相同版本的模块，便直接复用那个模块
3.在安装某个二级模块时，若发现第一层级有相同名称，但版本不同的模块，便只能嵌套在自身的父模块下方