NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

浅析NPM生态系统中的开源供应链安全

报告人：刘承威

2022年7月7日

Cyber Security Lab, School of Computer Science and Engineering, Nanyang Technological

个人简介

Chengwei Liu (刘承威)
Ph.D. Candidate, superivsed by Prof.Liu Yang (刘杨)
Nanyang Technological University, Singapore
School of Computer Science and Engineering
Cyber Security Lab
50 Nanyang Ave, #Block N4 #N4-B2C-06, Singapore 639798
chengwei001@e.ntu.edu.sg

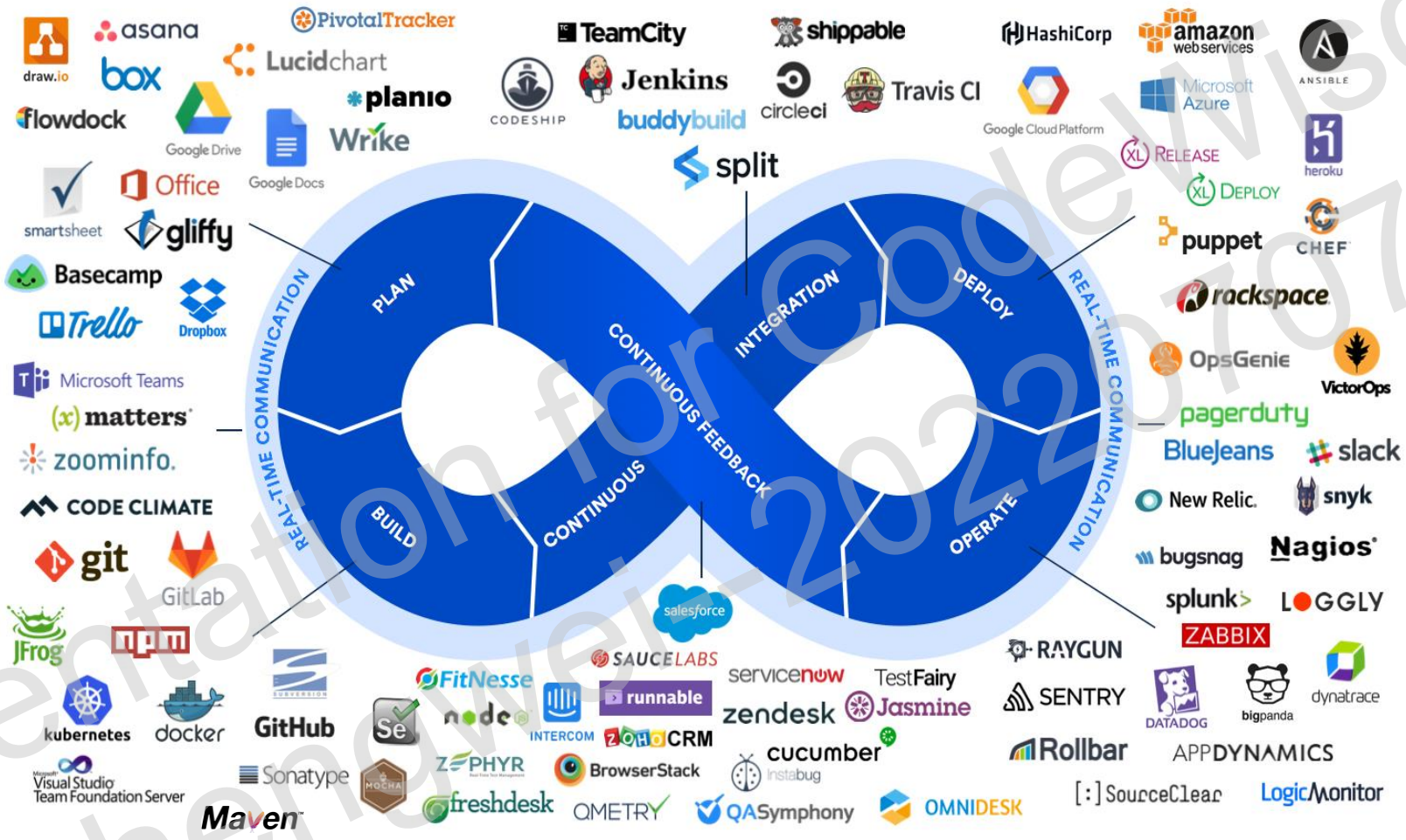
研究方向：软件安全分析，程序分析，开源软件安全，软件供应链安全，开源治理等

教育/工作经历：

- 2022.04至今---南洋理工大学网络安全实验室研究助理 （Research Associate）
- 2019.08至今---南洋理工大学 博士生
- 2012.09-2019.04 南京航空航天大学本科、研究生，师从杨志斌副教授

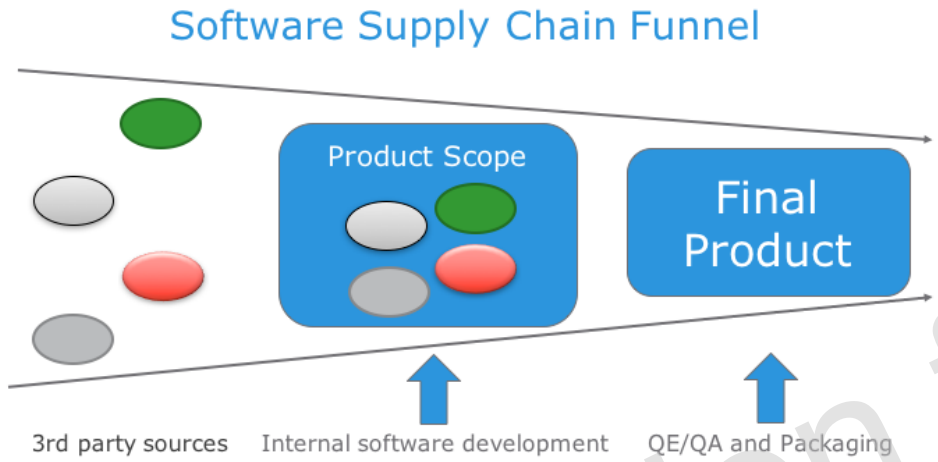
个人主页： https://lcwj3.github.io/

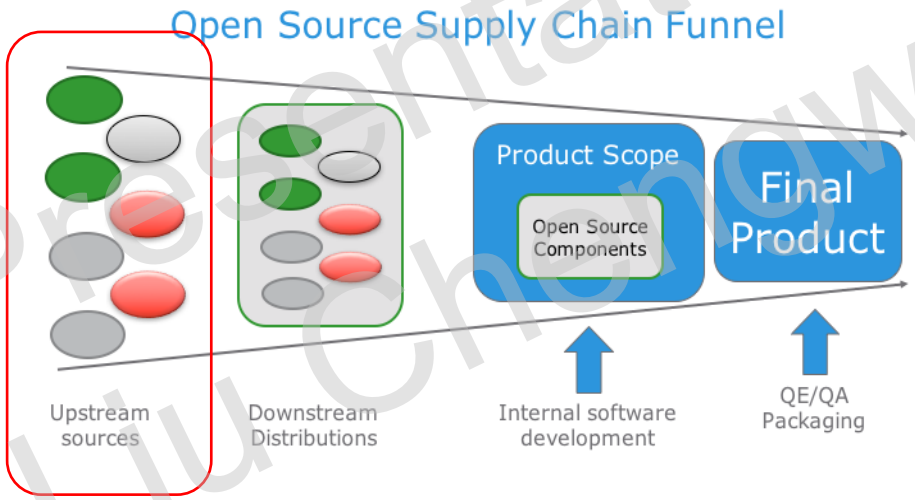
软件供应链



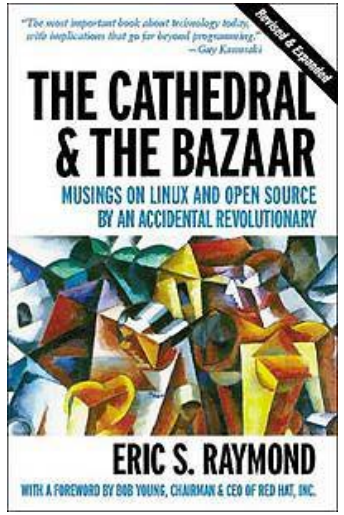
软件供应链指一个通过一级或多级软件设计、开发阶段编写软件，并通过软件交付渠道将软件从软件供应商送往软件用户的系统。

软件供应链 vs 开源供应链





传统软件供应链：



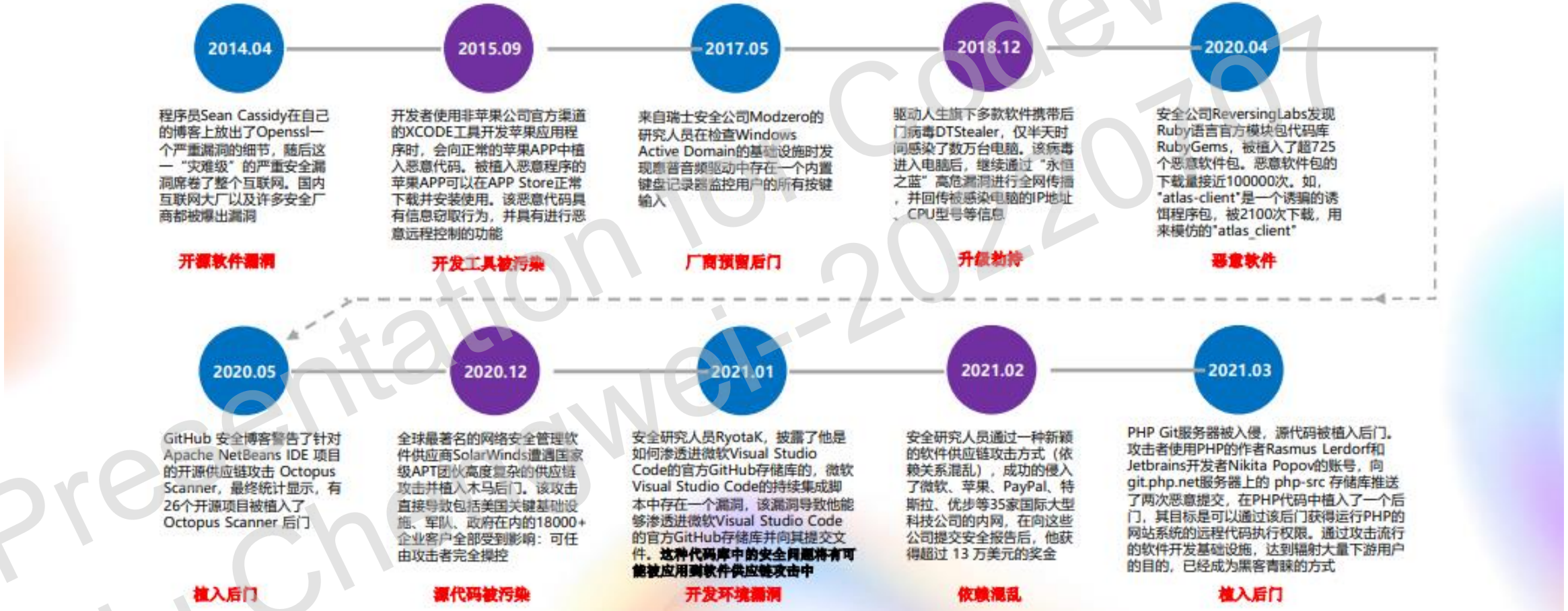
- 自顶向下“大教堂式”开发，预先审计
- 第三方依赖数量有限，相对可信
- 依赖关系相对简单
- 公开漏洞少
- 大多数闭源

如今开源供应链：

- 相对松散的“集市”式开发，相对无序，迭代迅速
- 第三方依赖数量剧增
- 依赖关系混乱复杂
- 大量公开漏洞
- 开源成为主流

开源供应链安全

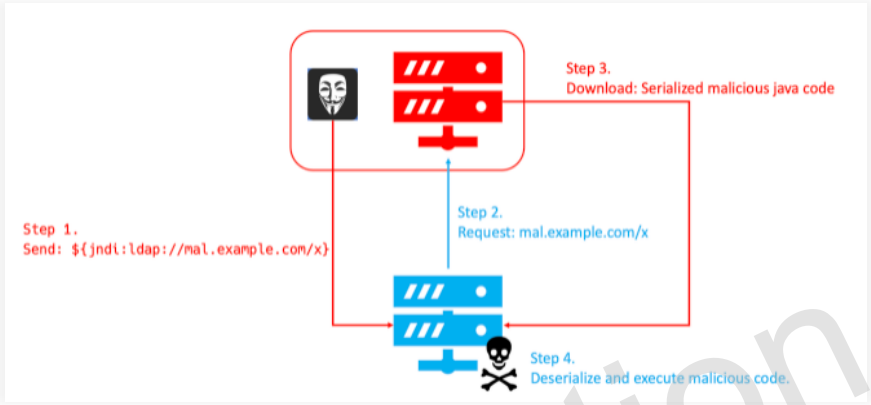
软件供应链攻击事件频发



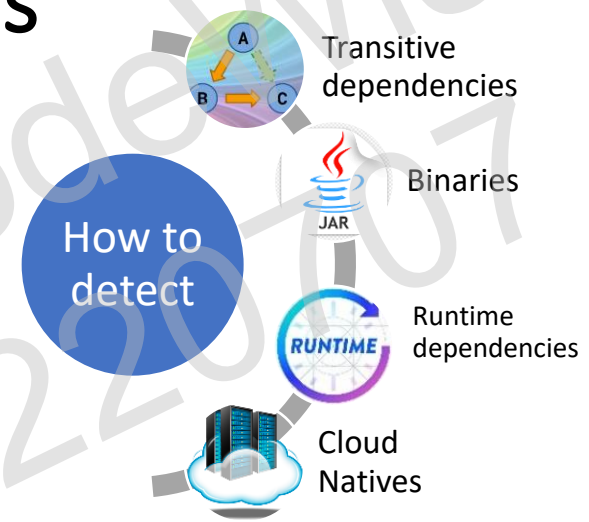
开源供应链安全

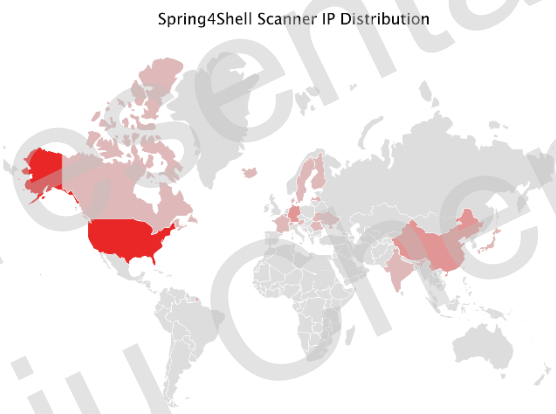
Apache LOG4J™ spring

OSS vulnerabilities emerge in big waves



Example of attack based on log4shell vulnerability (CVE-2021-44228)





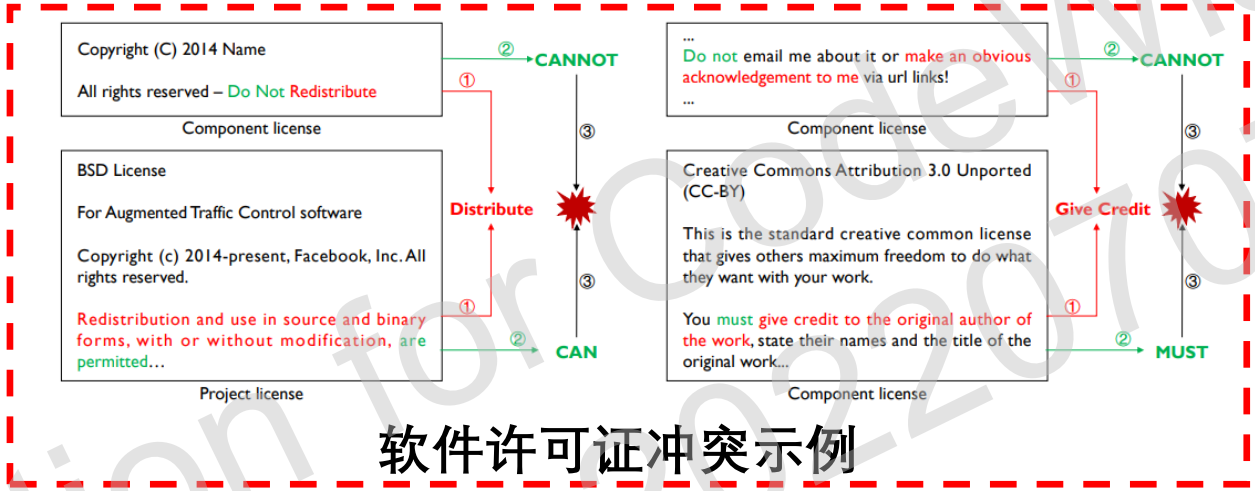
Part of IP location distribution of scanning and utilizing Spring4Shell vulnerability

Hackers utilize vulnerabilities, e.g., Log4Shell, Spring4Shell to deploy malicious software

开源供应链安全—安全漏洞之外

软件许可证合规问题

- **400+** 常用软件许可证，
- 各类**用户自定义**许可证
- 基于依赖的**许可证传染性**
- 愈演愈烈的**开源社区与云计算**巨头冲突



软件许可证冲突示例

开源治理问题

- 高危风险组件项目识别
- 组件项目质量监控管理
- 开源生态脆弱点分析
- 恶意组件/维护者风险分析
- 供应链“投毒”
- 非代码知识管理

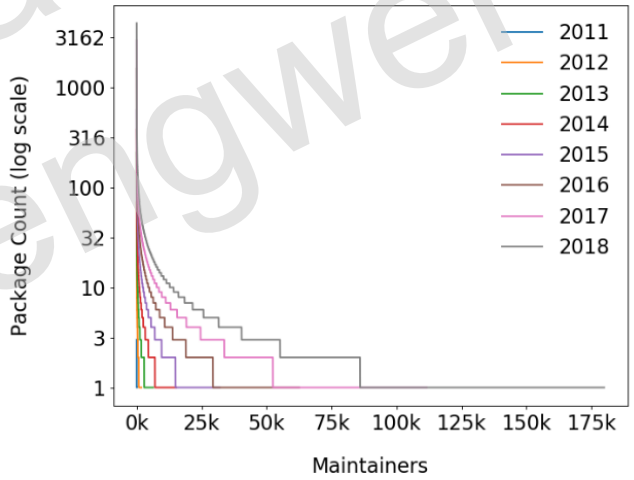


Figure 6: Evolution of maintainers sorted by package count

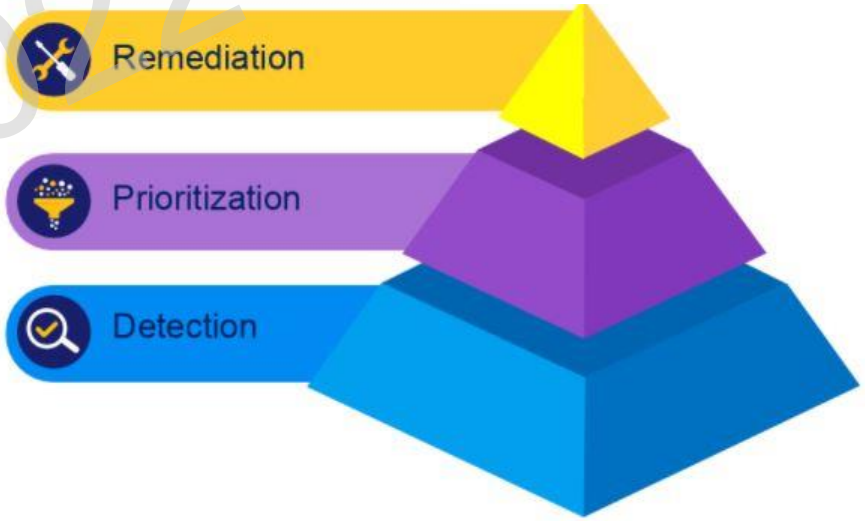
报告目录

NPM 生态系统中的开源供应链安全研究

- NPM 生态中安全漏洞影响传播及其演变分析
- 基于程序静态调用图的NPM漏洞影响力分析
- 关于开源治理的一些思考及尝试

Software Composition Analysis

SCANTIST



C O N T E N T S

1. NPM 生态中安全漏洞影响传播及其演变分析

Background

- npm 2.19M Packages
- Go 462K Packages
- Maven 457K Packages
- PyPI 419K Packages
- Packagist 344K Packages
- NuGet 303K Packages
- Rubygems 177K Packages
- CocoaPods 85.7K Packages
- Cargo 75.5K Packages
- Bower 69.5K Packages
- CPAN 38.9K Packages
- Pub 27.9K Packages
- Clojars 24.3K Packages
- CRAN 21.5K Packages
- Hackage 16.3K Packages
- conda 15.3K Packages
- Meteor 13.4K Packages
- Hex 12.6K Packages
- Homebrew 7.4K Packages
- Puppet 6.92K Packages
- Carthage 4.49K Packages
- SwiftPM 4.21K Packages
- Julia 3.05K Packages
- Elm 2.57K Packages
- Dub 2.37K Packages
- Racket 2.15K Packages
- Nimble 1.84K Packages
- Haxelib 1.67K Packages
- PureScript 576 Packages
- Alcatraz 464 Packages
- Inqlude 228 Packages

Fast Development of OSS Environment

160,000+ Known vulnerabilities – 17,447 in the last year

1 in 4 Data breaches due to open source vulnerabilities

$ 700M Largest corporate breach ever

Rapid Growth of Security Vulnerabilities

86.55 the average dependency tree size of NPM packages

78% Vulnerabilities found in transitive dependencies

8.2 Vulnerable dependency paths introduced per vul dependency in NPM

Complex Dependency Relations

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

PRELIMINARIES

SEMANTIC VERSION

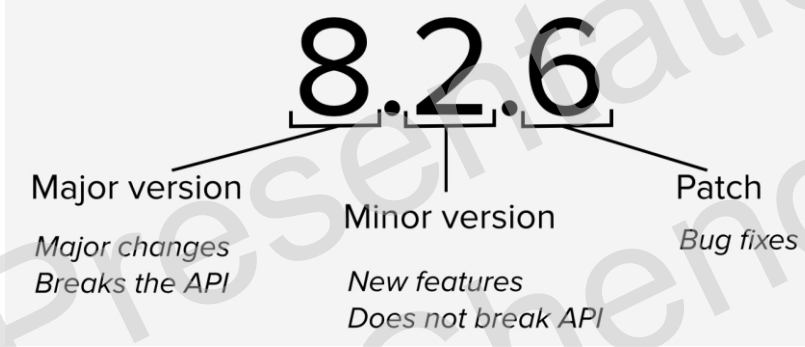
Dependency Constraints (in package.json)

Semantic Version (SemVer)

- Semantic Version is NOT STRICT RULES, and NPM has their own specific rules. (Restrictive, Compliant, Permissive)

Apart from SemVer, there are far more ways to define dependency, i.e**., tags, urls, file paths, etc.**

{major}.{minor}.{patch}-{tag}+{buildmetadata}



1.0.1 < 1.1.0-alpha.1 < 1.1.0

Examples of version constraints and their corresponding version range.

Constr.	Cargo	npm	Packagist	Rubygems
=1.0.0	[1.0.0]	[1.0.0]	[1.0.0]	[1.0.0]
1.0.0	[1.0.0, 2.0.0[	[1.0.0]	[1.0.0]	[1.0.0]
1.0	[1.0.0, 2.0.0[	[1.0.0, 1.1.0[	[1.0.0]	[1.0.0]
1	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	[1.0.0]	[1.0.0]
~1.2.3	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[
~1.2	[1.2.0, 1.3.0[	[1.2.0, 1.3.0[	[1.2.0, 2.0.0[	[1.2.0, 2.0.0[
~1	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	N/A
^1.2.3	[1.2.3, 2.0.0[	[1.2.3, 2.0.0[	[1.2.3, 2.0.0[	N/A
>1.2.3	]1.2.3, +∞[	]1.2.3, +∞[	]1.2.3, +∞[	]1.2.3, +∞[
~0.1.2	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[
^0.1.2	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	N/A

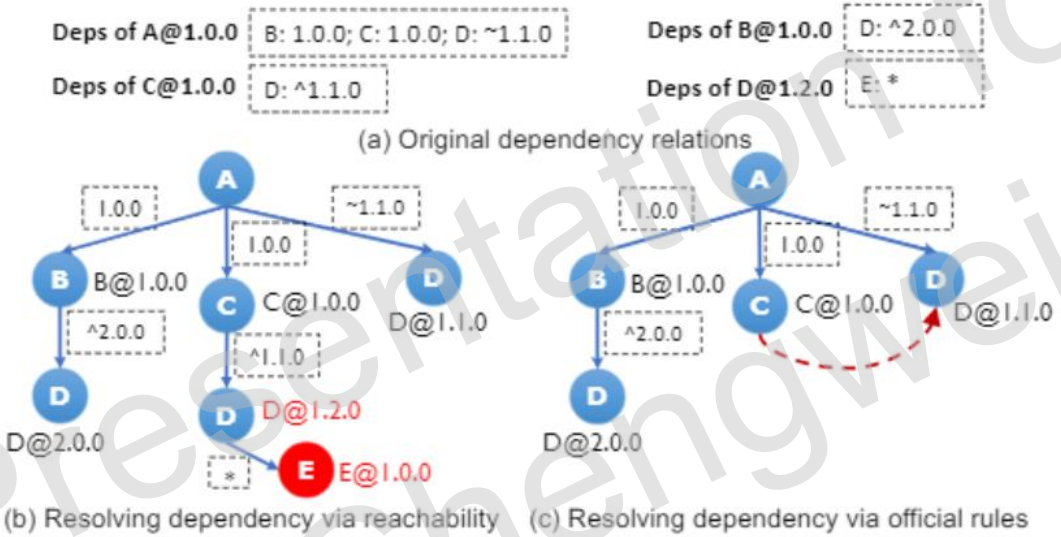
Motivation

Existing research

only considers direct dependencies or

reasoning transitive dependencies based on reachability analysis

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



(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

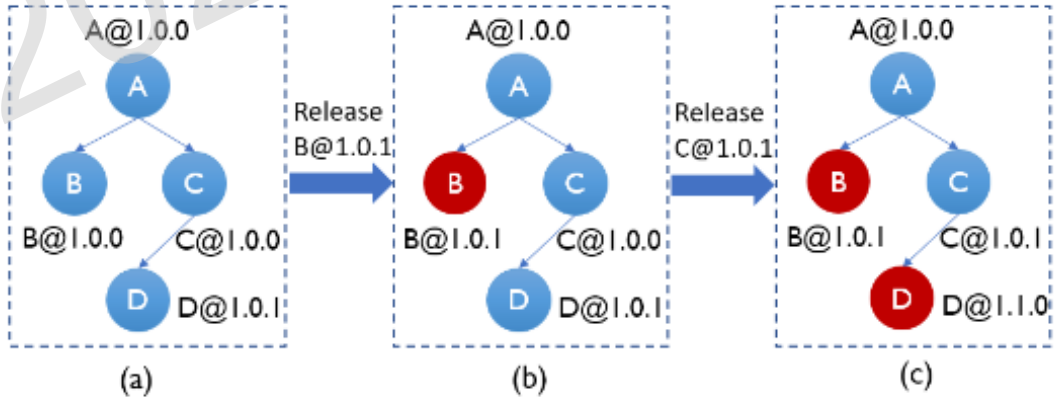


Figure 6: An example of vulnerability propagation evolution via dependency tree changes (DTCs)

Challenges

1. **Completeness**

   NPM Library and Known Vulnerability Collecting

   Dependency Constraint Resolving (Handling urls, tags, etc)

2. **Accuracy**

   Dependency Resolution

   existing work resolve dependency by reachability reasoning, which ignores NPM specific dependency resolution rules.

3. **Efficiency**

   Existing SCA requires installation

   Installation could fail due to requirement miss (i.e., OS version, NPM version, OS related component miss, etc.)

4. **Remediation**

   No precise remediation. Existing remediation only remediate on direct dependencies

Overview

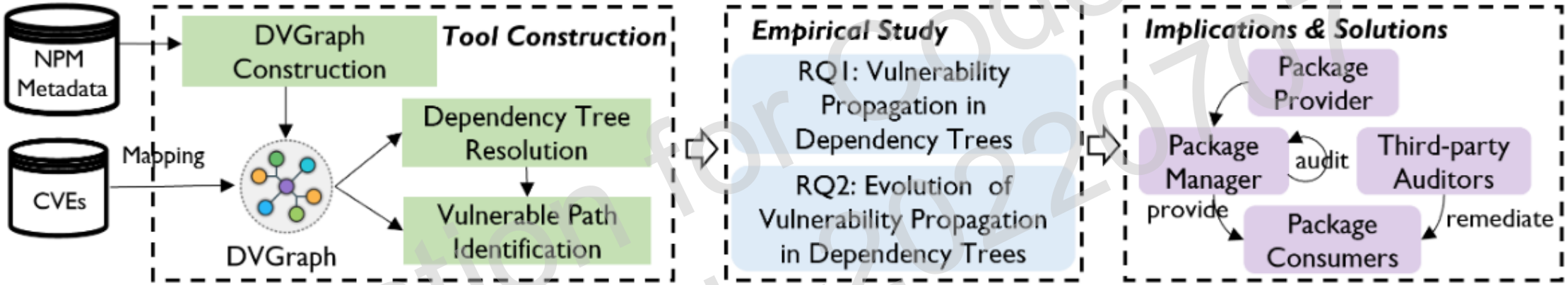


Figure 1: Overview of our work

Chengwei Liu, Sen Chen, et al. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. ICSE2022.

DVGraph

DVGraph Schema

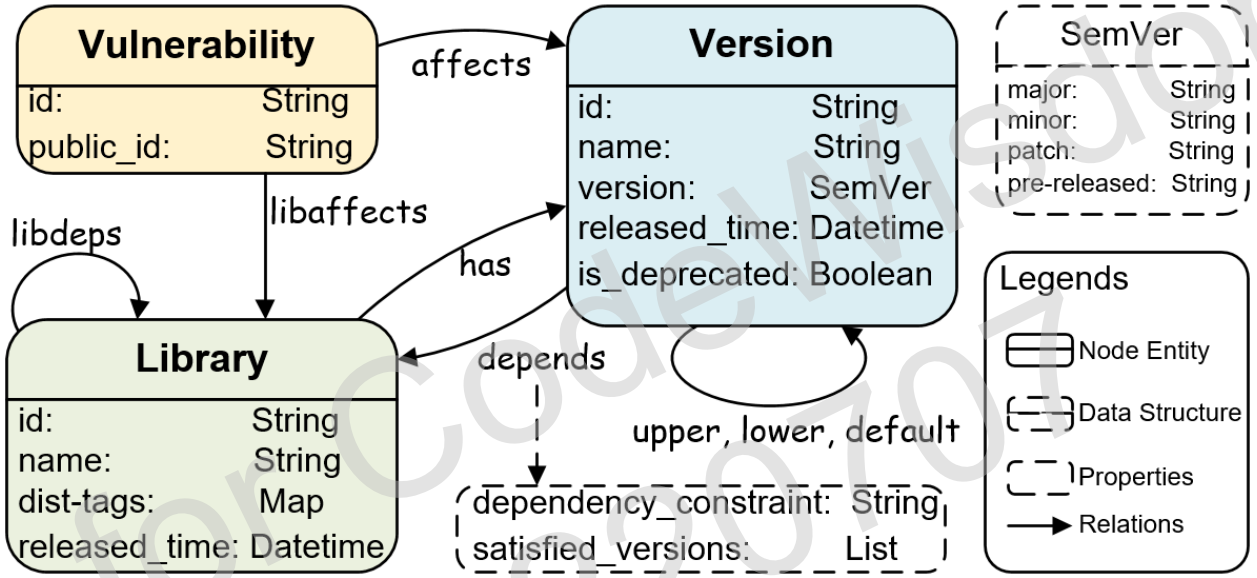


Table 1: Definitions of node entities and relations

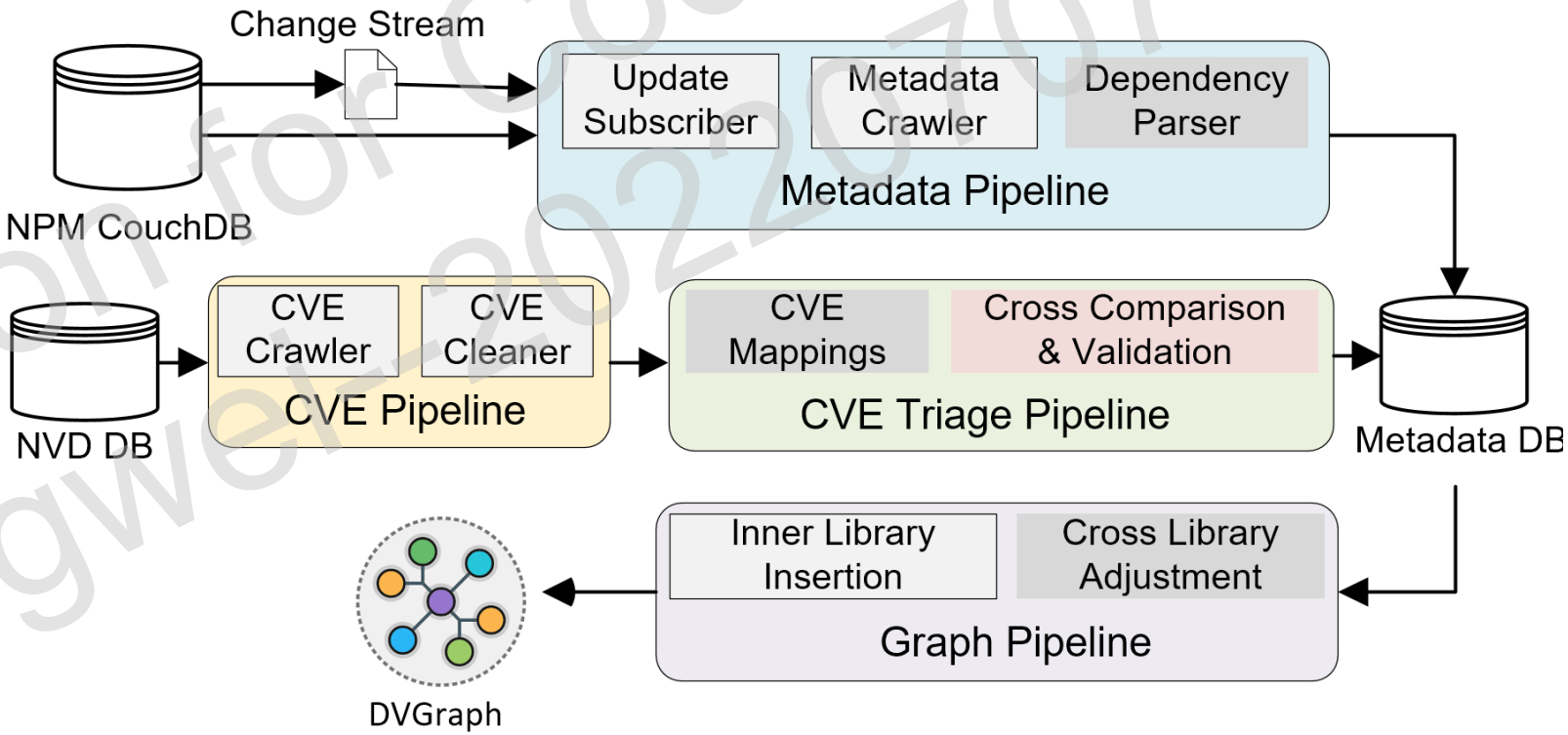
Criteria	Descriptions
Lib	Library entity contains properties (e.g., lib_id, lib_name, dist-tags).
Ver	Version entity contains properties (e.g. ver_id, released_time, is deprecated).
Vul	Vulnerability entity contains properties (e.g., vul_id and public_id).
has	$Lib_1 \rightarrow Ver_1$ presents library 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 & Maintenance

Main Challenges

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



DVGraph

DVGraph Statistics & Validation

Table 2.1: Graph statistics

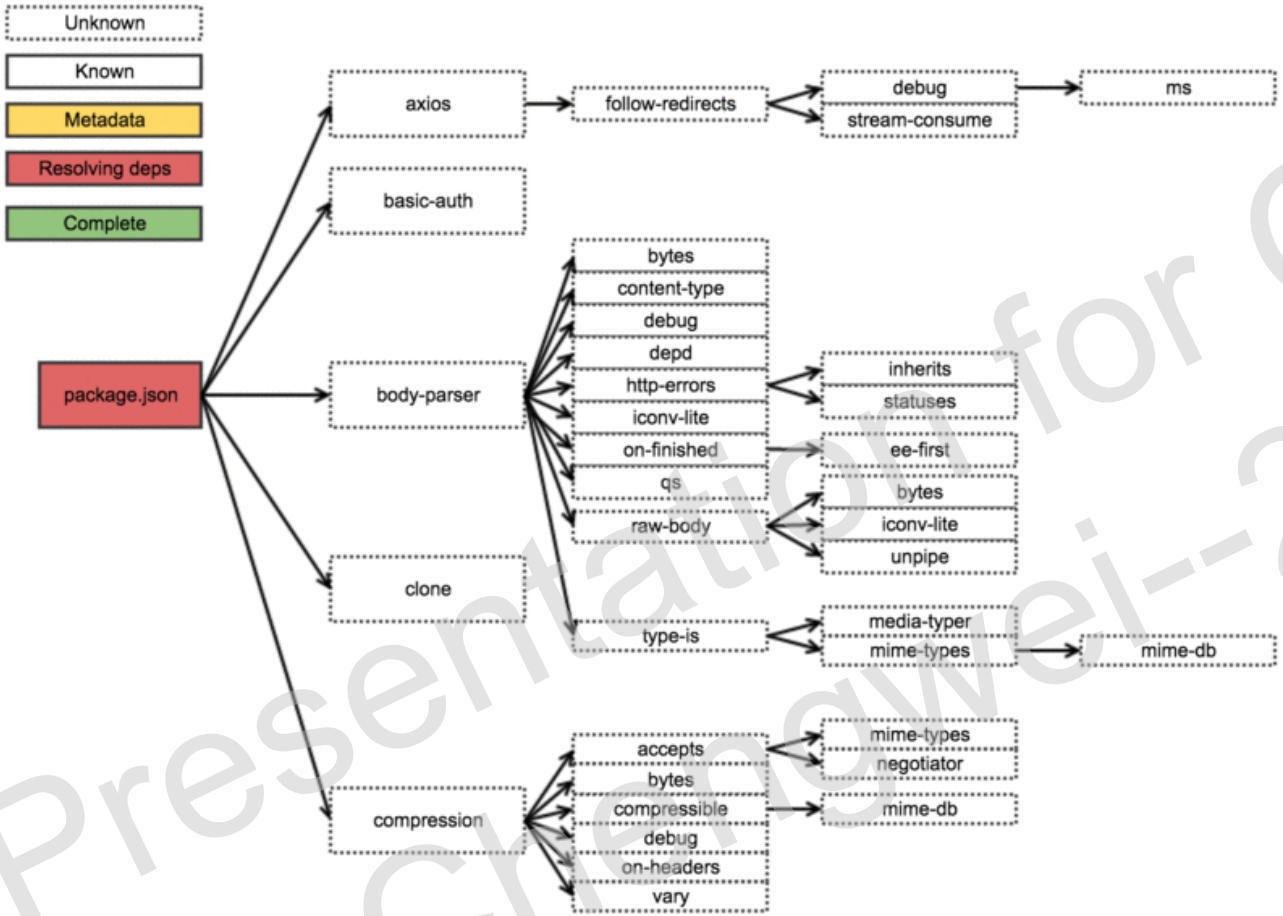
Elements	#Instances	Elements	#Instances
Lib	1,147,558	*has*	10,939,334
Ver	10,939,334	*upper*	9,804,406
Vul	815	*lower*	9,804,406
depends	62,232,906	*affects*	23,217
default	61,940,009	*libaffects*	830
libdeps	4,216,742	Graph size	15.15GB

Data Validation

1) We have captured 100% libraries and 99.96% versions from NPM registry.
2) Only 0.36% dependency relations are not successfully resolved
3) CVEs are cross validated with different sources.

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

…

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

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

Validation Data Set

103,609 versions (almost 1% of the entire NPM ecosystem) from 15,673 libraries are sorted out.

- Ground Truth: Dependency Trees from installation (**Install Tree**)
- Comparison Tool: npm-remote-ls (**Remote Tree**)
- Our Tool: DTResolver (**Graph Tree**)

DTResolver

Validation of Dependency Tree Resolution

90.58% of Graph Trees are exactly the same with Install Tree.
In comparison, only 53.33% of Remote Trees are same with Install Tree.

Two main reasons that cause the mismatch of differences between ***Install Tree*** and ***Graph Tree.***

1. Installation may not be complete.
2. Dependency tree from ***npm ls*** are deduped

DTResolver

Validation

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

We found 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

(DTCs from release to current for 50K library versions from validation set, 10.9M 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 is there still a considerable portion of CVEs not removed?

RQ2.4: Example of remediation by avoiding vulnerability introduction.

Empirical Study

Early Explorations (Not included in the paper)

PRQ1: ***Dependency Tree Complexity***

(Dependency Trees of all 10M library versions)

PRQ1.1: *How large are dependency trees, especially transitive dependencies?*

PRQ1.2*: What does NPM dependency resolution bring to dependency trees?*

PRQ2: ***Dependency Tree Changes***

(581,192 library versions released in the recent three months , 34 million dependency trees)

PRQ2.1: How frequently do developers change dependencies of their projects initiatively?

PRQ2.2: How frequently does the dependency tree change imperceptibly?

PRQ1: Dependency Tree Complexity

PRQ 1.1 How large are dependency trees, especially transitive dependencies?

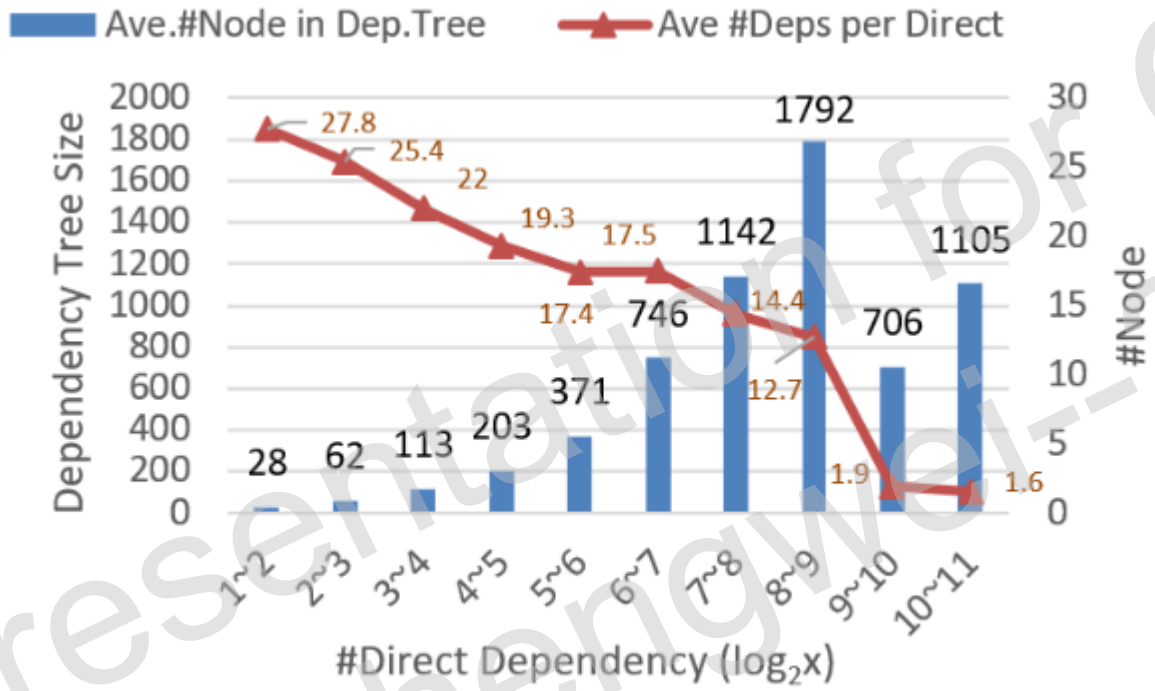


Figure 3.2: Distribution of dependency tree size over direct dependencies

Transitive dependencies are much larger than direct dependencies

each direct dependency introduces 22 transitive dependencies,

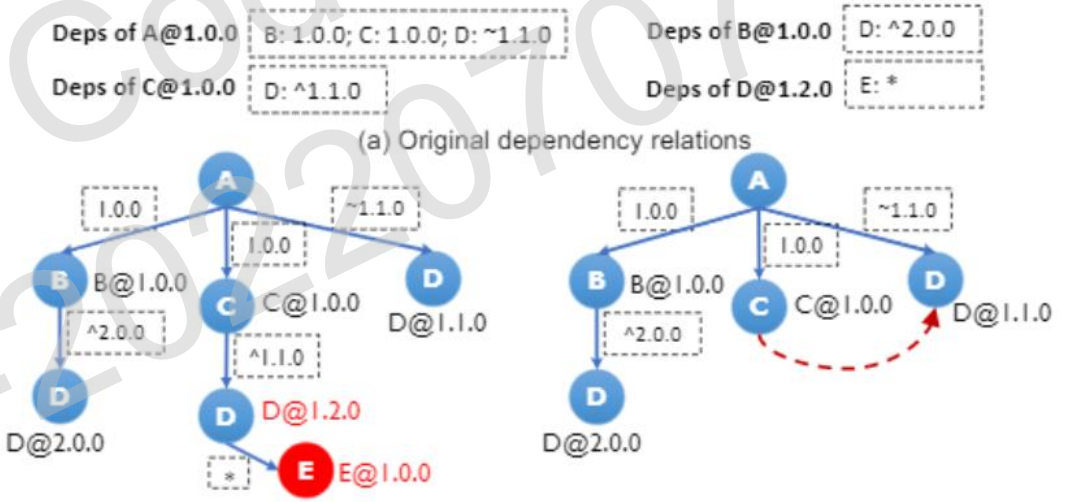
the portion of transitive dependencies drops along with the increase of direct dependencies due to package reuse.

PRQ1: Dependency Tree Complexity

PRQ 1.2 What does NPM dependency resolution bring to dependency trees?

Special Nodes in dependency tree

- **Shared-Dependency Node (SDN)** represents one single node is shared by multiple dependencies in one dependency tree
- **Multi-Version Node (MVN)** refers that multiple versions of the same library are installed for different dependencies,
- **Circle-Dependency Node (CDN)** represents nodes in dependency loop



(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

PRQ1: Dependency Tree Complexity

PRQ 1.2 What does NPM dependency resolution bring to dependency trees?

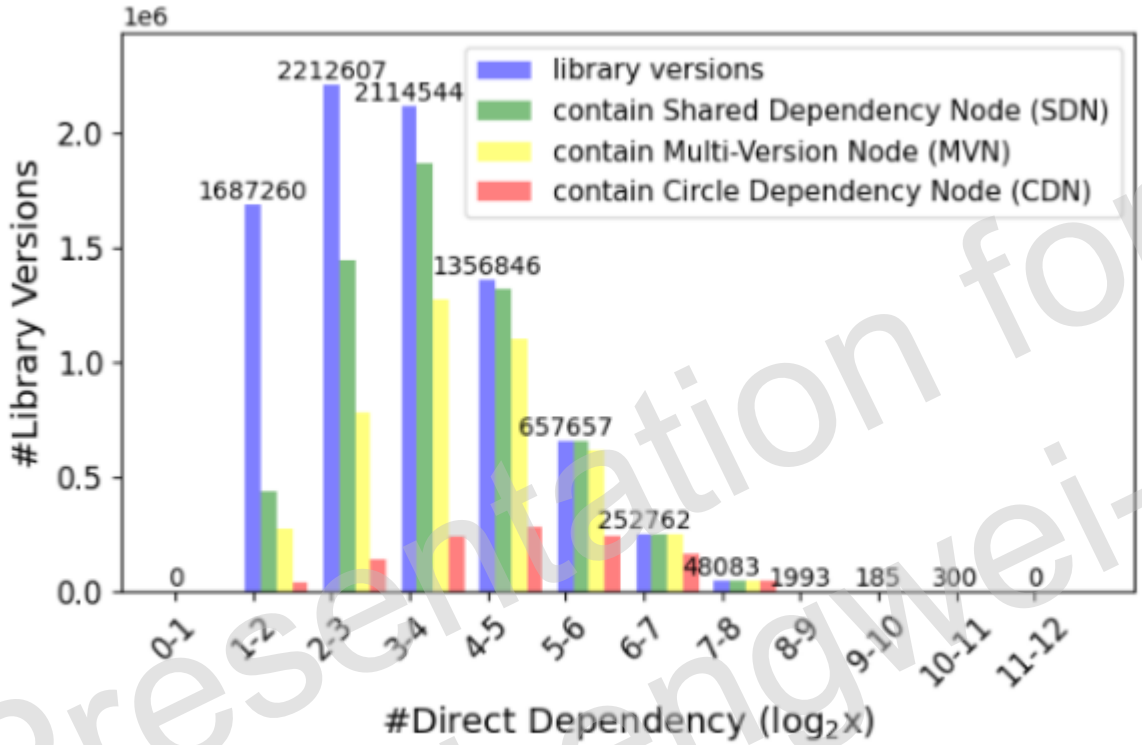


Figure 3.3: Distribution of containing complex structures over direct dependencie...

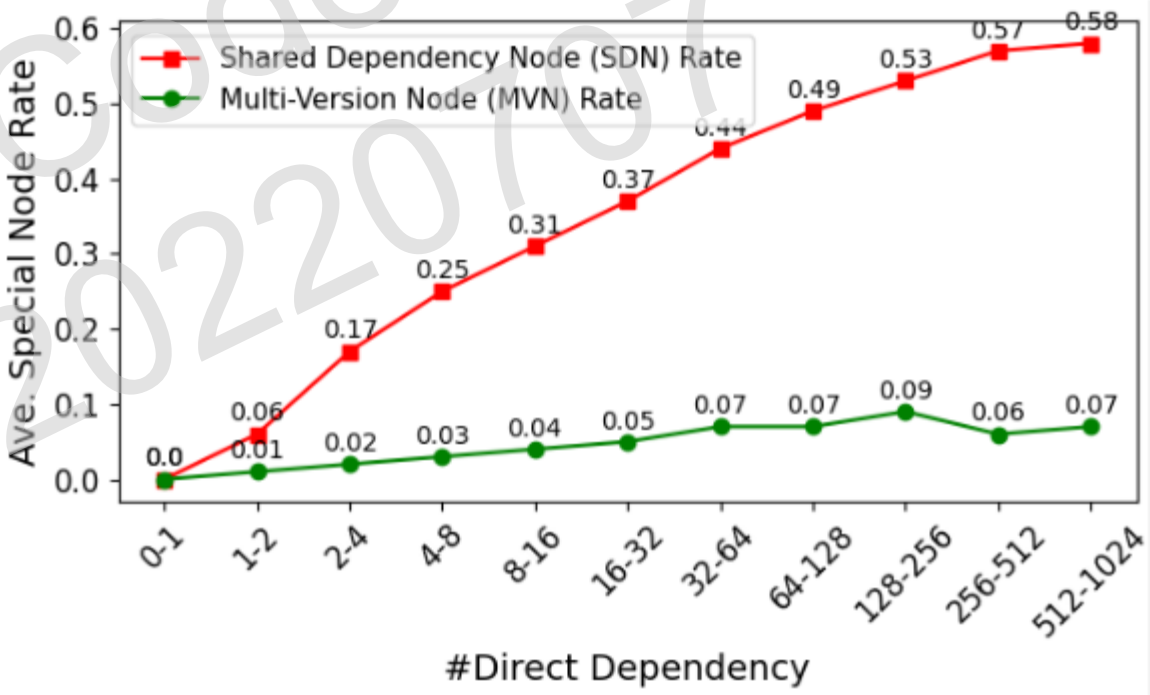


Figure 3.4: Complex structure rate over direct dependencies

With the increase of direct dependencies, the structural complexity of dependency trees also rises.

RQ1: Vulnerability Propagation via dependency trees

RQ1: **Vulnerability Affection via Dependencies**

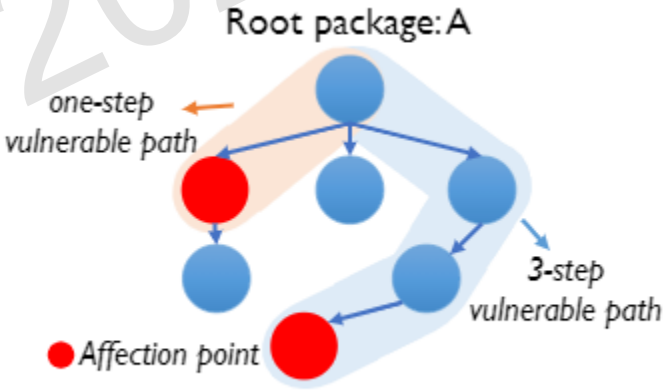
RQ1.1:How many packages are affected by existing known vulnerabilities in the NPM ecosystem?

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

Experiment Objects:

The 10.93 million dependency trees from

PRQ1 and corresponding vulnerabilities.



(b) Vulnerable Paths

RQ1: Vulnerability Propagation via dependency trees

RQ 2.1 How largely does existing known vulnerabilities affect the NPM ecosystem?

1) One-quarter versions belonging to 19.96% libraries are affected by vulnerabilities (416 out of 815 CVEs) from dependencies;
2) Latest versions of libraries (16.17%) are still under potential risks of being affected by vulnerabilities via their dependencies;
3) A considerable portion of vulnerable libraries (Over 100) that are used by others, still have vulnerable latest versions.

RQ1: Vulnerability Propagation via dependency trees

RQ 2.2 How do vulnerabilities propagate to affect root packages?

1) There are centrality that some influential known CVEs widely exist in the dependency trees of a significant portion of packages; (25 CVEs s have affected over 10k libraries or 100k versions (1% of the entire ecosystem))
2) Packages are usually affected by 3.97 affection points, and each affection points affect root packages via 8 vulnerable paths;
3) Vulnerabilities still widely exist in direct dependencies of affected library versions (over 30%), even the latest versions.
4) Most of the vulnerable paths go through limited direct dependencies （49.57% for 1, 78.94% for 3）, which could be utilized to cut off vulnerable paths.

Table 3.1: 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

PRQ2: Dependency Tree Changes

PRQ 2.1 How frequently do developers change dependencies of their projects initiatively?

Has Version Update: 70.82% of libraries (812,662),

Has Direct Dependency Changes: only 33.61% of them (385,681).

In total, 9,804,419 times of version updates,

and 29.17% of them (2,859,805) come with direct dependency changes.

Version Updates	Releases	Has Dependency Change
Major	2.96%	44.68%
Minor	13.33%	33.80%
Patch	83.71%	27.88%

In total, 9,804,419 times of version updates have been identified, we find that

- *133 libraries (0.01%) have time intervals that are shorter than 1 second,*
- *55,041 libraries (4.8%) have time intervals that are shorter than 1 minute,*
- *560,553 libraries (48.85%) have time intervals that are shorter than 1 hour.*

Besides, we also count the number of short time intervals and find that:

- *1,127 time intervals (0.01%) are shorter than 1 second,*
- *152,692 time intervals (1.56%) are shorter than 1 minute,*
- *3,000,093 time intervals (30.6%) are shorter than 1 hour.*

PRQ2: Dependency Tree Changes

PRQ 2.2 How frequently do developers change dependencies of their projects initiatively?

1) Library maintainers have some extremely frequent version releases that 4.80% and 48.85% of libraries have ever released multiple versions within a minute and an hour.

2) Dependency trees exponentially amplify the frequent DTCs that almost 90% of dependency trees only persist for less than a day.

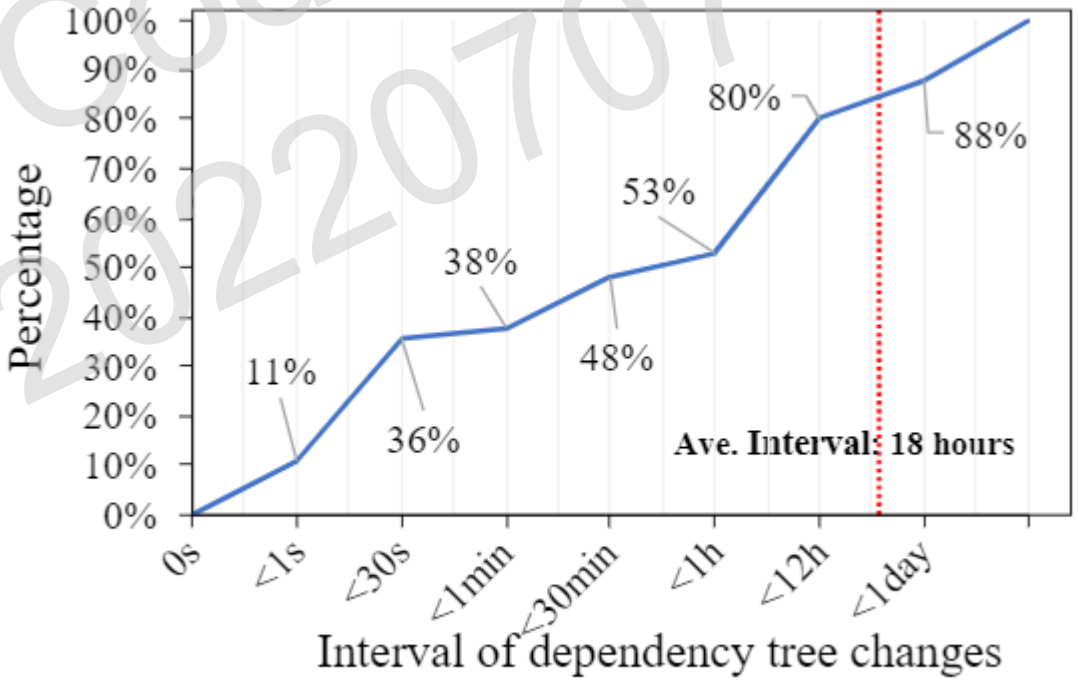


Figure 3.5: Distribution of intervals between dependency tree changes

RQ2: Vulnerability Propagation Evolution in Dependency Trees

RQ2: **Vulnerability Affection Evolution in Dependencies**

RQ2.1: How does known vulnerability propagation evolve over time?

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

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

RQ2.4: Example of remediation by avoiding vulnerability introduction.

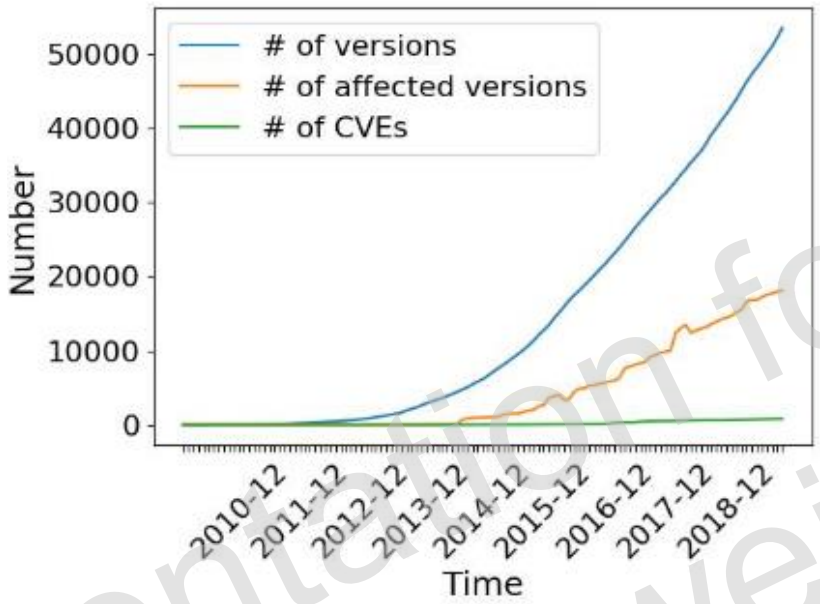
Experiment Objects:

53,541 versions are selected from Experiment objects of DTResolver

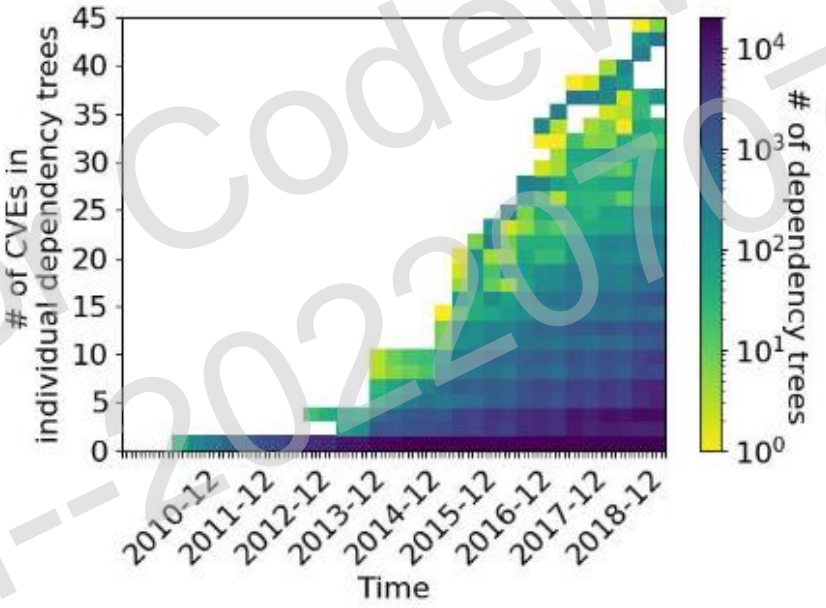
10,906,781 dependency trees of them from release to experiment time

RQ2: Vulnerability Propagation Evolution in Dependency Trees

RQ 2.1 How does known vulnerability propagation evolve over time?



(a) Evolution of library versions and CVEs



(b) Evolution of CVE density in dependency trees

1) Known vulnerabilities are causing a larger impact across the NPM ecosystem over time as expected.
2) They are not only affecting more library versions but also affecting them with more affection points in dependency trees.

RQ2: Vulnerability Propagation Evolution in Dependency Trees

RQ 2.2 How long do vulnerabilities live in dependency trees?

1) Most of the CVEs (93%) have already been introduced to dependency trees before they were discovered, and the fix versions of these CVEs (87%) were also mostly released before CVE publish.

2) Based on these timely releases, 90% of the vulnerable dependencies can be removed along with time (averagely still takes a year), but there are still 40% of vulnerabilities unable to get thoroughly excluded.

RQ2: Vulnerability Propagation Evolution in Dependency Trees

RQ 2.3 Why is there still a considerable portion of CVEs not removed?

Not Removed Paths (NRP): Vulnerable paths remained in latest dependency trees

Removed Paths (RP): Vulnerable paths removed by DTCs.

Vulnerable Latest Version, the highest satisfying version of affection point is vulnerable;

Fixed-Ver. D.C., the dependency constraints are fixed versions instead of ranges

(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) Affection points in 61.54% of NRPs are Vulnerable Latest Versions, while only 16.12% of RPs are in such cases.

Outdated Maintenance (provider) and Unsuitable Dependency Constraint (consumer) are the main reasons that postpone or even block the automated vulnerability removal in dependency trees over time.

RQ2: Vulnerability Propagation Evolution in Dependency Trees

RQ 2.4 What have library maintainers done to prevent CVEs from spread? (Not in the paper)

CVEs	# of affected versions	CVEs	# of affected libraries	Measures	Corresponding library
CVE-2019-10747	948208	CVE-2019-10747	73947	deprecate all vul versions	set-value
CVE-2018-3728	741754	CVE-2018-3728	62227	deprecate library	hoek
CVE-2016-10540	388126	CVE-2016-10540	41423	deprecate all vul versions	minimatch
CVE-2018-1000620	601414	CVE-2018-1000620	52334	deprecate library	cryptiles
CVE-2018-20834	592691	CVE-2018-20834	48356	releasing new versions	tar
CVE-2018-16487	819360	CVE-2018-16487	77433	releasing new versions	lodash
CVE-2019-10744	867148	CVE-2019-10744	79459	releasing new versions	lodash
CVE-2019-1010266	712971	CVE-2019-1010266	70956	releasing new versions	lodash
CVE-2018-3721	790100	CVE-2018-3721	75817	releasing new versions	lodash
CVE-2015-8857	135050	CVE-2015-8857	13825	releasing new versions	uglify-js
CVE-2015-8858	157787	CVE-2015-8858	15870	releasing new versions	uglify-js
CVE-2019-16769	253044	CVE-2019-16769	15988	releasing new versions	serialize-javascript
CVE-2018-3738	113267			releasing new versions	protobufjs
CVE-2017-16137	509455	CVE-2017-16137	38626	releasing new versions	debug
CVE-2017-16113	101154			no action	parsejson
CVE-2017-16138	347945	CVE-2017-16138	33770	releasing new versions	mime
CVE-2017-18214	109054			releasing new versions	moment
CVE-2017-1000048	353823	CVE-2017-1000048	35533	releasing new versions	qs
CVE-2019-10742	166666			releasing new versions	axios
CVE-2016-2515	144461	CVE-2016-2515	16033	deprecate library	hawk
CVE-2017-16026	155293	CVE-2017-16026	17510	deprecate library	request
CVE-2014-10064	146043	CVE-2014-10064	16787	releasing new versions	qs
CVE-2014-7191	146043	CVE-2014-7191	16787	releasing new versions	qs
CVE-2017-16119	148736	CVE-2017-16119	13307	releasing new versions	fresh
CVE-2018-3750	111870	CVE-2018-3750	10870	releasing new versions	deep-extend

RQ2: Vulnerability Propagation Evolution in Dependency Trees

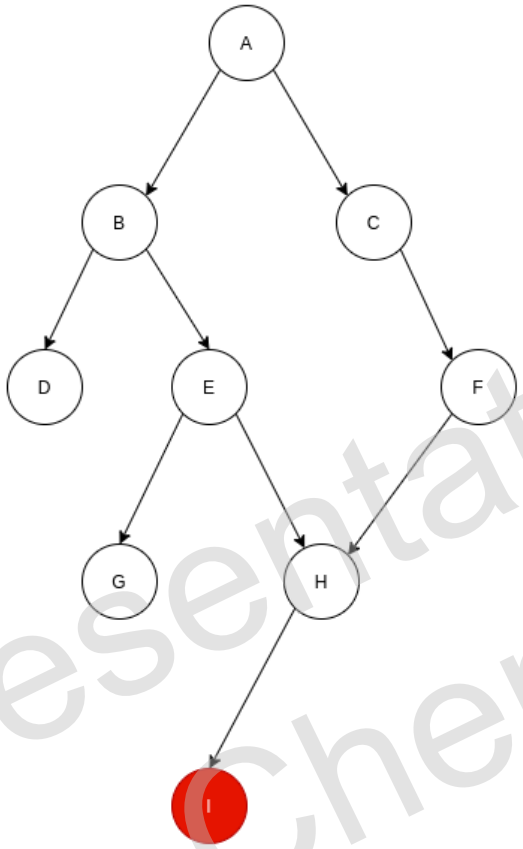
RQ 2.4 What have library maintainers done to prevent CVEs from spread? (Not in the paper)

CVE ID	library	Countermeasure	# of affected library	# of affected library whose latest version using old vulnerable version	# of these latest version that are released in 2019
CVE-2019-10744	lodash	Release a new version	79,459	59,519	9.55%
CVE-2019-10747	set-value	Deprecating vulnerable versions	73,947	57,871	38.02%

Countermeasures (i.e., releasing new versions and deprecating vulnerable ones, etc.) have been taken by third-party library maintainers when CVEs are discovered, while the effects are still limited.

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: Recursively resolve clean versions for next level dependencies (Forward Checking)

(2) BottomUp: Roll back and alter upper nodes when no clean version found (Backtracking), until resolve to clean tree.

Evaluation of DTReme

Objects: top 1K most popular JavaScript repositories (in Jan. 2021) from Github Trending

Original dependency tree are collected from installation. (DefDep)

Comparison Tool: npm audit (AuditDep)

Our Tool: DTReme (RemeDep)

Apart from unsuitable repos, we have 460 projects as experiment objects.

Table 2.2: Comparison of remediation effects between *npm audit fix* and our remediation

# of affection 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

❖ Sometimes *npm audit fix* violates SemVer

Considerable user projects contain unavoidable vulnerabilities even though we have exhausted all possible dependency trees!

Question

1. **Why are there still so many vulnerabilities unable to be remediated?**

2. **Are these vulnerabilities necessarily to be remediated? Any trade-off?**

C O N T E N T S

2. 基于程序静态调用图的NPM漏洞影响力分析

Motivation

1. Existing research demystifies vulnerability impact at package level, imprecise! So large the gap between “Contain” and “Reach” is to vulnerability impact analysis?

snyk Dependabot WhiteSource BLACKDUCK BY SYNOPSYS

2. Existing SCA result from different tools varies and contains lots of false positives.

1. Existing SCA tools detect vulnerabilities on package level
2. Vulnerabilities may not affect downstream users if not triggered
3. Vulnerability data could be incorrect.

Overreacted

npm audit: Broken by Design

July 7, 2021 · 14 min read

Security is important. Nobody wants to be the person advocating for less security. So nobody wants to say it. But somebody has to say it.

So I guess I’ll say it.

The way `npm audit` works is broken. Its rollout as a default after every `npm install` was rushed, inconsiderate, and inadequate for the front-end tooling.

Have you heard the story about the boy who cried wolf? Spoiler alert: the wolf eats the sheep. If we don’t want our sheep to be eaten, we need better tools.

As of today, `npm audit` is a stain on the entire npm ecosystem. The best time to fix it was before rolling it out as a default. The next best time to fix it is now.

In this post, I will briefly outline how it works, why it’s broken, and what changes I’m hoping to see.

Motivating Example

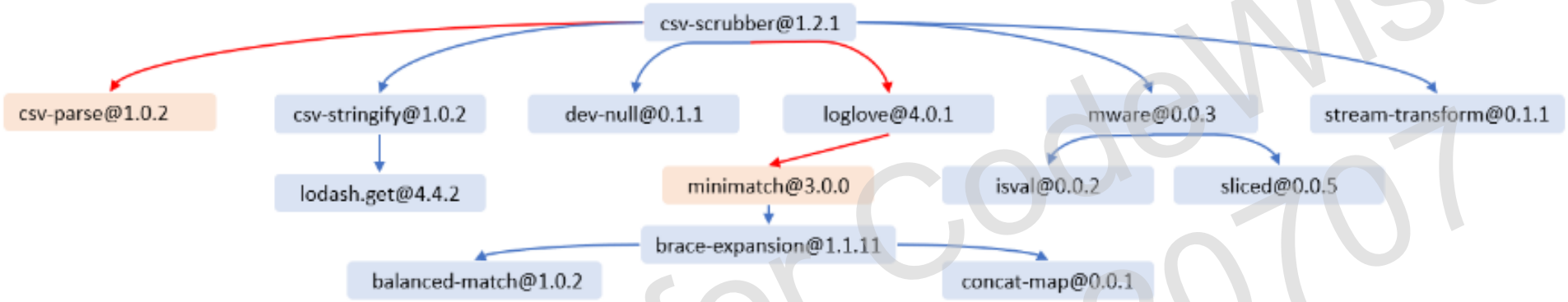


Fig. 2. The dependency tree of *csv-scrubber@1.2.1*.

There are 2 call paths to reach the vulnerable code of minimatch@3.0.0, while the vulnerable code from direct dependency csv-parse@1.0.2 is unreachable!

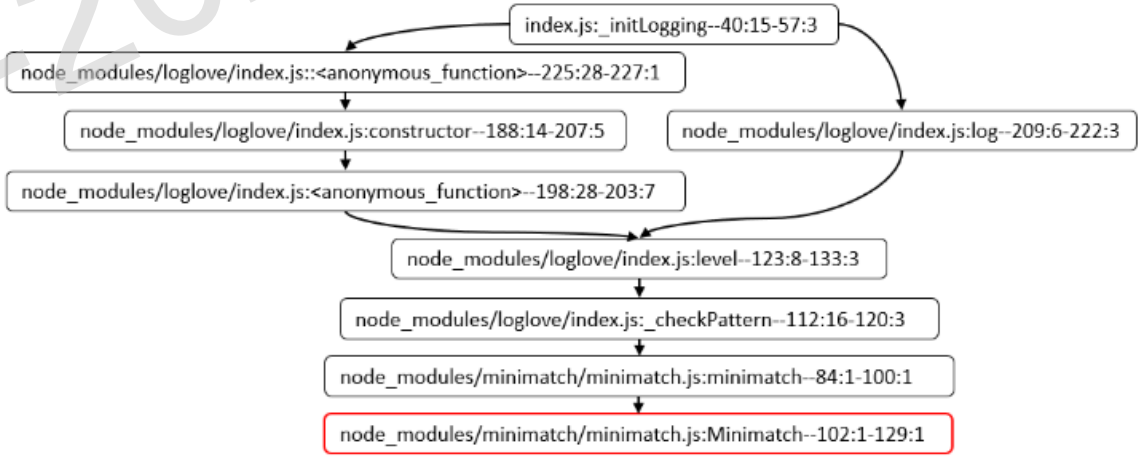


Fig. 3. The call paths to reach CVE-2016-10540 in *minimatch@3.0.0*.

Methodology (Ongoing Work)

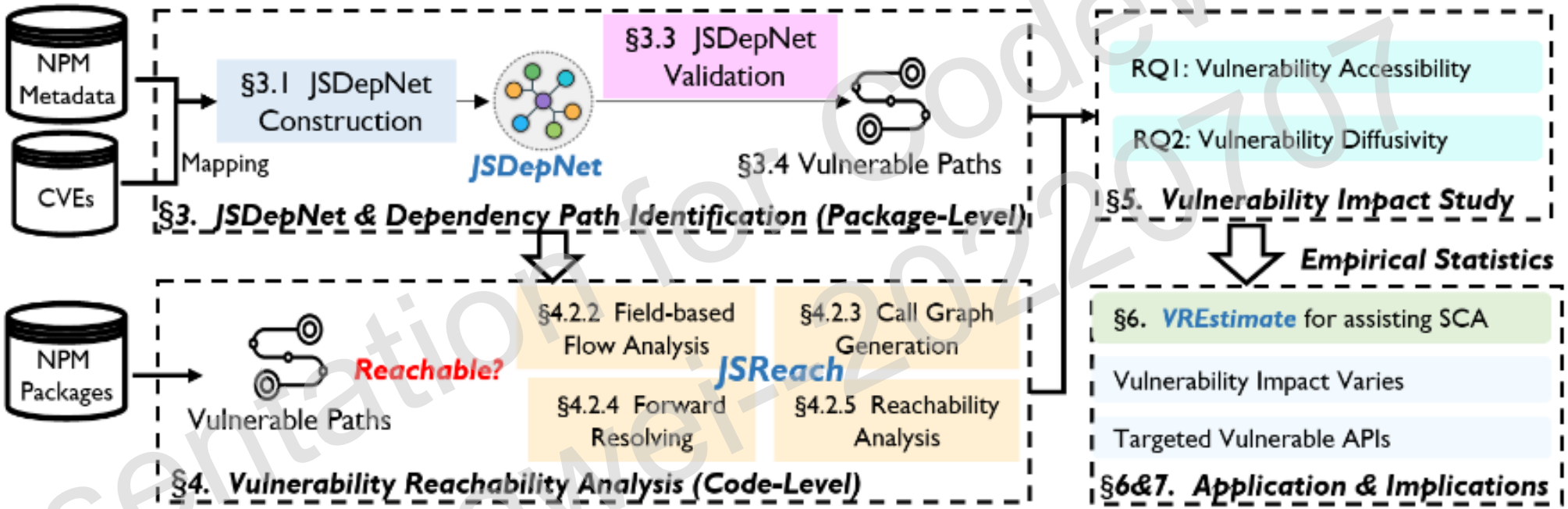


Fig. 1. The overview of our work.

Methodology (Ongoing Work)

Challenges on call graph generation with missing dependencies

```
// module A
var b = require(B);
module.exports = function a(p){return b(p);};
// module B
module.exports = function b(p){return function c(){p();};};
// module D
module.exports = function d() {return function e();};
// module F
var a = require(A);var d = require(D);
function f(){x = a(d);y = x(); y();};
```

Fig. 7. An example of missing modules in the middle.

Inspired by the concept of Access Path from JAM, we developed JSReach based on CodeQL with 87.85% precision and 95.81% recall on call graph generation with only dependency paths provided.

Study

- **Accessibility**: captures how easy are vulnerabilities can be reached by dependent packages via direct API calls;

$$Accessibility_{vul} = \frac{\forall p.p \in reach\_depts(vp)}{\forall p.p \in depts(vp))}, vul \subseteq vp$$

Experiment Objects:

107 well selected vulnerable versions (108 CVEs) from popular libraries, 11,594 direct dependents are collected.

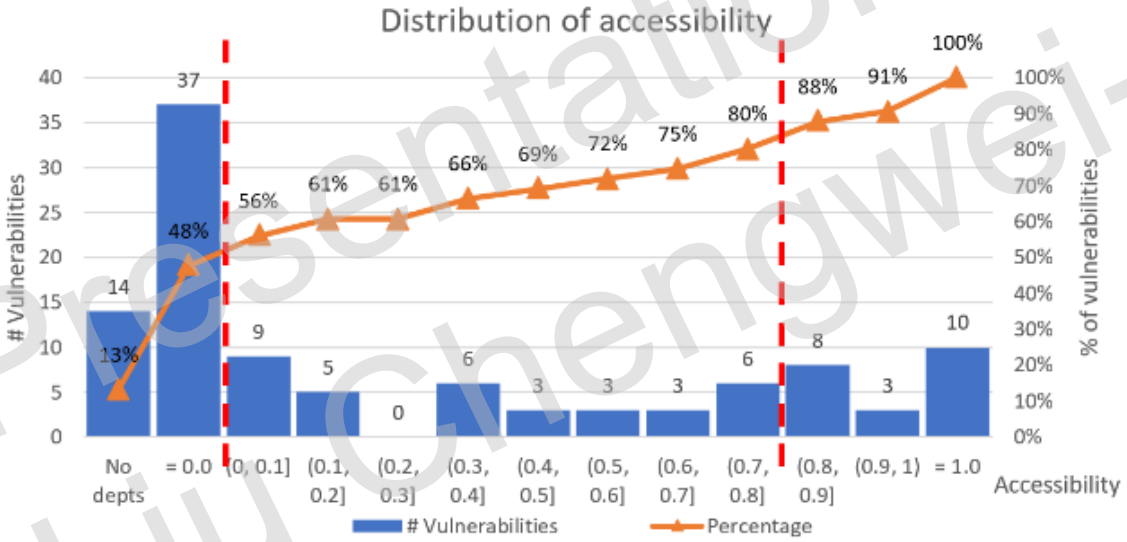


Fig. 9. Distribution of accessibility of vulnerabilities

(1) 69% of vulnerabilities can only be accessed by less than half of dependent packages, but

(2) there also exist 20% of vulnerabilities having high accessibility that they can be accessed by over 80% of dependent packages.

Study

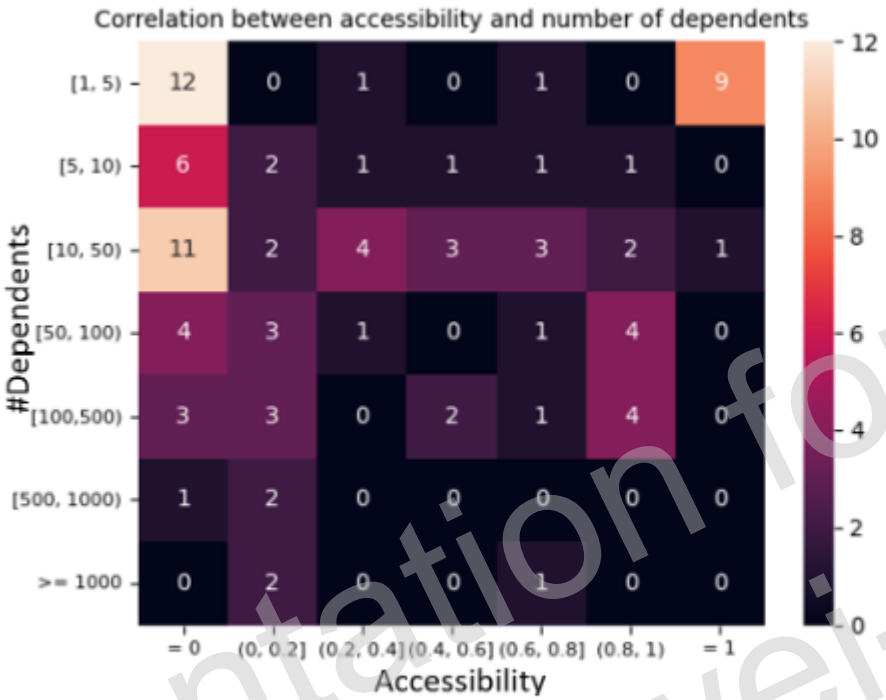


Fig. 10. Correlation of accessibility of vulnerabilities and number of dependent packages.

Vul Pack.	CVE	#depts	#r_depts	Acc.
jquery:2.2.4	CVE-2015-9251	2,336	27	1.16%
lodash:4.17.4	CVE-2018-3721	1,567	1,174	74.92%
lodash:4.17.11	CVE-2019-10744	1,040	79	7.60%
lodash:4.17.10	CVE-2018-16487	626	63	10.06%
string:3.3.3	CVE-2017-16116	551	33	5.99%
axios:0.18.0	CVE-2019-10742	496	0	0.00%
handlebars:3.0.8	CVE-2015-8861	465	0	0.00%
lodash:4.17.19	CVE-2020-8203	311	25	8.04%
request:2.67.0	CVE-2017-16026	274	256	93.43%
jquery:3.3.1	CVE-2019-11358	245	2	0.82%
bootstrap:3.3.7	CVE-2018-20676	225	8	3.56%
node-fetch:2.6.0	CVE-2020-15168	196	157	80.10%
jquery:3.4.1	CVE-2020-11022	178	0	0.00%
qs:0.6.6	CVE-2014-7191	177	88	49.72%
jquery:1.8.3	CVE-2012-6708	111	74	66.67%
marked:0.3.6	CVE-2017-1000427	104	90	86.54%
tar:1.0.3	CVE-2015-8860	103	89	86.41%

Table 10. Accessibility of CVEs with more than 100 direct dependents.

1) There are 17 CVEs that have more than 100 direct dependent packages, while only 6 of them have high accessibility (higher than 50%).
2) (2) CVEs from the same library may also have completely different accessibility.

Study

- **Diffusivity**: Extended to the breadth and distance that how largely can downstream dependents access the vulnerabilities via transitive dependency paths

Table 11. Overall diffusion rate of all vulnerabilities for multi-steps.

Step	#Seeds	#Depts	#Ave. depts	#Reac_depts	Diffusion rate
1	94	11,594	123	2,994	25.82%
2	524	2,125	4	869	40.89%
3	166	472	3	292	61.86%
4	53	166	3	115	69.28%
5	23	58	3	38	65.52%
6	7	16	2	10	62.50%
7	4	4	1	3	75.00%

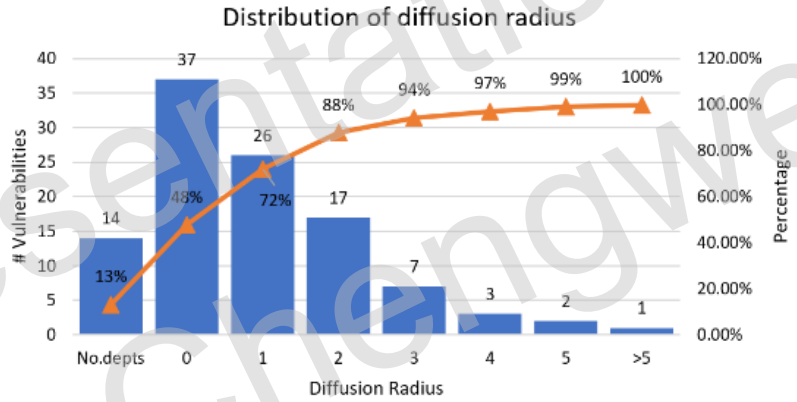


Fig. 12. Distribution of diffusion radius of vulnerabilities.

Table 12. Detailed diffusion rate of vulnerable packages with diffusion radius ⩾ 3.

Vulnerable package	DR_1	DR_2	DR_3	DR_4	DR_5	DR_6	DR_7
node-fetch:2.6.0	81.40%	65.23%	54.17%	66.95%	67.44%	62.50%	75.00%
qs:0.6.6	46.89%	21.82%	78.57%	100.00%	66.67%	NA	NA
lodash:4.17.4	75.12%	73.32%	63.35%	93.55%	1	NA	NA
marked:0.3.3	91.49%	26.83%	83.33%	50.00%	0	NA	NA
merge:1.2.0	82.83%	37.68%	55.56%	42.86%	NA	NA	NA
ms:0.7.0	60.00%	87.50%	83.33%	33.33%	NA	NA	NA
xmlhttprequest:1.6.0	32.76%	75.00%	100.00%	NA	NA	NA	NA
morgan:1.9.0	86.30%	72.73%	100.00%	NA	NA	NA	NA
lodash:4.17.19	8.28%	50.00%	100.00%	NA	NA	NA	NA
lodash:4.17.10	9.82%	31.58%	100.00%	NA	NA	NA	NA
tar:1.0.3	86.41%	10.61%	100.00%	NA	NA	NA	NA
marked:0.3.5	78.64%	18.06%	66.67%	NA	NA	NA	NA
request:2.67.0	92.72%	25.13%	35.29%	NA	NA	NA	NA

Most of the vulnerabilities can only propagate to affect dependent packages within limited dependency steps!

VREstimate

For traditional SCA scenario, we can still use JSReach to tell the reachability of vulnerabilities.

But what if the source code of user projects are unavailable?

$$Reachability(vul, P) = \prod_{i=1}^{len(P)} DR_i$$

$$Reachability(vul) = \max_{i<count(P)} \{Reachability(vul, P_i)\}$$

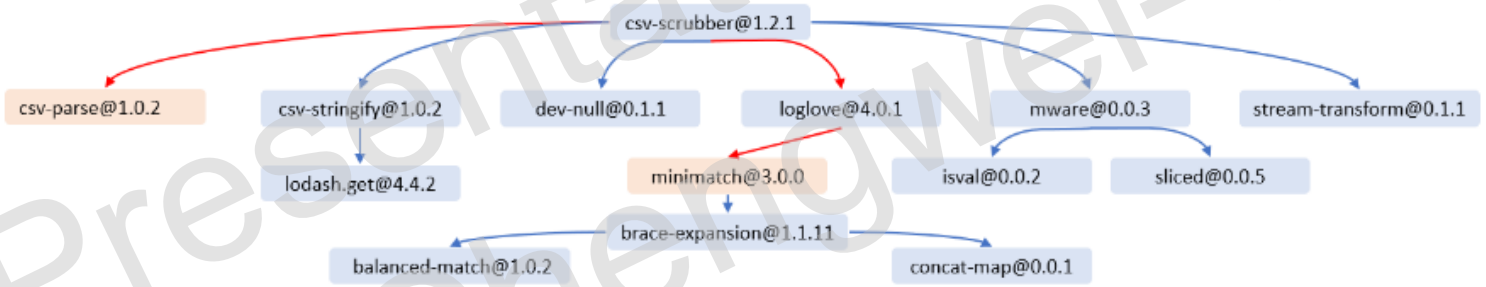


Fig. 2. The dependency tree of *csv-scrubber@1.2.1*.

For CVE-2016-10540,

$Reachability$(CVE-2016-10540)

= Reachability(CVE-2016-10540, path)

= 0.8 * 0.58 = 0.464

VREstimate

Experiment Objects:

173 projects from top 1000 most starred JavaScript projects from Github.

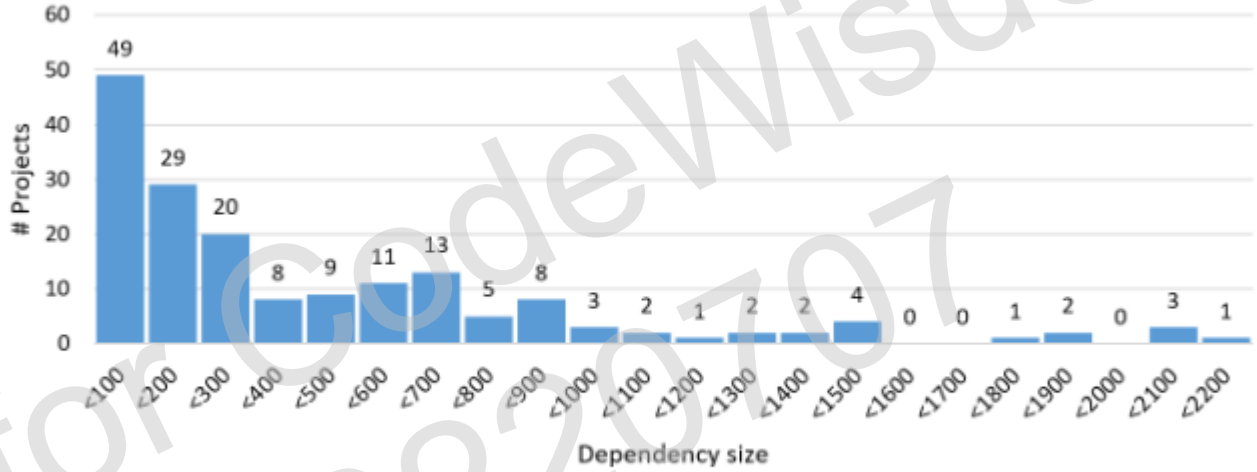


Fig. 13. Distribution of dependency size of selected projects.

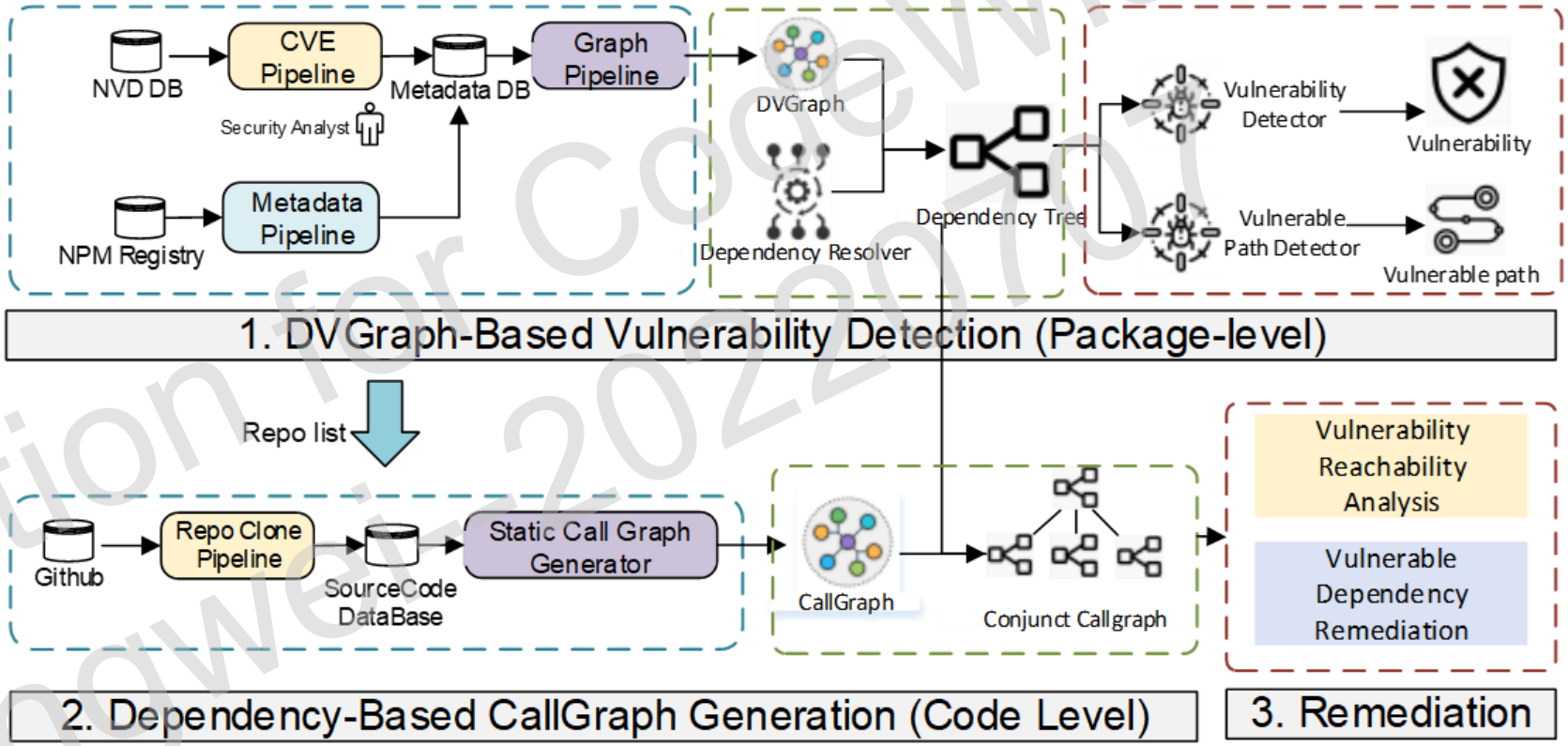
Table 13. Distribution of vulnerability reachability against different levels of *Reachability* metric.

Level	Range	#*vul_points*	#Reachable *vul_points*	#Unreachable *vul_points*	Reachable Rate
L1	R=0	81	7	74	8.64%
L2	R∈ (0, 0.25]	271	42	229	15.50%
L3	R∈ (0.25, 0.5]	35	5	30	14.26%
L4	R∈ (0.5, 0.75]	38	12	26	31.58%
L5	R∈ (0.75, 1]	11	6	5	54.54%
Total	N.A.	436	72	364	**16.51%**

Static Call Graph

Approaching Vulnerability

1. Contain
2. Reach
3. Trigger?



C O N T E N T S

3. 关于开源治理的一些思考及尝试

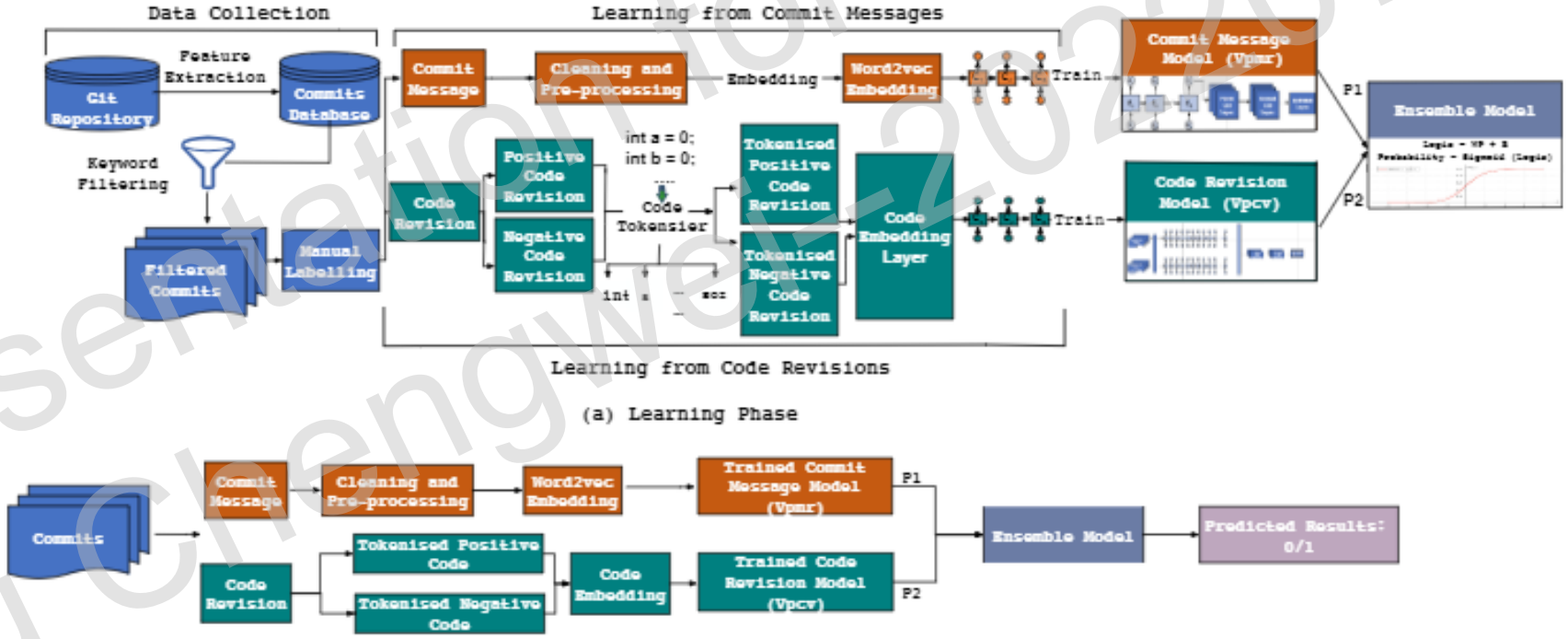
Opportunity and Challenge co-exist

- **Vulnerability Detection & Management**
  - **Affected libraries and versions**
  - **Vulnerable code**
  - **Patches**
  - **POCs**
- **Software Component Analysis (SCA)**
  - **Package managers/Dependency resolutions**
  - **Code Clone**
  - **Variation of SCA tools**
- **Vulnerability Impact Analysis**
  - **Weak links**
  - **Fragile points**
  - **Alerts**
- **Trustworthy OSS**
  - **Trustworthy TPLs (Security, Quality)**
  - **Trustworthy Developers (Maintainers, Develop Teams)**

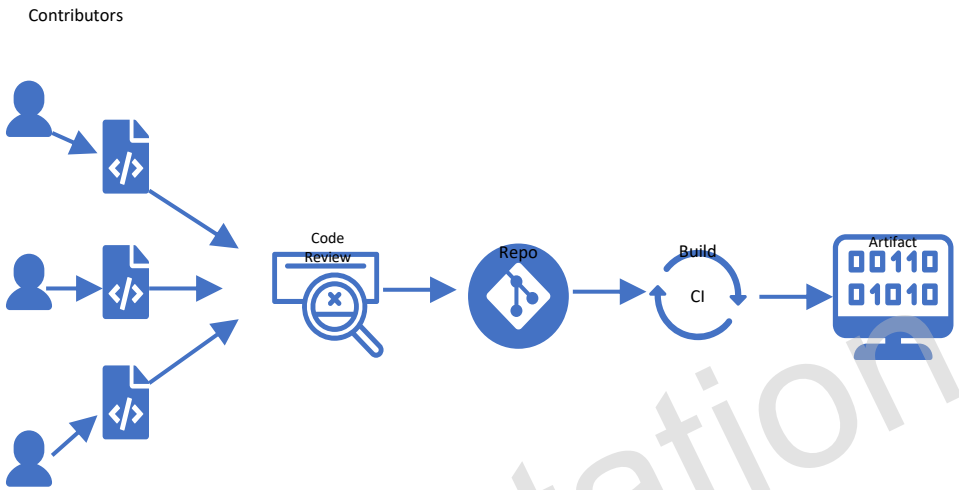
Vulnerability Analysis	• AI-Based Vulnerability Data Collection • Vulnerability Detection
SCA 2.0	• Software Compositional Analysis for Full SDLC • SAST Integration
Ecosystem Analysis	• Supply Chain Security • Vulnerability Impact Analysis • Ecosystem-wide Pre-warning
OSS Risk Analysis	• License Compliance / Conflict Detection • Development Risk Analysis • Malicious Maintainer Identification
OSS Governance	• Software Digitization Platform • OSS Health Profile– Osspert • OSS-Ops

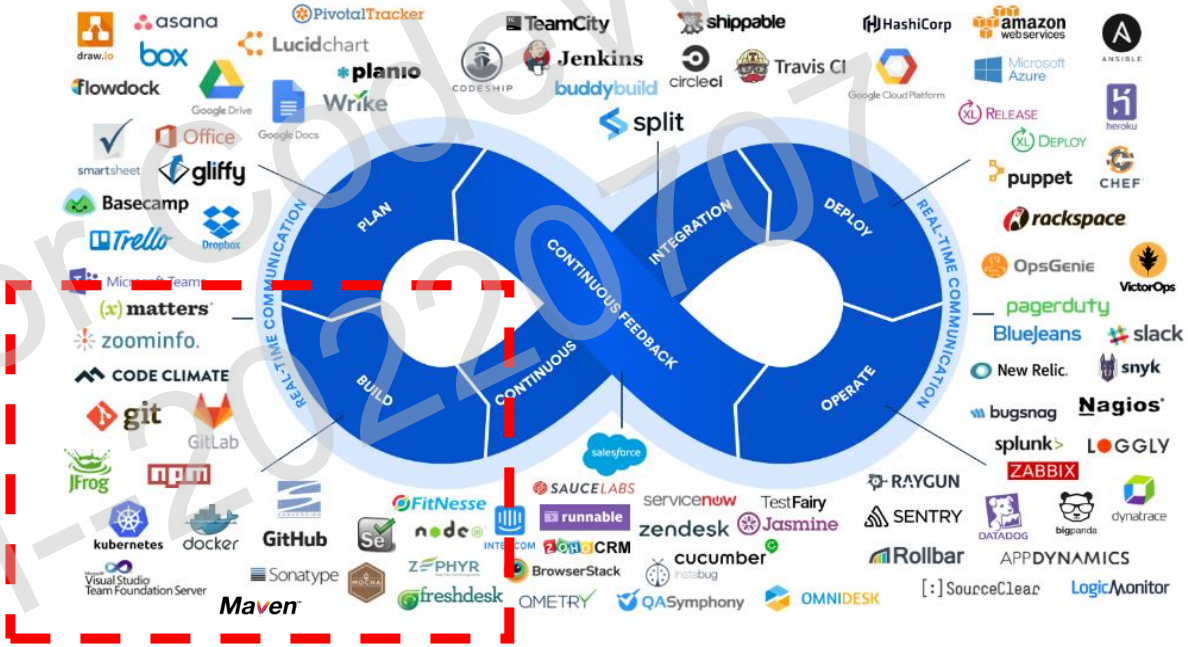
Automated Vulnerabilities Identification via Deep learning

- Learning both from commit messages and code revisions
  - LSTM + CNN network architecture



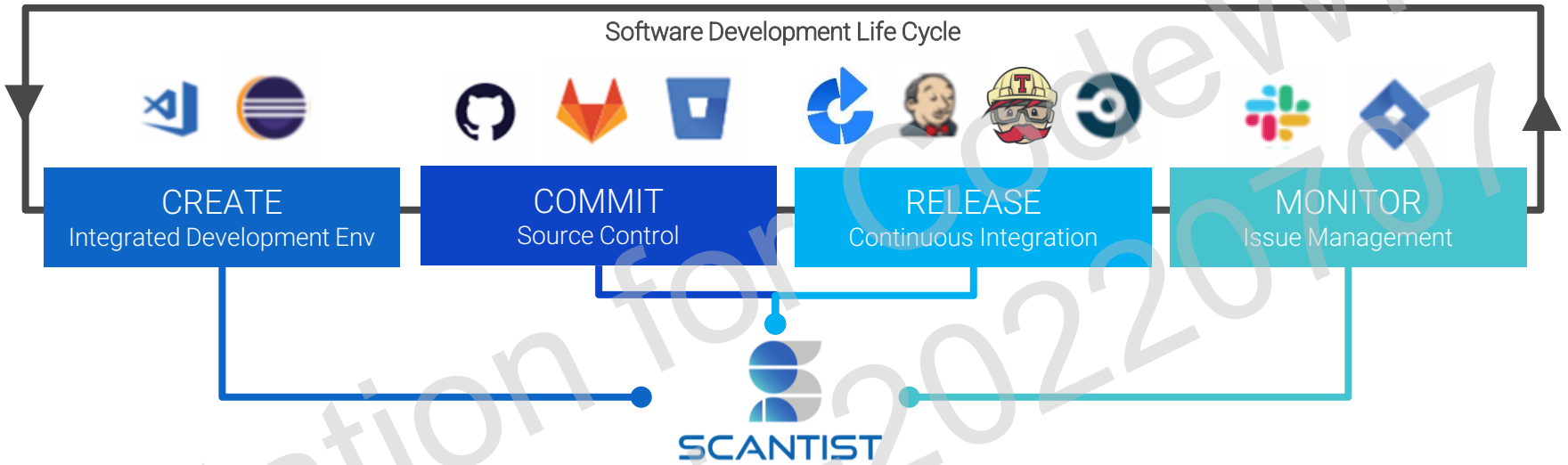
SCA2.0：Software Compositional and Risk Analysis towards the full SDLC





Currently SCA tools mainly focus on dependency identification and related vulnerability, license detection and analysis, which should be extended to the monitoring and management throughout the full SDLC.

SCA2.0：Software Compositional and Risk Analysis towards the full SDLC



From **Source code** to **Binaries**, From **development** to **production**,
Scantist aims to secure the continuous security of software

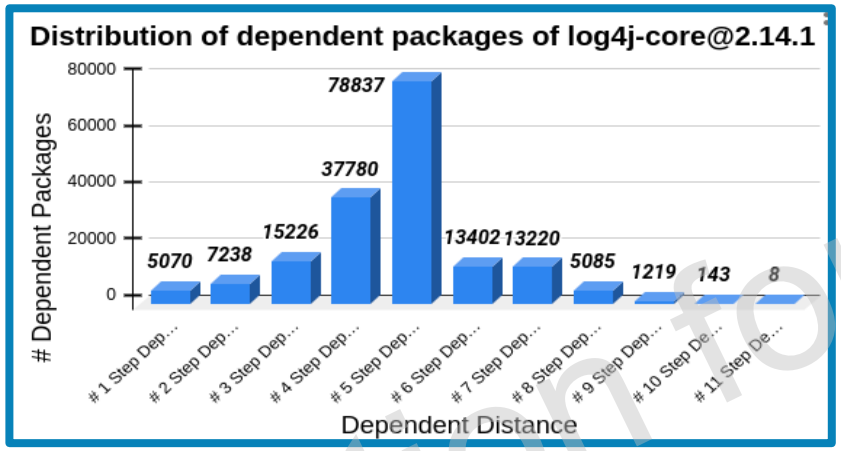
- Code Clone/IP Detection
- Security Analysis for Auto-Generated Code
- Security Lints
- Code Smell Detection
- IDE Abnormal Behaviour Detection
- IDE Security Plug-ins

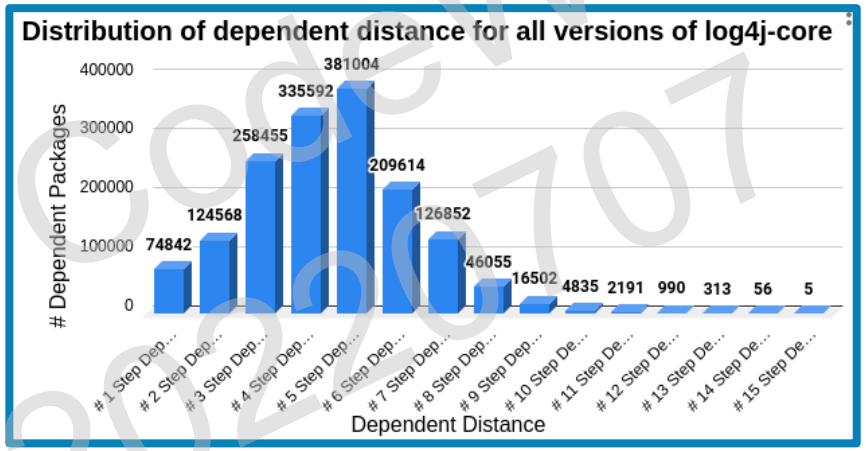
- Vulnerability Detection
- Patch Generation
- Software Architecture Analysis
- Development Measurement
- Risk analysis for developers
- Contribution Measurement
- Identification of abnormal commits and PRs
- Auto Audit
- Pre-build Dependency Resolution

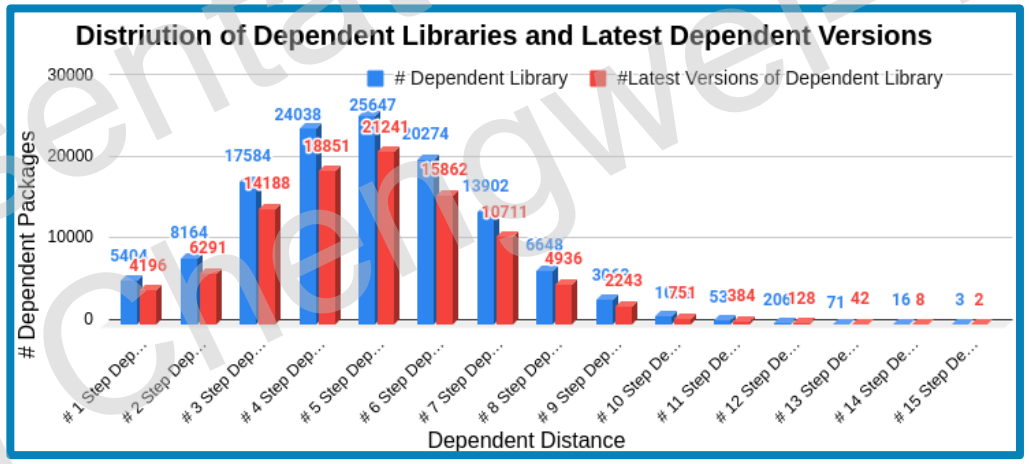
- SCA
- Vulnerability Detection from Dependencies
- Reachability Analysis
- Remediation
- Compatibility Analysis
- License Conflict Detection （Project, Library, File）
- TPL Health Analysis(
- Technical Lags
- Security Analysis for Build Scripts

- Runtime Environment Security
- Runtime Dependency Analysis
- Cloud Natives Dependency Analysis
- Cross Language Dependency Analysis
- System Dependency Analysis

Supply Chain Security Analysis and Pre-Warning --Log4Shell Impact Analysis







Apache LOG4J™

Till Feb, 83% of TPLs in Maven haven't released the patch version for log4j RCE vulnerability.

OSS Health Profile--Osspert

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

THANKS & QUESTIONS

Email: chengwei001@e.ntu.edu.sg
Website:lcwj3.github.io

Presented by Liu Chengwei