PLUMBER: Boosting the Propagation of Vulnerability Fixes in the *npm* Ecosystem

Ying Wang, Peng Sun, Lin Pei, Yue Yu†, Chang Xu *Senior Member, IEEE*, Shing-Chi Cheung *Fellow, IEEE*, Hai Yu, and Zhiliang Zhu

Abstract—Vulnerabilities are known reported security threats that affect a large amount of packages in the *npm* ecosystem. To mitigate these security threats, the open-source community strongly suggests vulnerable packages to timely publish vulnerability fixes and recommends affected packages to update their dependencies. However, there are still serious lags in the propagation of vulnerability fixes in the ecosystem. In our preliminary study on the latest versions of 356,283 active *npm* packages, we found that 20.0% of them can still introduce vulnerabilities via direct or transitive dependencies although the involved vulnerable packages have already published fix versions for over a year. Prior study by Chinthanet et al. [1] lays the groundwork for research on how to mitigate propagation lags of vulnerability fixes in an ecosystem. They conducted an empirical investigation to identify lags that might occur between the vulnerable package release and its fixing release. They found that factors such as the branch upon which a fix landed and the severity of the vulnerability had a small effect on its propagation trajectory throughout the ecosystem. To ensure quick adoption and propagation of a release that contains the fix, they gave several actionable advice to developers and researchers. However, it is still an open question how to design an effective technique to accelerate the propagation of vulnerability fixes.
Motivated by this problem, in this paper, we conducted an empirical study to learn the scale of packages that block the propagation of vulnerability fixes in the ecosystem and investigate their evolution characteristics. Furthermore, we distilled the remediation strategies that have better effects on mitigating the fix propagation lags. Leveraging our empirical findings, we propose an ecosystem-level technique, PLUMBER, for deriving feasible remediation strategies to boost the propagation of vulnerability fixes. To precisely diagnose the causes of fix propagation blocking, PLUMBER models the vulnerability metadata, and *npm* dependency metadata and continuously monitors their evolution. By analyzing a full-picture of the ecosystem-level dependency graph and the corresponding fix propagation statuses, it derives remediation schemes for pivotal packages. In the schemes, PLUMBER provides customized remediation suggestions with vulnerability impact analysis to arouse package developers' awareness. We applied PLUMBER to generating 268 remediation reports for the identified pivotal packages, to evaluate its remediation effectiveness based on developers' feedback. Encouragingly, 47.4% our remediation reports received positive feedback from many well-known *npm* projects, such as `Tensorflow/tfjs`, `Ethers.js`, and `GoogleChrome/workbox`. Our reports have boosted the propagation of vulnerability fixes into 16,403 root packages through 92,469 dependency paths. On average, each remediated package version is receiving 72,678 downloads per week by the time of this work.

Index Terms—*npm* Ecosystem, Vulnerable Dependencies, Empirical Study

1 INTRODUCTION

Context. *npm* is the largest software registry for JavaScript programming language, which hosts over 1.8 million third-party packages as of January 2022. Unfortunately, *npm*'s fast growth comes with security risks. Vulnerabilities disclosed in third-party packages (i.e., *vulnerable packages*) is a growing concern for developers. An investigation [2] on a *npm* snapshot of November 2, 2017 indicates that, among 610,097 packages, 21.9% of them directly depend on vulnerable packages. Such vulnerability impact on the *npm* ecosystem can significantly increase if transitive dependencies are taken into account.

Vulnerabilities if adversely exploited can cause immense damages to the ecosystem [3, 4]. In fact, there are many examples of such incidents. A well-known example is the vulnerability *CVE-2021-28918* [5] in package `netmask`. It exposed private IP addresses and led to a variety of attacks such as malware delivery. Upon its disclosure, over 278,000 client projects were believed to be affected by the security vulnerability. To mitigate these security risks, the ecosystem strongly suggests vulnerable packages to timely publish vulnerability fixes and recommends affected packages to update their dependencies.

Problem and motivation. Our preliminary study on the 3,948 vulnerability reports for *npm* packages found that 60.6% of the involved packages had timely released fix versions after the vulnerabilities were discovered. To facili-

Ying Wang is affiliated with the Software College, Northeasthern University, the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, and the State Key Lab for Novel Software Technology, Nanjing University, China. E-mail: wangying@swc.neu.edu.cn.
Peng Sun, Lin Pei, and Hai Yu are with the Software College, Northeasthern University, China. E-mail:{sunpengneu, peilin_neu}@163.com, yuhai@mail.neu.edu.cn.
Yue Yu is with the National Laboratory for Parallel and Distributed Processing and College of Computer, National University of Defense Technology, Changsha, China. yuyue@nudt.edu.cn.
Chang Xu is with the State Key Lab for Novel Software Technology and the Department of Computer Science and Technology, Nanjing University, China. E-mail: changxu@nju.edu.cn.
Shing-Chi Cheung is with the Hong Kong University of Science and Technology, and Guangzhou HKUST Fok Ying Tung Research Institute, China. E-mail: scc@cse.ust.hk.
Zhiliang Zhu is with National Frontiers Science Center for Industrial Intelligence and Systems Optimization, Key Laboratory of Data Analytics and Optimization for Smart Industry, and Software College, Northeastern University, China. E-mail: zzl@mail.neu.edu.cn.
Yue Yu is the corresponding author.
Manuscript received 2022.

tate vulnerability fix adoption, the open-source community launched tools such as `npm audit` [6] and `Dependabot` [7] to alert the projects that directly or transitively depend on vulnerable package versions. However, there are still serious lags in the propagation of vulnerability fixes in the *npm* ecosystem. We investigated the latest versions of 356,283 active *npm* packages on the *npm* snapshot of August 1, 2021, and found that 20.0% of them can introduce vulnerabilities via dependencies, although the involved vulnerable packages have published fixed versions for over a year (see descriptions in Sec 4).

We made two observations on the causes for the lags in propagating vulnerability fixes:

- *Most packages remediated only the vulnerabilities in their highest major version trains, without transplanting the fixes to their earlier popular versions (i.e., backporting*[1]*). However, many downstream packages have difficulties in upgrading their dependencies to higher versions due to incompatibility issues or inactive transitive dependencies.* A recent study [8] on over a million *npm* packages indicated that only a small proportion (6.6%) of them have backporting practices. It implies that many affected downstream packages cannot timely benefit from the vulnerability fix versions.
- *Many packages did not realize that they were the cruxes of blocking fix propagation in the ecosystem, affecting a large amount of downstream packages.* Due to the lack of a full picture of the ecosystem-level dependency graph, package developers hardly realized that they were the pivotal factors preventing fixes from propagating to many of their downstream packages.

Figure 1 gives two illustrative examples. As shown in Figure 1(a), the latest version of package `browser-sync` transitively depends on a vulnerable package `engine.io`@3.5.0. However, `browser-sync`'s developers did not realize that the vulnerability *SNYK-JS-ENGINEIO-1056749* had already been remediated by `engine.io` since version 4.0.0. Besides, `browser-sync`'s direct dependency `socket.io` can also introduce this vulnerability fix via `engine.io`@4.0.0 since version `socket.io`@3.0.0. As a popular package in the *npm* ecosystem, `browser-sync`@2.27.4 was a package used by 2,417 projects. Unfortunately, the propagation of the fix in `engine.io` to these projects was blocked by this pivotal package (i.e., `browser-sync`@2.27.4).

Another example is shown in Figure 1(b). A vulnerability *SNYK-JS-WS-1296835* was transitively introduced to 146 active *npm* projects transitively via 1,308 dependency paths. All these paths share a common dependency chain: `graphql`@1.2.0 $\rightarrow$ `mqtt`@2.18.8 $\rightarrow$ `ws`@3.3.3. We observed that package `ws` had published the vulnerability fix versions {$\geq$ 3.4.5}, and these fix versions could be resolved by {`mqtt` $\geq$ 3.2.0}. However, the fix propagation was blocked by `graphql`, since this package was inactively maintained by developers (latest update was on Jan 2019).

Nevertheless, the large amount of affected downstream packages could quickly incorporate the vulnerability fix if `mqtt` backports to its lower version train 2.18.* (i.e., `mqtt` publishes a new release `mqtt`@2.18.*, in which it upgrades dependency `ws` from vulnerable version 3.3.3 to fix version 3.4.5). In this manner, `mqtt`@2.18.* with vulnerability fix would be automatically resolved by its specified version constraint ^ 2.3.0, propagating into 1,308 dependency paths. Unfortunately, the pivotal package `mqtt` was unaware of this fact. Until a backporting request was reported in `mqtt`#1306 [9], developers transplanted such a vulnerability fix to `mqtt`@2.18.9, in order to mitigate the security threats to its downstream projects.

To ease presentation, in this paper, we refer to the package that blocks the fix propagation on dependency paths as a *blocking package*. And the dependency path from a *blocking package* to a vulnerable package is referred to as a *blocking chain*. All the dependency paths passing through the same blocking chain would suffer from the same vulnerability issues. The above examples show that if the developers of pivotal blocking packages can be better informed of their impacts on vulnerability fix propagation, such situation can be significantly improved.

Related Work. Prior works have made important advances in remediating vulnerabilities, which focus on three aspects: (1) *Vulnerability impacts on the ecosystem* [2, 10, 10–16]; (2) *Reducing the false alarms for reporting vulnerable dependencies* [17–22]; (3) *Lags in vulnerable package updates* [1, 23–28] (see detailed discussion in Sec 7). Study by Chinthanet et al. [1] lays the groundwork for research on how to mitigate propagation lags of vulnerability fixes in an ecosystem. They conducted an empirical investigation to identify lags that might occur between the vulnerable package release and its fixing release. They found that factors such as the branch upon which a fix lands and the severity of the vulnerability had a small effect on its propagation trajectory throughout the ecosystem. To ensure quick adoption and propagation of a release that contains the fix, they gave actionable advice to developers and researchers: *developing better awareness mechanisms for quicker planning of a dependency update.* However, none of the existing work explores the characteristics of blocking packages and blocking chains in the ecosystem. It is still an open question how to design an effective technique

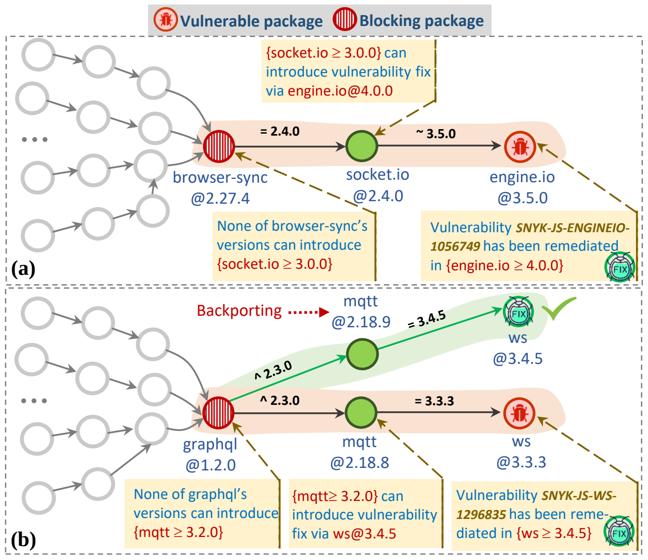

Vulnerable package
Blocking package
{socket.io ≥ 3.0.0} can introduce vulnerability fix via engine.io@4.0.0
= 2.4.0
~ 3.5.0
browser-sync @2.27.4
socket.io @2.4.0
engine.io @3.5.0
None of browser-sync's versions can introduce {socket.io ≥ 3.0.0}
Vulnerability SNYK-JS-ENGINEIO-1056749 has been remediated in {engine.io ≥ 4.0.0}
FIX
(a)
Backporting
mqtt @2.18.9
= 3.4.5
ws @3.4.5
^ 2.3.0
^ 2.3.0
= 3.3.3
graphql @1.2.0
mqtt @2.18.8
ws @3.3.3
None of graphql's versions can introduce {mqtt ≥ 3.2.0}
{mqtt≥ 3.2.0} can introduce vulnerability fix via ws@3.4.5
Vulnerability SNYK-JS-WS-1296835 has been remediated in {ws ≥ 3.4.5}
(b)

Fig. 1: Illustrative examples of blocking propagation of vulnerability fixes

1. Backporting refers to the process of applying a software update, typically a patch that fixes a bug or vulnerability to a lower version of the software [8]. Backporting enables users of lower releases to benefit from more recent vulnerability fixes.

to accelerate the propagation of vulnerability fixes.

Goal and challenges. To bridge the research gap, in this paper, we develop PLUMBER, an ecosystem-level technique that derives feasible remediation strategies for pivotal packages, boosting the propagation of vulnerability fixes. To achieve this goal, two challenges should be addressed:

- *Acquiring up-to-date vulnerability metadata and npm dependency metadata.* To precisely identify the pivotal packages that block the fix propagation, we should obtain the vulnerability metadata and ecosystem-level dependency metadata, and continuously updating these two pieces of metadata with package evolution in *npm*.
- *Understanding the evolution characteristics of blocking chains and their impacts on the vulnerability fix propagation.* To mitigate the fix propagation lags, we need to derive effective remediation strategies from the evolution characteristics of blocking chains and their impacts on the vulnerability fix propagation.

Approach and results. To address the two challenges, we first conducted a large-scale empirical study on the recent snapshots of the *npm* ecosystem, to learn: (RQ1) the scale of blocking packages and their impacts on other projects; (RQ2) the evolution characteristics of blocking chains on consecutive *npm* snapshots; (RQ3) the remediation strategies that have better effects on propagating vulnerability fixes. Leveraging our empirical findings, we propose PLUMBER, a novel technique to: (1) model the vulnerability and *npm* dependency metadata and incrementally update their evolution; (2) identify pivotal blocking chains that hinder the vulnerability fixes from propagating through dependency paths; (3) analyze the features of packages on blocking chains and customize remediation schemes. In the remediation scheme, PLUMBER arouses package developers' awareness of the lags in fix version updates and explains the benefits of their dependency updates to the whole ecosystem.

To evaluate the remediation effectiveness of PLUMBER, we conducted an ecosystem-level study. We applied PLUMBER to generating 268 remediation reports for the top influential blocking chains. 47.4% of our remediation reports received positive feedback from many well-known *npm* projects, such as `Tensorflow/tfjs` [29], `Ethers.js` [30], and `GoogleChrome/workbox` [31]. These reports indeed aroused the pivotal packages' awareness to perform dependency upgrades/backporting/migrations in order to remediate vulnerabilities, benefiting the whole ecosystem. Encouragingly, PLUMBER's generated reports boosted the propagation of vulnerability fixes into 16,403 active *npm* projects via 92,469 dependency paths. On average, each remediated package version received 72,678 downloads per week. The results indicate its wide remediation effectiveness and impact. It is thus necessary to deploy PLUMBER to continuously capture the pivotal blocking chains as well as suggest proper remediation strategies with impact analysis.

Contributions. In summary, this paper makes the following contributions:

- *A thorough empirical study.* We conducted the first empirical study to characterize the situations where packages are blocking the propagation of vulnerability fixes in an ecosystem. Our findings shed light on the bottlenecks of fix propagation due to package dependency in an ecosystem.
- *An ecosystem-level technique for facilitating vulnerability fix propagation.* We developed the PLUMBER tool to boost the propagation of vulnerability fixes in the *npm* ecosystem, via remediating pivotal blocking chains. In its generated reports, PLUMBER provides customized remediation suggestions with vulnerability impact analysis to arouse developers awareness.
- *A large-scale vulnerability fix propagation experiment.* We applied PLUMBER to generating remediation reports for dominant blocking chains. Our reports have boosted the propagation of vulnerability fixes into 16,403 root packages through 92,469 dependency paths. On average, each remediated package version received 72,678 downloads per week.
- *A reproduction package.* We provided a reproduction package at the PLUMBER website (**http://plumber-npm.com/**) for future research, which includes: (1) large-scale vulnerability and npm dependency metadata on recent ecosystem snapshots, (2) a benchmark containing 362 package updates performed by developers, which not only remediated blocking chains, but also enabled significant propagation effects of vulnerability fixes, (3) 268 remediation reports generated by PLUMBER, and (4) an available PLUMBER tool.

Paper organization. The remainder of this paper is organized as follows. Section 2 describes the necessary background information of vulnerabilities in the *npm* ecosystem. Section 3 introduces the terminology used throughout this paper. In Section 4, we thoroughly conduct an empirical study and present our main findings. Section 5 proposes our PLUMBER technique to boost the propagation of vulnerability fixes in the *npm* ecosystem. Section 6 conducts an ecosystem level study to understand the remediation challenges and the effectiveness of PLUMBER. Finally, Sections 7 and 8 discuss the threats to validity and related work, respectively, and Section 9 concludes this paper.

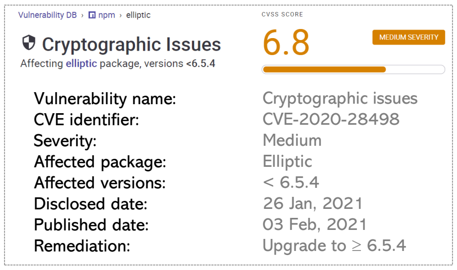
2 BACKGROUND

2.1 Disclosed vulnerabilities in the *npm* ecosystem

A *vulnerability* is a security threat[2] that exposes some package versions in a software ecosystem to attacks. We refer to the packages where such vulnerabilities disclosed as *vulnerable packages* and their affected versions as *vulnerable package versions*.

Nowadays, several recognized vulnerability databases such as *GitHub Advisory DB* [32], *Snyk Vulnerability DB* [33] and *NPM Security Advisories* [34] provide the continuous security monitoring service, which collects vulnerability reports of third-party packages to warn project developers against such vulnerable package versions. As shown in Figure 2, each vulnerability report contains information about: (1) affected package, (2) affected package versions, (3) unique CVE identifier, (4) severity levels (e.g., critical, high, medium and low based on CVSS score [35]), (5) dates when

2. Security threats can allow unauthorized actions or access to be performed. These actions are typically used to break through the system and violate its security policies.


Vulnerability DB › npm › elliptic

CVSS SCORE

Cryptographic Issues

6.8 MEDIUM SEVERITY

Affecting elliptic package, versions <6.5.4

Vulnerability name:	Cryptographic issues
CVE identifier:	CVE-2020-28498
Severity:	Medium
Affected package:	Elliptic
Affected versions:	< 6.5.4
Disclosed date:	26 Jan, 2021
Published date:	03 Feb, 2021
Remediation:	Upgrade to ≥ 6.5.4

Fig. 2: An illustrative vulnerability report for npm package

TABLE 1: Statistics of exposed vulnerabilities in the *npm* ecosystem

Database	#Vulnerability reports	#Vulnerability reports with fixes	Fixing ratio
GitHub Advisory DB[1]	1,990	1,115	56.0%
Snyk Vulnerability DB[2]	3,119	2,225	69.9%
NPM Security Advisories[3]	1,614	1,166	71.3%
Union set	3,948	2,391	60.6%

1. https://github.com/advisories; 2. https://snyk.io/vuln;
3. https://www.npmjs.com/advisories. †The three databases are maintained by different communities and contain overlapping reports.

it is disclosed and published, and (6) remediated versions (if available).

We crawled the vulnerability reports for *npm* packages published before 1 August 2021 from three databases and listed the statistics in Table 1. On average, 60.6% of vulnerable packages have remediated the disclosed vulnerabilities. However, our investigation on the latest versions of 356,283 active *npm* projects revealed that 71,320 of them (20%) are still directly or transitively depending on vulnerable package versions, although the vulnerability fix versions of these packages have already been released for over a year (see detailed descriptions in Sec 3). There are serious lags in the propagation of vulnerability fixes in the *npm* ecosystem.

2.2 npm package version constraints

A version of package p_a can explicitly declare a dependency relationship to another package p_b with a specified version constraint. For example, constraint < 2.1.0 defines the version range [0.0.0, 2.1.0), signifying that any version below 2.1.0 of p_b is allowed to be installed. In fact, *the highest available version within this range* will be selected for installation by the package manager.

npm customizes *node-semver* [36], a semantic version parser for Node.js. It inherits the version operators from semantic versioning (i.e., <, = , >), and also introduces advanced range syntax (i.e., ∗, ^, ∼), to allow flexible and backward compatible version control. For example, 2.1.∗ = ∼ 2.1.0, to allow the range [2.1.0, 2.2.0) of backward compatible package versions. While 2.∗.0 = ^ 2.0.0 denotes version range [2.0.0, 3.0.0). In this paper, we refer to the version ranges specified with ∗, ^, ∼ as *open version constraints*. Such range syntax allows flexibility, which enables projects to automatically deploy the new versions of their required packages that satisfy the specified version range (highest available version installation rule). In other words, new package versions with the vulnerability fixes can be automatically propagated into the downstream projects if open constraints are assigned.

2.3 Terminology

This section introduces the following concepts used throughout this paper. We let $p_i@v_m$ be a version of *npm* package (e.g., `electron@12.0.0`), where p_i denotes the unique package name released in *npm* and v_m denotes its version number. Since a package version $p_i@v_m$ usually directly and transitively depends on a collection of third-party packages, the referenced package versions and dependency relations among them form a package dependency graph $G_{i,m}$. We refer to $p_i@v_m$ as a *root project* in its corresponding package dependency graph $G_{i,m}$.

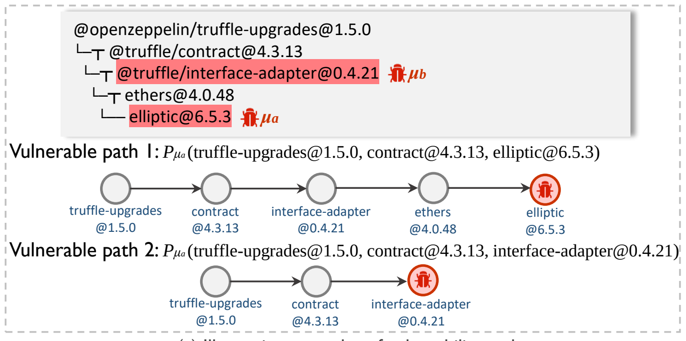
Definition 1. ***Vulnerable path:*** Suppose that $p_j@v_n$ is a vulnerable package version where vulnerability μ_t disclosed. In the package dependency graph $G_{i,m}$, we define a dependency path from root project $p_i@v_m$ to vulnerable package version $p_j@v_n$, $P_{\mu_t}(p_i@v_m, p_w@v_k, \cdots, p_j@v_n) = p_i@v_m \rightarrow p_w@v_k \rightarrow \cdots \rightarrow p_j@v_n$, as a *vulnerable path*. This dependency path is u_t-vulnerable if $p_i@v_m$ directly or transitively uses $p_j@v_n$, i.e., $p_i@v_m$ potentially invokes the vulnerable APIs defined in $p_j@v_n$. Note that there could be more than one vulnerable paths from a root project to a vulnerable package version. Remediated vulnerable paths can reduce the potential for accessing the disclosed vulnerable code, which minimizes the risk of being attacked through the vulnerabilities.

Definition 2. ***Safe version set:*** Suppose that vulnerable package p_j has remediated vulnerability μ_t in its versions $\{v_1, v_2, \cdots, v_u\}$, then we consider $p_j.S_{\mu_t} = \{v_1, v_2, \cdots, v_u\}$ as p_j's *safe version set* for μ_t. For each package on a vulnerable path $P_{\mu_t}(p_i@v_m, p_w@v_k, \cdots, p_j@v_n)$, we consider its safe version set as the versions that can directly or transitively depend on a safe version $v \in p_j.S_{\mu_t}$, or the versions that deprecated package p_j.

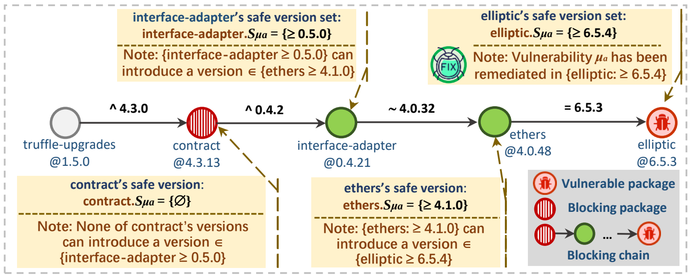
Definition 3. ***Blocking package:*** On vulnerable path $P_{\mu_t}(p_i@v_m, \cdots, p_w@v_k, \cdots, p_j@v_n)$, if package p_w's safe version set $p_w.S_{\mu_t} = \emptyset$, and all the packages depended by p_w have already published safe versions for vulnerability μ_t, we define p_w as a *blocking package*. We consider p_w blocks the propagation of vulnerability μ_t's fix through path $P_{\mu_t}(p_i@v_m, \cdots, p_w@v_k, \cdots, p_j@v_n)$, causing the lags of vulnerable package updates.

Definition 4. ***Blocking chain:*** Let p_w be a blocking package on vulnerable path $P_{\mu_t}(p_i@v_m, \cdots, p_w@v_k, p_e@v_g, \cdots, p_j@v_n)$. We define the subpath from blocking package to vulnerable package $B_{\mu_t}(p_w@v_k, p_e@v_g, \cdots, p_j@v_n) = p_w@v_k \rightarrow p_e@v_g \cdots \rightarrow p_j@v_n$ as a blocking chain. All the vulnerable paths passing through the blocking chain $B_{\mu_t}(p_w@v_k, p_e@v_g, \cdots, p_j@v_n)$ suffer from the same vulnerability issues.

Consider a dependency graph as shown in Figure 3(a). Vulnerabilities μ_a and μ_b disclosed in packages `interface-adapter`@0.4.21 and `elliptic`@6.5.3, respectively. To diagnose how root project `truffle-upgrades`@1.5.0 introduces μ_a and μ_b, we identify two vulnerable paths P_{μ_a}(`truffle-upgrades`@1.5.0, `contract`@4.3.13, `interface-adapter`@0.4.21, `ethers`@4.0.48, `elliptic`@6.5.3) and P_{μ_b}(`truffle-upgrades`@1.5.0, `contract`@4.3.13, `interface-adapter`@0.4.21). In Figure 3(b), we illustrate how to calculate the safe version set of each package


@openzeppelin/truffle-upgrades@1.5.0
@truffle/contract@4.3.13
@truffle/interface-adapter@0.4.21
ethers@4.0.48
elliptic@6.5.3
Vulnerable path 1: P_{μ_a}(truffle-upgrades@1.5.0, contract@4.3.13, elliptic@6.5.3)
Vulnerable path 2: P_{μ_a}(truffle-upgrades@1.5.0, contract@4.3.13, interface-adapter@0.4.21)

(a) Illustrative examples of vulnerability paths


interface-adapter's safe version set: interface-adapter.$S_{\mu a}$ = {≥ 0.5.0}
Note: {interface-adapter ≥ 0.5.0} can introduce a version ∈ {ethers ≥ 4.1.0}
elliptic's safe version set: elliptic.$S_{\mu a}$ = {≥ 6.5.4}
Note: Vulnerability μ_a has been remediated in {elliptic: ≥ 6.5.4}
contract's safe version: contract.$S_{\mu a}$ = {∅}
Note: None of contract's versions can introduce a version ∈ {interface-adapter ≥ 0.5.0}
ethers's safe version: ethers.$S_{\mu a}$ = {≥ 4.1.0}
Note: {ethers: ≥ 4.1.0} can introduce a version ∈ {elliptic ≥ 6.5.4}
Vulnerable package
Blocking package
Blocking chain

(b) Illustrative examples of safe version set, block package and blocking chain

Fig. 3: Examples for illustrating the introduced concepts

on the former path. Because vulnerability μ_a only affects {elliptic $\leq$ 6.5.3}, elliptic's safe version set is {$\geq$ 6.5.4}. However, although on the vulnerable path, ethers@4.0.48 depends on elliptic@6.5.3, since version 4.1.0, ethers upgraded elliptic to its safe version within {$\geq$ 6.5.4} to remediate vulnerability μ_a. Accordingly, ethers's safe version set for μ_a is {$\geq$ 4.1.0}. As an analogy, the safe version sets of remaining packages on this vulnerable path can be obtained. Notably, none of contract's versions can introduce μ_a's fix, while package interface-adapter referenced by contract has already resolved such a vulnerability issue in versions {$\geq$ 0.5.0}. We consider contract as a blocking package, which blocks the propagation of vulnerability μ_a's fix through path P_{μ_a}(truffle-upgrades@1.5.0, contract@4.3.13, interface -adapter@0.4.21, ethers@4.0.48, elliptic@6.5.3). In the *npm* ecosystem, all the vulnerable paths passing through the blocking chain B_{μ_a}(contract@4.3.13, interface-adapter@0.4.21, ethers@4.0.48, elliptic@6.5.3) suffer from the same vulnerability issues.

Definition 5. ***Remediating vulnerable path:*** Suppose that p_w is a blocking package on vulnerable path $P_{\mu_t}(p_i@v_m, \cdots, p_e@v_g, p_w@v_k, \cdots, p_j@v_n)$, and package p_j has remediated vulnerability μ_t in versions $p_j.S_{\mu_t}$ ($v_n \notin p_j.S_{\mu_t}$). To remediate such a vulnerable path, allowing the vulnerability μ_t's fix to be propagated into root project $p_i@v_m$, two conditions must be satisfied: (1) p_w's safe version set $p_w.S_{\mu_t} \neq \emptyset$ (via updating version of package on the vulnerable path); (2) one safe version $v \in p_w.S_{\mu_t}$ can be resolved by the version constraint c_w specified by package $p_e@v_g$ for p_w.

In Figure 3, we can tell that ethers's versions {$\geq$ 4.1.0} have remediated such a vulnerability by referencing elliptic's safe versions {$\geq$ 6.5.4}. However, {ethers $\geq$ 4.1.0} cannot be resolved by constraint $\sim$ 4.0.32 (i.e., [4.0.32, 4.1.0)) that is specified by its inactively maintained precursor interface-adapter. To resolve this vulnerability issue, truffle-upgrades's developer filed an issue report #1439 [37] to ethers.js. They requested ethers.js to backport to version 4.0.49, in which it upgraded elliptic to version 6.5.4 (vulnerability μ_a's fix version). As shown in Figure 4, elliptic@4.0.49 can be transitively referenced by constract @4.3.13. Therefore, constract @4.3.13 turn to be a safe version for μ_a. In this manner, such a vulnerability fix is propagated into root project truffle-upgrades@1.5.0, remediating the vulnerable path P_{μ_a}(truffle-upgrades@1.5.0, contract@4.3.13, interface -adapter@0.4.21, ethers@4.0.48, elliptic@6.5.3).

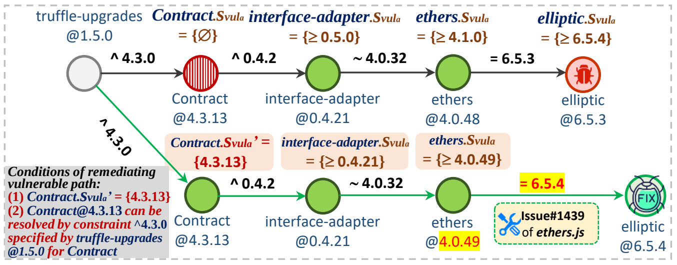

truffle-upgrades @1.5.0
Contract.S_{vul_a} = {∅}
interface-adapter.S_{vul_a} = {≥ 0.5.0}
ethers.S_{vul_a} = {≥ 4.1.0}
elliptic.S_{vul_a} = {≥ 6.5.4}
^4.3.0
^0.4.2
~4.0.32
= 6.5.3
Contract @4.3.13
interface-adapter @0.4.21
ethers @4.0.48
elliptic @6.5.3
Contract.S_{vul_a}' = {4.3.13}
interface-adapter.S_{vul_a} = {≥ 0.4.21}
ethers.S_{vul_a} = {≥ 4.0.49}
Conditions of remediating vulnerable path:
(1) Contract.S_{vul_a}' = {4.3.13}
(2) Contract@4.3.13 can be resolved by constraint ^4.3.0 specified by truffle-upgrades @1.5.0 for Contract
^4.3.0
^0.4.2
~4.0.32
= 6.5.4
Contract @4.3.13
interface-adapter @0.4.21
ethers @4.0.49
Issue#1439 of ethers.js
elliptic @6.5.4

Fig. 4: A real example of unblocking the vulnerable path

3 Empirical Study

In this section, we empirically investigate the scale of packages that block the propagation of vulnerability fixes in the *npm* ecosystem. By digging into evolution history of blocking chains, we characterize their common remediation patterns, which can shed light on boosting the the propagation of vulnerability fixes. Our analysis is guided by the following three research questions:

***RQ1* (Scale of blocking packages)**: *What is the scale of packages that block the propagation of vulnerability fixes in the npm ecosystem? To what extent do they affect other projects?*

Motivation: The concepts of *blocking package* and *blocking chain* are first introduced in this paper. Our motivation for RQ1 is to reveal the scale of *blocking packages* and *blocking chains* in the *npm* ecosystem. The investigation can help the package vendors understand to what extent they affect the propagation of vulnerability fixes.

Methodology: To answer RQ1, we first identify all the vulnerable paths, blocking packages and blocking chains in a recent snapshot of the *npm* ecosystem (August 1, 2021), and then analyze their scale and negative impacts.

***RQ2* (Evolution of blocking chains)**: *How do the blocking chains evolve in the npm ecosystem? How long have they existed in the npm ecosystem?*

Motivation: Our motivation for RQ2 is to inspect whether the blocking chains can be effectively remediated via the natural evolution of package versions. The investigation reveals the challenges of remediating blocking chains.

Methodology: To answer RQ2, we investigate the evolution trends of blocking chains across consecutive *npm* snapshots over the past year (from August 1, 2020 to August 1, 2021).

***RQ3* (Remediation patterns)**: *How were the blocking chains removed from the vulnerable paths? Are there common remediation patterns that can be distilled to facilitate the propagation of vulnerability fixes?*

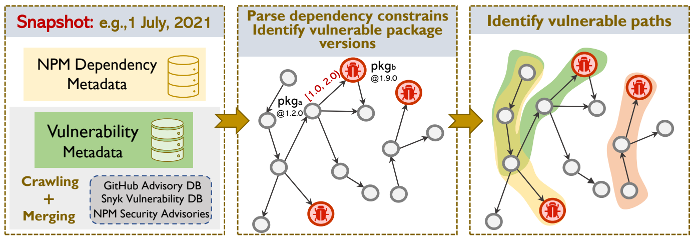


Fig. 5: An overview of VP model construction

Motivation: Our motivation for RQ3 is to investigate how the blocking chains or vulnerable paths were being remediated via package updates during evolution. The distilled common remediation patterns shed light on designing effective techniques to boost the propagation of vulnerability fixes in the *npm* ecosystem.

Methodology: To answer RQ3, we select a collection of real instances of blocking chains that have better remediation effects as subjects. Via replaying the package evolution history of these collected instances, we observe their package updates that can remediate blocking chains, their preconditions of propagating the vulnerability fixes, and the corresponding remediation costs.

In the following, we present our data collection procedure and study results in detail.

3.1 Vulnerability Propagation Model

To investigate our RQs, we construct a **V**ulnerability **P**ropagation (VP) model, which locates the vulnerable package versions in the ecosystem-level dependency graph to facilitate a holistic analysis of vulnerability affects. Figure 5 shows an overview of VP model construction. It first crawls a snapshot of *npm dependency metadata* and *vulnerability metadata*. By parsing the version constraints on package dependency relations and identifying vulnerable package versions, all the vulnerable paths can be identified.

Collecting vulnerability metadata. To collect more complete vulnerability metadata, we crawl vulnerability reports published before 1 August 2021, from three recognized vulnerability databases, i.e., *GitHub Advisory DB*, *Snyk Vulnerability DB* and *NPM Security Advisories*. Since the vulnerability information is usually described in plain text, for each vulnerability report, VP model needs to filter programming languages and identify its *unique identifier*, *affected package*, *affected package versions*, *fix versions* and *severity level*.

Collecting npm dependency metadata. *npm* public registry [38] provides RESTful APIs to obtain the metadata of all packages published in *npm*. The metadata describes the complete info of all releases of a given package. With the aid of such APIs, we capture the dependency metadata of *npm* ecosystem and described it as a directed graph $G = (V, E, C)$, to retrieve packages and resolve their dependencies. It consists of a set of package versions V and a set of dependency relations $E = \{p_i@v_a \rightarrow p_j@v_b | p_i@v_a, p_j@v_b \in V\}$. Each relation $p_i@v_a \rightarrow p_j@v_b \in E$ is determined by its corresponding dependency constraint $c(p_i@v_a, p_j) \in C$, where $c(p_i@v_a, p_j)$ denotes a version constraint specified by $p_i@v_a$ for p_j, and v_b is the highest version of p_j that satisfies the constraint (*npm*'s dependency resolution rule).

In our study, we analyze the dependency metadata of the *npm* snapshot on 1 August, 2021. To understand the status quo of vulnerability affects on the whole ecosystem, on such a *npm* snapshot, we filter out the legacy packages and only collect the latest versions of active packages and the package versions they directly or transitively depend on. Note that, we consider a package is active if it published new versions in the past year, following the assumption given by approaches [1, 8, 18].

Identifying vulnerable paths. By mapping the vulnerability metadata into *npm* dependency metadata $G = (V, E, C)$, in set V, we locate all the vulnerable package versions with detailed vulnerability info. We consider the latest versions of active packages in V as root nodes and vulnerable package versions in V as leaf nodes, to identify all the vulnerable paths via reachability analysis. However, a vulnerable package version may contain multiple vulnerabilities and each vulnerability corresponds to a safe version set. To precisely analyze the affects of each known vulnerability, when identifying vulnerable paths, a vulnerable package version with multiple vulnerabilities are distinguished as multiple leaf nodes. As such, a vulnerable path involves a unique vulnerability.

Statistics of VP model. Table 2 shows the statistics of metadata collected by VP model. To ease presentation, we denote ***root projects*** as the latest versions of active packages. Our vulnerability metadata includes 3,948 vulnerability reports, 2,391 of them (60.6%) have published vulnerability fix versions, especially for the ones with higher severity levels. For *npm* dependency metadata, VP model collected 521,413 latest versions of active packages which directly or transitively depend on 689,350 package versions. Among the 2,542,753 dependency relations among packages, it identified 1,065,723 vulnerable paths that are affected by the vulnerable packages having published fix versions. In our study, we considered the above 1,065,723 vulnerable paths that block the the propagation of vulnerability fixes as subjects to study their characteristics.

Data validation. The VP model took four months for three authors of this paper who had over two years vulnerability analysis experience to implement and test. The two experienced authors conducted daily manual validation of vulnerability metadata independently. Disagreements were reconciled with the third author joining the discussions. Specifically, we carefully checked two cases during the data validation process: (1) *Case 1*: Multiple databases share the vulnerability reports with the same CVE identifier. We only kept a vulnerability with unique CVE identifier in our dataset and filter out the duplicated ones. (2) *Case 2*: A vulnerability with unique CVE identifier may be disclosed in multiple packages. We recorded all the affected packages and their corresponding versions. The *Cohen's kappa* for their data validation is 0.92, indicating a near perfect inter-rater agreement [39]. For the eight disagreement cases, the information of shared vulnerability reports (e.g., affected package versions) were different from multiple vulnerability databases. Eventually, the third author checked the release notes of vulnerable package and kept our records to be consistent with that originated from package publishers. The three authors reached a consensus on our final dataset.

TABLE 2: Statistics of metadata collected by PLUMBER (on the recent *npm* snapshot of 1 August, 2021)

Vulnerability metadata			
#vulnerabilities **3,948**			
critical: **473**	high: **1,910**	medium: **1,374**	low: **191**
#vulnerabilities without fixes **1,557**			
critical: **96**	high: **677**	medium: **643**	low: **141**
#vulnerabilities with fixes **2,391**			
critical: **140**	high: **1,021**	medium: **1,078**	low: **152**
#vulnerable packages: **2,289**	*#vulnerable package versions*: **38,166**		
#vulnerabilities per package **1.7±2.9**			
***npm* dependency metadata**			
#root projects **356,283**			
#package versions directly or transitively depended by root projects **689,350**			
#dependency relations **2,542,753**			
#vulnerable paths **1,245,694**			
#vulnerable paths whose corresponding vulnerabilities have released fixes **1,065,723**			

Root projects denote the latest versions of active packages.

3.2 RQ1. Scale of blocking packages

3.2.1 Methodology.

To answer RQ1, we gather the statistics of vulnerable paths, blocking packages and blocking chains, to analyze their scale and negative impacts on the *npm* ecosystem. To achieve this, for each vulnerability μ_t, we first identify the corresponding safe versions S_{μ_t} of a vulnerable package p_j where it is disclosed. For each vulnerable path $P_{\mu_t}(p_i@v_m, \cdots, p_k@v_g, p_w@v_h, \cdots, p_j@v_n)$, starting from the vulnerable package p_j, we iteratively find each package's safe versions for μ_t, until a blocking package p_k whose safe version set $p_k.S_{\mu_t} = \emptyset$ is located. Accordingly, a blocking chain $B_{\mu_t}(p_k@v_g, p_w@v_h, \cdots, p_j@v_n)$ from blocking package to vulnerable package on each vulnerable path can be identified. In particular, we count the number of vulnerable paths and root projects that are affected by blocking chains, to investigate their influence on impeding propagation of vulnerability fixes.

3.2.2 Results.

Table 3 shows our investigation results of RQ1. In our study, we only focused on the 2,289 vulnerable packages that have published fix versions for their involved vulnerabilities. However, such vulnerability fix versions are not widely spread. 71,320 out of 356,283 (20.0%) active root projects in the *npm* ecosystem still directly or transitively depend on these vulnerable packages via 1,065,723 vulnerable paths. On average, each root project are affected by 4.4±7.5 vulnerabilities. Even worse, 48.5% of the widespread vulnerabilities are rated as high/critical severity level issues.

We identified 45,148 blocking packages in the *npm* ecosystem, which involved 358,422 blocking chains. All the published versions of blocking packages could only reference the vulnerable package versions through 983,336 vulnerable paths, causing the lags in propagation of vulnerability fixes into 68,413 root projects. In addition, we observed that 82,387 out of 1,065,723 vulnerable paths (7.7%) did not pass through blocking packages, since their root projects have ever introduced the vulnerability fix versions but then rolled back to vulnerable versions due to incompatibility issues. For instance, on a vulnerable path, root project `zos@2.4.3` introduces vulnerable package `lodash@4.17.10` through a direct dependency `truffle`. Whereas its previous version `zos@2.4.2` has ever transitively referenced the fix version `lodash @4.17.21` for vulnerability *CVE-2021-23337*, but then rolled it back because version 4.17.21 induced an incompatibility issue#1147 [40] in `truffle`.

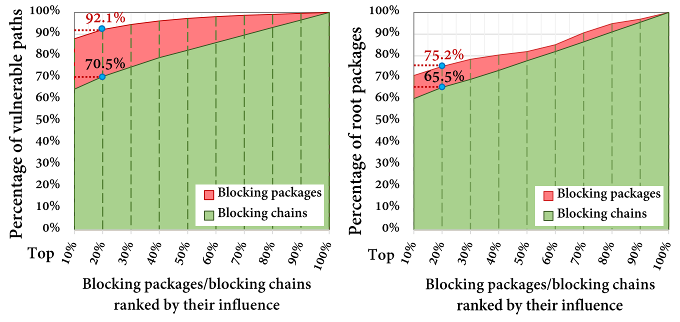


Fig. 6: The influence of blocking packages/blocking chains

Interestingly, we observed a clear centrality on both blocking packages and blocking chains. Figure 6 shows the influence of blocking packages/blocking chains on the whole ecosystem. The y-axis denotes the percentage of vulnerable paths/root projects that pass through the corresponding blocking packages/blocking chains presented in x-axis. While x-axis indicates the blocking packages/blocking chains that are ranked by the number of vulnerable paths/root projects affected by them. We can tell that the top 20% influential blocking packages hinder the vulnerability fixes from propagating through 92.1% vulnerable paths, affecting 75.2% root projects. Similarly, a small proportion (20%) of influential blocking chains can affect the vast majority (75.6%/65.5%) of vulnerable paths/root projects.

Table 4 lists the statistics of Top 10 influential blocking packages and blocking chains in the *npm* ecosystem. Take the most influential blocking package `swap-core` [41] as an example. It affects 41,278 vulnerable paths, transitively introducing two vulnerabilities into 395 root projects. In total, various versions of package `swap-core` induce 291 influential blocking chains. Among them, `swap-core@0.1.0` is the most popular version with 24,283 weekly downloads, which introduces a blocking chain `swap-core@0.1.0` → `hdwallet-provider@1.0.17` → `web3@1.2.1` → `web3-bzz@1.2.1` → `underscore@1.9.1` into 234 root projects via 6,503 vulnerable paths. Such influential blocking chains are the cruxes of impeding propagation of vulnerability fixes in the ecosystem. Accordingly, remediating the influential blocking chains can benefit the whole ecosystem, enabling the vulnerability fixes to be spread into a large amount of projects simultaneously.

Answer to RQ1: *In the recent snapshot of the npm ecosystem, there are 45,148 blocking packages and 358,422 blocking chains causing the lags in propagation of vulnerability fixes through 983,336 dependency paths. There is clear centrality on both influential blocking packages and blocking chains. 20% of blocking packages and blocking chains affect the vast majority of vulnerable paths.*
Implication: *Remediating the influential blocking chains enables the vulnerability fixes propagate into a large amount of root projects, which significantly benefits the whole ecosystem.*

TABLE 3: Statistics of investigation results in RQ1 (on the recent *npm* snapshot of 1 August, 2021)

Vulnerable paths	critical	high	medium	low
#root projects affected by the vulnerabilities **71,320**	**690**	**34,548**	**59,918**	**16,160**
#vulnerable paths referenced by each root project (Avg) **14.9±70.8**	**2.4±10.9**	**5.9±40.7**	**14.4±62.0**	**5.1±33.9**
#vulnerablities referenced by each root project (Avg) **4.4±7.5**	**1.1±0.3**	**3.0±5.3**	**3.0±3.4**	**1.9±1.7**
Blocking packages and blocking chains				
#blocking packages	**45,148**			
#blocking chains	**358,422**			
#vulnerable paths affected by blocking packages	**983,336**			
#root projects affected by blocking packages	**68,413**			
#vulnerable paths affected by each blocking package (Avg)	**21.8±344.3**			
#root projects affected by each blocking package (Avg)	**3.6±65.9**			
#vulnerable paths affected by each blocking chain (Avg)	**2.7±77.2**			
#root projects affected by each blocking chain (Avg)	**1.9±25.3**			

We only consider the vulnerable paths that are affected by the vulnerabilities with fixes

TABLE 4: Statistics of top 10 influential blocking packages and blocking chains (on the recent *npm* snapshot of August 1, 2021)

Top 10 influential blocking packages

Rank	Blocking package	#Induced blocking chains	#Affected paths	#Affected root projects	Vulnerabilities
1	`swap-core`	291	41,278	395	CVE-2021-23358, CVE-2020-28498
2	`glob-base`	1	33,656	17,683	CVE-2020-28469
3	`embark-utils`	769	28,937	81	CVE-2020-28499, CVE-2021-23358
4	`watchpack-chokidar2`	1	15,608	10,340	CVE-2020-28469
5	`@0x/subproviders`	928	11,164	130	CVE-2020-28500, CVE-2021-23337
6	`react-scripts`	57	8,368	4,166	CVE-2021-33587, CVE-2020-28469
7	`tree-sync`	2	8,278	1,306	npm:underscore.string:20170908
8	`koa-ejs-remote`	2	7,431	150	CVE-2020-28168
9	`glob-stream`	2	6,849	3,117	CVE-2020-28469
10	`@svgr/plugin-svgo`	3	6,601	5,134	CVE-2021-33587

Top 10 influential blocking chains

Rank	Blocking chain	#Affected paths	#Affected root projects	Vulnerabilities
1	`glob-base@0.3.0`→`glob-parent@2.0.0`	33,656	17,683	CVE-2020-28469
2	`watchpack-chokidar2@2.0.1`→`chokidar@2.1.8`→`glob-parent@3.1.0`	15,608	10,340	CVE-2020-28469
3	`@evolab/coreutils@4.2.0`→`url-parse@1.4.7`	6,276	76	CVE-2021-3664,CVE-2021-27515
4	`web3-eth-abi@1.2.6`→`ethers@4.0.0`→`elliptic@6.3.3`	6,087	856	CVE-2020-28498
5	`koa-ejs-remote@1.1.6`→`axios@0.18.1`	4,830	150	CVE-2020-28168
6	`Bwrapper@4.1.0`→`BVcheck@4.0.0`→`Bversion@3.1.0`→`Sregex@2.0.0`	4,312	1,533	SNYK-JS-SEMVERREGEX-1047770
7	`squeak@1.3.0`→`lpad-align@1.1.2`→`meow@3.7.0`→`Tnewlines@1.0.0`	4,252	1,486	CVE-2021-33623
8	`glob-stream@6.1.0`→`glob-parent@3.1.0`	4,210	2,255	CVE-2020-28469
9	`glob-watcher@5.0.5`→`chokidar@2.1.8`→`glob-parent@3.1.0`	3,870	1,535	CVE-2020-28469
10	`@S/plugin-svgo@5.5.0`→`svgo@1.3.2`→`css-select@2.1.0`→`css-what@3.4.2`	3,394	2,843	CVE-2021-33587

TABLE 5: Statistics of vulnerabilities disclosed in July 2020

#vulnerabilities with fixes: 61			
critical:1	high:25	medium:30	low: 5
#vulnerable packages: 35		*#vulnerable package versions*: 312	

TABLE 6: Statistics of *npm* snapshots (from August 1, 2020 to August 1, 2021)

Snapshots	#Root projects	#Dependency relations
s_1: *August 1, 2020*	220,819	1,576,547
s_2: *October 1, 2020*	238,833	1,596,379
s_3: *December 1, 2020*	253,339	1,613,350
s_4: *Febrary 1, 2021*	269,140	1,676,070
s_5: *April 1, 2021*	289,673	1,764,397
s_6: *June 1, 2021*	313,208	1,871,487
s_7: *August 1, 2021*	356,283	2,542,753

3.3 RQ2. Evolution of Blocking Chains

3.3.1 *Methodology.*

To answer RQ2, we construct a collection of consecutive snapshots of *npm* dependency metadata (denoted as s_1-s_7) over the past year (from August 1, 2020 to August 1, 2021), in order to investigate the evolution trends of blocking chains. The interval between two consecutive snapshots (s_{i-1} and s_i ($1 < i \leq 7$)) is taken as two months, since the *npm* projects in snapshot s_1 published new releases every 1.85 months on average. To observe the evolution of blocking chains ever since they appeared, for vulnerability metadata, we focus only on the vulnerabilities that were disclosed with fixes before the date of snapshot s_1.

The statistics of our mined vulnerabilities and seven *npm* snapshots are listed in Tables 5 and 6, respectively. On each snapshot, we consider the latest versions of active packages as root projects and collect all the package versions directly or transitively referenced by them. As previously presented, a root project is regarded as active if it published new versions in the past year (from the date of snapshot s_i). We identify all the blocking chains and vulnerable paths induced by the above vulnerabilities in snapshots s_1-s_7, and then investigate their evolutionary trends.

Specifically, we perform two tasks in the investigation: (a) by comparing the statistics of snapshot s_i ($1 < i \leq 7$) with that of s_1, we investigated the scale of blocking chains, vulnerable paths and affected root projects in s_1 that have been remediated by developers during a year of evolution. (b) by comparing the statistics of two consecutive snapshots s_{i-1} and s_i, we counted the number of remediated blocking chains, vulnerable paths and affected root projects during every two months interval. Besides, we also concern whether s_i introduced new cases compared with s_{i-1}. We analyze the three cases as follows:

- *Remaining cases*: Since the package versions keep evolving across s_1-s_7, we consider two blocking chains/vulnerable paths in s_m and s_n ($m \leq n$) as the equivalent ones, if they share the same packages in the same sequence but s_n involves updated versions. Similarly, the root projects in s_m are regarded equivalent with their updated versions in s_n. Such equivalent cases in s_n are the remaining ones

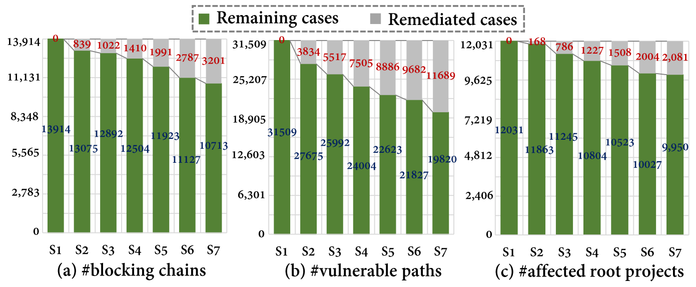


Fig. 7: Evolutionary trend analysis (comparing s_i with s_1)

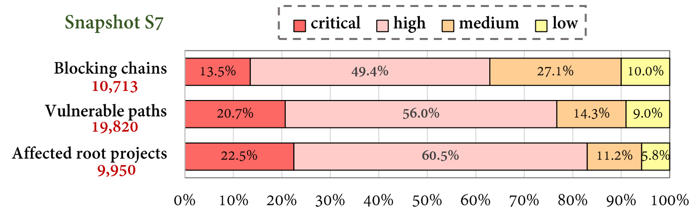


Fig. 8: Severity of vulnerabilities that affect the remaining cases

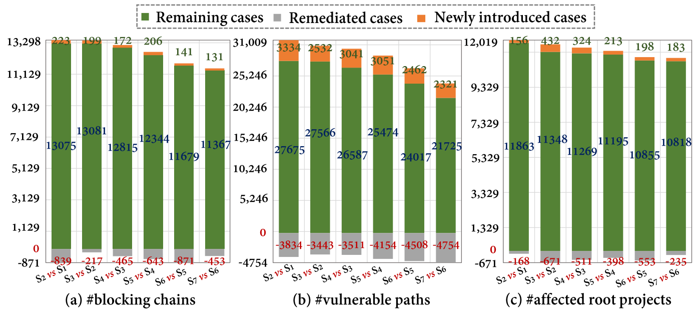


Fig. 9: Evolutionary trend analysis (comparing s_i with s_{i-1})

of s_m that have not been remediated yet.

- *Remediated/newly introduced cases*: Suppose BC_k, VP_k and ARP_k are the sets of blocking chains, vulnerable paths and affected root projects identified in snapshot s_k, respectively. We then consider $BC_m - BC_n$, $VP_m - VP_n$ and $ARP_m - ARP_n$ as the cases that has been remediated during the interval between snapshots s_m and s_n ($m \leq n$). While $BC_n - BC_m$, $VP_n - VP_m$ and $ARP_n - ARP_m$ are regarded as the blocking chains, vulnerable paths and affected root projects newly introduced into snapshot s_n, since the date of snapshot s_m.

3.3.2 Results.

Figure 7 shows the scale of remediated/remaining blocking chains, vulnerable paths and affected root projects in s_i ($1 < i \leq 7$), comparing with the initial snapshot s_1. After a year of evolution, among the 13,914 blocking chains identified in s_1, 10,713 of them (77.0%) still exist in the *npm* ecosystem. Only 37.1% (11,689/31,509) of vulnerable paths have been remediated, and the number of root projects affected by these blocking chains fall by 17.3% (2,081/12,031). Figure 8 depicts the severity levels of vulnerabilities that affect the remaining cases in the latest snapshot s_7. It is worth noting that 76.7% of remaining vulnerable paths and 83.0% root projects still reference the high/critical severity level vulnerabilities. Such low remediation rates of vulnerable paths and affected root projects reveal that the pivotal blocking chains do not arouse developers' awareness or have challenges to be remediated.

Figure 9 shows the statistics of remediated/newly introduced blocking chains, vulnerable paths and affected root projects between two consecutive snapshots. As of the vulnerable packages are disclosed in July 2020, the number of new blocking chains (179 on average) emerge in each snapshot signicantly lower than the remediated ones (581 on average). However, on average, each snapshot newly introduced 2,790 vulnerable paths and 251 affected root projects via referencing blocking packages as the projects evolve. The phenomenon occurs because some popular blocking packages (i.e., high downloads) can be easily adopted as dependencies during evolution, even if they would transitively introduce vulnerable packages into projects. For instance, glob-base@0.3.0 is influential in the ecosystem (4,755,881 weekly downloads), which depends on a vulnerable package glob-parent@$\leq$ 5.1.2 (*CVE-2020-28469*). We consider glob-base as a blocking package since all its published versions cannot reference the vulnerability's fix versions (safe version set is $\emptyset$ for *CVE-2020-28469* on the vulnerable paths). Between the interval of snapshots s_1 and s_7, such a popular version glob-base@0.3.0 has been newly introduced into 143 root projects via 843 dependency paths. The results indicate that remediating pivotal blocking chains cannot only boost the propagation of vulnerability fixes into the existing vulnerable paths but also reduce the risk of introducing new vulnerable paths into root projects.

Answer to RQ2: *In the npm snapshot of August 1, 2020, 77.0% of identified blocking chains still exist after a year of evolution. During this period, the number of vulnerable paths and root projects affected by these blocking chains fall by 37.1% and 17.3%. 9,808 blocking chains bundled with higher level vulnerabilities are still referenced by 9,904 active root projects through 17,612 vulnerable paths.*
Implication: *The low remediation rates of vulnerable paths and affected root projects indicate that developers do not realize the influence of blocking chains or have challenges to remediate them.*

3.4 RQ3. Remediation Patterns

3.4.1 Methodology.

To answer RQ3, we empirically investigated how the blocking chains or vulnerable paths were being remediated via package updates during the evolution across snapshots s_1-s_7 (collected in RQ2) and distilled common remediation patterns. It is worth noting that these package updates are not necessarily performed for remediating vulnerabilities. Some of the reactions might be preventive maintenance activities for package dependencies, but resulted in the propagation of vulnerability fixes.

Subject selection. We focus on two types of blocking chains that have been remediated via package updates with significant effects of propagating vulnerability fixes:

- ***Type A.*** *The blocking chains existed in snapshots s_1-s_{i-1} ($1 < i \leq 7$), while were remediated in snapshot s_i.* Note that, as the remediation of these blocking chains, all the vulnerable paths passing through them can introduce the vulnerability fixes.
- ***Type B.*** *The blocking chains that existed in snapshots s_1-s_7, while the number of vulnerable paths affected by them significantly reduced during evolution.* On average, the blocking chains were passed through by 59.4±13.4 vulnerable

TABLE 7: Statistics of the selected blocking chain instances

126 Type A blocking chain instances			
#blocking packages	#vulnerable packages	#root projects	#vulnerable paths (Avg)
38	28	542	34.4±32.7

243 Type B blocking chain instances			
#blocking packages	#vulnerable packages	#root projects	#vulnerable paths (Avg)
62	22	604	92.8±12.5

paths in s_1 and had fallen by 46.9%±10.3% in s_7. In our study, we consider the blocking chains as *Type B* cases if the number of their involved vulnerable paths is more than 59.4 in s_1 (above average value), and have fallen by over 46.9% in s_7 (above average value).

The remediation patterns of these two types of blocking chains can help us understand how to facilitate the propagation of vulnerability fixes in the *npm* ecosystem. For each type of blocking chain, we only selected the instances involving high/critical severity level vulnerabilities, which are worthy of developers' attentions. Table 7 provides the demographic information of our selected blocking chain instances. As we can see, the selected 369 blocking chains involve diverse blocking packages and vulnerable packages. In total, they affect 887 root projects via 26,884 vulnerable paths. We distill common features in their vulnerability remediation and summarize their taxonomy of remediation patterns.

Data analysis. Specifically, we performed three tasks to analyze the characteristics and remediation patterns of our selected blocking chain instances:

- *Task 1.* For each blocking chain instance, we investigate its three aspects of characteristics:
  (a) *whether the packages on the blocking chain are active* (i.e., publishing new versions in the past year from the date of snapshot). *Characteristic a* helps us understand which packages can update their dependencies to remediate such a blocking chain.
  (b) *whether the packages on the blocking chain are specified with open version constraints* (i.e., the range syntax such as 2.1.$*$, $\sim$ 2.1.0 and $^\wedge$ 2.1.0, enables new package versions that satisfy its specified constraints to be installed). With the aid of *Characteristic b*, we can observe how vulnerability fixes are propagated into blocking chains/vulnerable paths.
  (c) *whether the packages need to upgrade/downgrade their direct dependencies' major versions, in order to introduce the concerned vulnerability fixes*. According to the semantic versioning strategy [42], developers are suggested to increase their projects' major version number, if they make incompatible API changes. As such, upgrading/downgrading dependencies' major versions would take project developers' more efforts on modifying or testing code. We can estimate the remediation costs of blocking chains/vulnerable paths via *Characteristic c*.
- *Task 2.* For each blocking chain of *Type A*, we first identified the snapshot s_i ($1 < i \leq 7$) in which it had been remediated. For each package on such a blocking chain, we collected all its versions publishing between the interval of snapshots s_1 and s_i. Based on the chronological order of these version releases, we iteratively replayed the version updates of each package on this blocking chain, to check which version update enabled the remediation of blocking chain (i.e., blocking package can use the vulnerability fix version). Finally, we recorded the duration (#days) from the date of the blocking chain being remediated to the date of snapshot s_1.
- *Task 3.* For each blocking chain of *Type B*, we collected the vulnerable paths passing through it on snapshot s_1 but were remediated on snapshot s_i ($1 < i \leq 7$). Furthermore, for each package on such remediated vulnerable paths, we collected all its versions publishing between the interval of snapshots s_1 and s_i. Based on the chronological order of these version releases, we iteratively replayed the version updates of each package on the selected vulnerable paths, and checked which version update enabled the remediation of vulnerable paths (i.e., root project can use the vulnerability fix version). Finally, we recorded the duration (#days) from the date of the vulnerable path being remediated to the date of snapshot s_1.

Similar to many existing empirical studies [43, 44], we followed an open coding procedure [45] to inductively categorize the common remediation patterns in a bottom-up manner, which involves two labeling processes:

- *Labeling process 1:* Initially, two authors performed an in-depth analysis on the above characteristics of blocking chains and their involved vulnerable paths, to understand how they were remediated as the packages evolved. They first analyzed 50% of both *Types A* and *B* blocking chains independently and marked those unclear or insufficient categories. They then discussed and adjusted their category tags during meetings, with the help of a third author to resolve conflicts. In this way, we constructed the pilot taxonomy.
- *Labeling process 2:* Next, the first two authors continued to label remediation pattern of the remaining 50% of blocking chains and iteratively refined the results as well as the labeling strategy. Conflicts were reconciled with the third author joining the discussions. We adjusted the pilot taxonomy and obtained the final result.

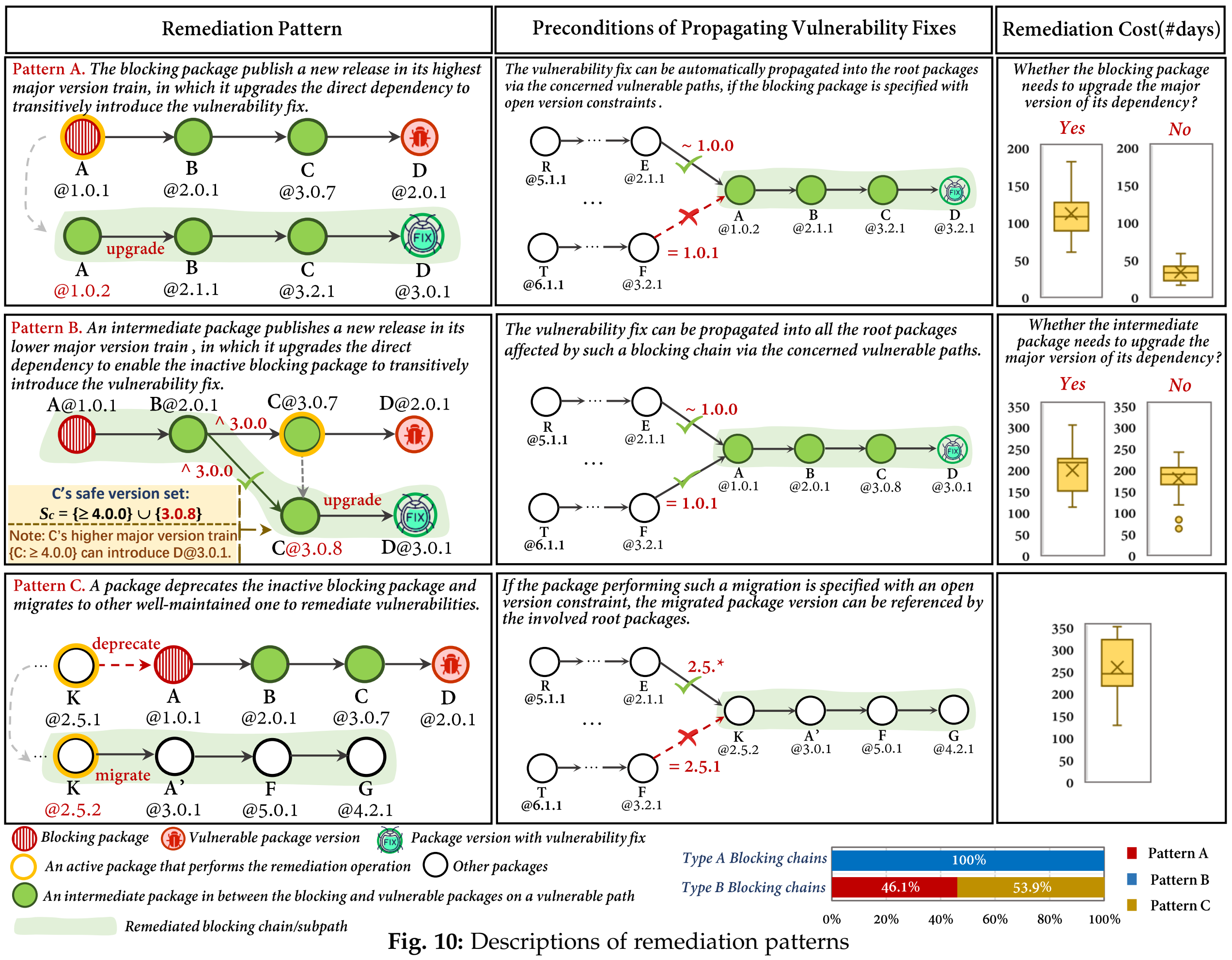
Specifically, the number of conflicts in the two labeling processes are 31 and 26, respectively. Since the labeling involves much manual work, errors could be induced in any of the three tasks when analyzing block chain instances, causing labeling conflicts between the two authors. The *Cohen's kappa* for our two labeling processes are 0.79 and 0.80, respectively, indicating a substantial inter-rater agreement [39].

3.4.2 Results.

Via replaying the package evolution history of the collected blocking chain instances across snapshots $s_1 - s_7$, we observed three common remediation patterns. Figure 10 summarizes the taxonomy of remediation patterns, their preconditions of propagating the vulnerability fixes, and the corresponding costs. In the following, we discuss each remediation pattern in detail. To ease presentation, we refer to the package in between the blocking and vulnerable packages on a blocking chain as an *intermediate package*.

Remediation pattern A. *The blocking package publishes a new release in its highest major version train, in which it upgrades the direct dependency to transitively introduce the vulnerability fix (46.1% of* Type B *blocking chain instances).* 46.1% of *Type B* blocking chain instances were remediated via *Pattern A*. Among the 243 *Type B* blocking chain instances, 112 of them are induced by active blocking packages. The lags in

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3243262


Remediation Pattern | Preconditions of Propagating Vulnerability Fixes | Remediation Cost(#days)

Pattern A. The blocking package publish a new release in its highest major version train, in which it upgrades the direct dependency to transitively introduce the vulnerability fix.

The vulnerability fix can be automatically propagated into the root packages via the concerned vulnerable paths, if the blocking package is specified with open version constraints.

Whether the blocking package needs to upgrade the major version of its dependency?

Pattern B. An intermediate package publishes a new release in its lower major version train, in which it upgrades the direct dependency to enable the inactive blocking package to transitively introduce the vulnerability fix.

C's safe version set: $S_C = \{\geq 4.0.0\} \cup \{3.0.8\}$

Note: C's higher major version train {C: ≥ 4.0.0} can introduce D@3.0.1.

The vulnerability fix can be propagated into all the root packages affected by such a blocking chain via the concerned vulnerable paths.

Whether the intermediate package needs to upgrade the major version of its dependency?

Pattern C. A package deprecates the inactive blocking package and migrates to other well-maintained one to remediate vulnerabilities.

If the package performing such a migration is specified with an open version constraint, the migrated package version can be referenced by the involved root packages.

Blocking package; Vulnerable package version; Package version with vulnerability fix; An active package that performs the remediation operation; Other packages; An intermediate package in between the blocking and vulnerable packages on a vulnerable path; Remediated blocking chain/subpath

	Pattern A	Pattern B	Pattern C
Type A Blocking chains		100%	
Type B Blocking chains	46.1%		53.9%

Fig. 10: Descriptions of remediation patterns

dependency updates of such blocking packages are the key causes of hindering root projects from introducing vulnerability fixes. As such, they finally remediated vulnerabilities in their highest available versions, by upgrading the direct dependencies to the safe versions.

In this scenario, for the vulnerable paths on which the blocking packages were specified with open version constraints (i.e., the remediated higher versions can be resolved by the constraints), vulnerability fixes could be automatically propagated into the root projects. Among the 112 instances that adopt remediation *Pattern A*, 34 of them needed to upgrade the major versions of dependencies with an average duration of 107.6 days. While the remaining 78 instances only took 33.5 days (Avg) for the remediation.

Remediation pattern B. *An intermediate package publishes a new release in its lower major version train, in which it upgrades the direct dependency to enable the inactive blocking package to transitively introduce a vulnerability fix (100% of* Type A *blocking chain instances).* 100% of *Type A* blocking chain instances are remediated via *Pattern B*. These instances share a common feature: the involved blocking packages are inactively maintained by developers (86.5% of them have not published new releases for over three years), which can hardly remediate the concerned vulnerabilities. To allow the inactive blocking package to transitively reference the vulnerability fix, an intermediate package published a new release in its lower major version train, in which it upgraded its direct dependency to safe versions. In studies [8, 46], the above process is referred as *backporting*. Such backporting practices can bring the benefit of vulnerability fixes to the substantial number of downstream packages that have to depend on the lower major version train of intermediate packages due to incompatibility issues.

By further investigation, we can tell that in our collected *Type A* instances, such intermediate packages have two shared characteristics: (a) they had remediated the vulnerabilities in their higher major version train while not instantly realized the vulnerability affects on downstream packages induced by their lower major version train; (b) they actively maintained package releases in multiple major version trains; (c) they are specified with open version constraints on the blocking chains (i.e., the remediated versions can be automatically resolved by the constraints).

With the aid of *Pattern B*, the vulnerability fix can be automatically propagated into all the root projects affected by such a blocking chain. However, these intermediate packages took more time to realize the vulnerability affects on other packages induced by their lower versions. In addition, they spent much effort on code changes and testing for the cases that needed to upgrade the major versions of dependencies. On average, *Pattern B* took 195.1 days for the remediation.

Remediation pattern C. *A package deprecates the inactive blocking package and migrates to other well-maintained one to remediate vulnerabilities (53.9% of Type B blocking chain instances).* 53.9% of *Type B* blocking chain instances use

Pattern C to remediate vulnerabilities. These *Type B* instances share two characteristics: (a) the blocking chains are induced by inactive blocking packages (on average they have not published new releases for 2.6 years); (b) the intermediate packages on such blocking chains are either inactively maintained or specified with locked version constraints. Consequently, even if the vulnerabilities can be remediated by intermediate packages, the remediated package versions cannot be propagated via such locked version constraints. To address the vulnerability issues, the packages affected by the above 131 intractable *Type B* blocking chains, finally deprecated the inactive blocking package and migrated to other well-maintained ones.

If the packages that deprecate the blocking ones, are popular in the *npm* ecosystem and specified with open version constraints, their remediated versions can be automatically propagated into many root projects. Generally, package migration needs more efforts than package upgrades. To achieve this, developers have to identify high-quality packages with similar functionalities to replace the inactive blocking ones, and spend efforts to modify and test code as well. On average, *Pattern C* took 251.5 days for the remediation.

In general, remediation costs token by the three patterns follow: *Pattern A* < *Pattern B* < *Pattern C*. *Patterns B* and *C* bring more burden to package developers. Based on our empirical findings, for the blocking chains induced by active blocking packages, *Pattern A* is highly recommended to remediate vulnerabilities. For the blocking chains induced by inactive blocking packages, in the cases that the intermediate packages on vulnerable paths are specified with open version constraints, remediation *Pattern B* would be feasible if they can backport to the lower version trains. Otherwise, the packages affected by such blocking chains can adopt *Pattern C* to deprecate the inactive blocking packages.

To remediate vulnerabilities, many prior works have been proposed to suggest practitioners to timely upgrade vulnerable dependencies [1,23–28], backport to lower versions [8], and migrate the outdated dependencies [1,47]. By contrast, the three distilled remediation patterns in our study, synthesize three factors: (1) the above three dependency update operations; (2) characteristics of blocking chains; (3) preconditions of propagating the vulnerability fixes. Our empirical findings shed light on how to update pivotal packages' version constraints based on semantic versioning mechanism, for mitigating the lags in propagating vulnerability fixes in an ecosystem.

Answer to RQ3: *We distilled three common remediation patterns and their preconditions of propagating the vulnerability fixes. For the blocking chains induced by active blocking packages, such blocking packages can remediate vulnerabilities in their highest available version (Pattern A). For the blocking chains induced by inactive blocking packages, the intermediate packages can remediate vulnerabilities in their lower major version trains to enable the inactive blocking packages to transitively introduce vulnerability fixes (Pattern B). Besides, a package affected by a blocking chain can also deprecate the inactive blocking package and migrate to other well-maintained one to remediate vulnerabilities (Pattern C).*
Implication: *Combining the three common patterns to remediate pivotal blocking chains can significantly boost the propagation of vulnerability fixes in the npm ecosystem.*

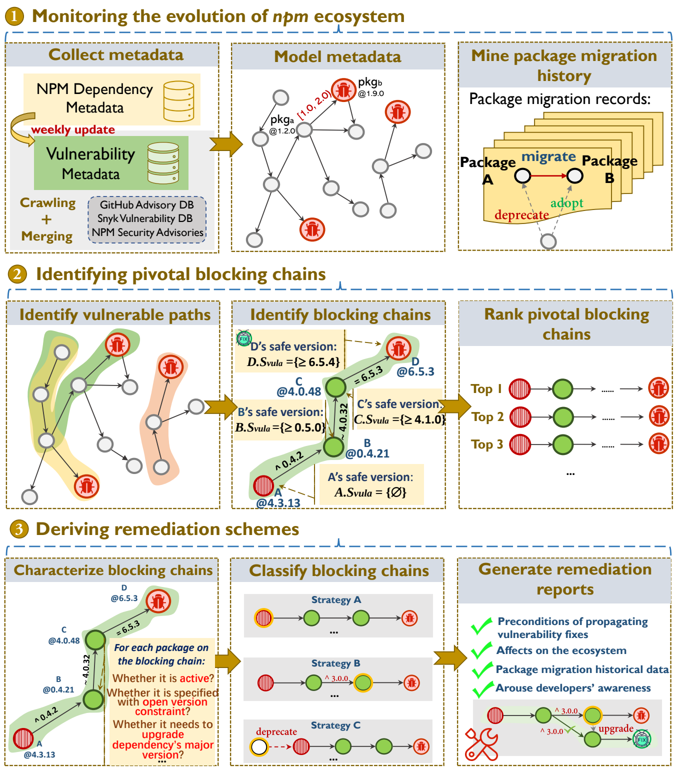

1 Monitoring the evolution of npm ecosystem
Collect metadata
NPM Dependency Metadata
weekly update
Vulnerability Metadata
Crawling + Merging
GitHub Advisory DB
Snyk Vulnerability DB
NPM Security Advisories
Model metadata
pkga @1.2.0
[1.0, 2.0)
pkgb @1.9.0
Mine package migration history
Package migration records:
Package A
migrate
Package B
deprecate
adopt
2 Identifying pivotal blocking chains
Identify vulnerable paths
Identify blocking chains
D's safe version: D.Svula ={≥ 6.5.4}
D @6.5.3
=6.5.3
C @4.0.48
C's safe version: C.Svula ={≥ 4.1.0}
~4.0.32
B's safe version: B.Svula ={≥ 0.5.0}
B @0.4.21
^0.4.2
A @4.3.13
A's safe version: A.Svula = {∅}
Rank pivotal blocking chains
Top 1
Top 2
Top 3
...
3 Deriving remediation schemes
Characterize blocking chains
For each package on the blocking chain:
Whether it is active?
Whether it is specified with open version constraint?
Whether it needs to upgrade dependency's major version?
...
Classify blocking chains
Strategy A
Strategy B
^3.0.0
Strategy C
deprecate
Generate remediation reports
Preconditions of propagating vulnerability fixes
Affects on the ecosystem
Package migration historical data
Arouse developers' awareness
^3.0.0
upgrade
FIX

Fig. 11: Architecture of PLUMBER

4 THE PLUMBER APPROACH

Inspired by our empirical findings, we propose PLUMBER, a novel technique to boost the propagation of vulnerability fixes in the *npm* ecosystem, via remediating pivotal blocking chains. Figure 11 gives the overall architecture of PLUMBER. Its core idea is to (1) model the vulnerability and *npm* dependency metadata and continuously monitor their evolution; (2) identify which pivotal blocking chains hinder the vulnerability fixes from propagating through vulnerable paths (based on findings in RQ1); (3) analyze the features of pivotal blocking chains and derive their remediation schemes combining the three patterns learnt from our empirical findings in RQ3. In the remediation scheme, PLUMBER can provide detailed information for package developers to arouse their awareness of the lags in fix version updates and the vulnerability affects on the *npm* ecosystem (based on findings in RQ2).

4.1 Monitoring the evolution of the npm ecosystem

To precisely capture the propagation of vulnerability fixes in the whole ecosystem, PLUMBER continuously collects vulnerability metadata and dependency metadata among *npm* packages. By comparing two contiguous snapshots of *npm* dependency metadata, we can also mine the migration history of packages, in order to provide clues for deriving remediation strategies (as we will show later in Section 4.3.1).

4.1.1 Modeling and collecting metadata

Metadata model. We model the metadata of *npm* snapshot s_i as a 2-tuple $\mathcal{M}(s_i) = (dm_i, G_i)$:

- $dm_i = \{\mu_1, \mu_2, \cdots, \mu_n\}$ is a collection of disclosed vulnerabilities, where $\mu_k = (Id_k, sl_k, p_k, ver_k)$. Fields Id_k and sl_k

denote unique identifier and severity level of μ_k. Fields p_k and $ver_k = \{v_1, v_2, \cdots, v_n\}$ represent the package name and its published versions affected by such a vulnerability.

- $G_i = (V_i, E_i, C_i)$ is a directed graph depicting the dependency metadata of *npm* snapshot s_i. V_i denotes a collection of package versions published on snapshot s_i. $E_i = \{p_k@v_a \rightarrow p_u@v_b | p_k@v_a, p_u@v_b \in V_i\}$ represents a set of dependency relations among the package versions in V_i. Each relation $p_k@v_a \rightarrow p_u@v_b \in E_i$ is determined by its corresponding dependency constraint $c(p_k@v_a, p_u) \in C_i$, where $c(p_k@v_a, p_u)$ denotes a version constraint specified by $p_k@v_a$ for p_u, and v_b is the highest version of p_u on snapshot s_i that satisfies the constraint (*npm*'s dependency resolution rule).

Metadata collection. As described in Section 3.1, PLUMBER crawls the above vulnerability metadata dm_i from three representative databases *GitHub Advisory DB*, *Snyk Vulnerability DB* and *NPM Security Advisories*. While the dependency metadata G_i is crawled with the aid of RESTful APIs provided by *npm* public registry [38].

PLUMBER weekly updates the metadata model $\mathcal{M}(S_i)$ to continuously monitor the evolution of vulnerability and *npm* dependency metadata. To analyze the propagation of vulnerability fixes in the *npm* ecosystem, dm_i only records the vulnerabilities that have been remediated with fixes.

4.1.2 Mining package migration history

During evolution, PLUMBER compares the dependency graphs G on two contiguous *npm* snapshots s_i and s_{i+1} to identify package migration records, which can provide clues for deriving remediation *Strategy C*. We formally define a package migration record as follows.

Definition 6. ***Package migration record:*** If a project deprecates package p_m and adopts p_n as a substitute, we define $p_m \Rightarrow p_n$ as a migration record, where p_m is *source package* while p_n is the *target one* in the migration.

Combining library migration mining approach [47] with our collected dependency metadata, PLUMBER collects a package migration record based on two criteria:

- If $p_k@v_a \rightarrow p_u@v_b \in E_i$ exists in snapshot s_i, while in snapshot s_{i+1}, $p_k@(v_a$+1) deprecates package p_u and depends on a new package p_t, PLUMBER records <p_u, p_t> as a dependency change pair. PLUMBER considers $p_u \Rightarrow p_t$ as possible migration record, iff such a dependency change pair appears over M times in different projects during evolution. PLUMBER takes the default value of $M = 5$, which can effectively reduce noise via our experiment validation (see discussions in Section 5.3). We leave its adaptive tuning to future work.
- For each possible migration record $p_u \Rightarrow p_t$, to guarantee its reliability, PLUMBER searches for its corresponding migration commits on GitHub using keywords "replace p_u" OR "deprecate p_u" OR "remove p_u" OR "switch p_u" OR "migrate p_u" OR "delete p_u". PLUMBER determines $p_u \Rightarrow p_t$ as a certain migration record, iff the newly added package p_t can be identified in the returned code commits.

It is worth noting that for a source package, PLUMBER could find multiple target packages in the migration records. When deriving remediation *Strategy C*, all possible suggestions can be provided for developers to migrate problematic packages.

Algorithm 1: Identifying blocking chains

```
Input: $P_{\mu_t}(p_k@v_m, p_w@v_r, \cdots, p_u@v_n)$, $V_i$ and $ver_t$
Output: $B_{\mu_t}$
1 $Queue \leftarrow ReverselyTraverse(P_{\mu_t})$;
2 $p_b \leftarrow Queue.pop()$;
3 $p_b.S_{\mu_t} \leftarrow PublishedV(V_i, p_b) - ver_t$;
4 while $p_b.S_{\mu_t} \neq \emptyset$ do
5     $p_a \leftarrow Queue.pop()$;
6     $p_a.S_{vul_t} \leftarrow ResolvedSV(PublishedV(V_i, p_a), p_b)$;
7     $p_b \leftarrow p_a$;
8 return $B_{\mu_t}(p_b@v_r, \cdots, p_u@v_n)$;
```

4.2 Identifying pivotal blocking chains

The metadata model $\mathcal{M}(s_i)$ = (dm_i, G_i) built by PLUMBER contains sufficient information for identifying vulnerable paths and blocking chains.

4.2.1 Identifying vulnerable paths

Let $vl \subset V_i$ be a set of vulnerable package versions affected by the vulnerabilities in dm_i, and $vr \subset V_i$ be a set of latest versions of active packages (i.e., publishing versions in the past year). On dependency graph G_i, PLUMBER considers package versions $p_k@v_m \in vr$ as root nodes and vulnerable package versions $p_u@v_n \in vl$ as leaf nodes, to identify all the vulnerable paths $\mathcal{P} = \{P_{\mu_t}(p_k@v_m, p_w@v_r, \cdots, p_u@v_n) | \mu_t \in dm_i\}$ via reachability analysis. Note that, PLUMBER only considers vulnerable paths whose root nodes are the latest versions of active packages, in order to analyze the status quo of vulnerability affects on the whole ecosystem.

However, a package version may contain multiple vulnerabilities and each vulnerability corresponds to a specific safe version set. For example, `Dotty` 0.1.0 contains two vulnerabilities *CVE-2021-23624* [48] and *CVE-2021-25912* [49]. While the safe versions of `Dotty` for these two vulnerabilities are {$\geq$ 0.1.1} and {$\geq$ 0.1.2}, respectively. To precisely find the blocking packages, when identifying vulnerable paths, a package version containing multiple vulnerabilities are distinguished as multiple leaf nodes. In this manner, PLUMBER ensures that a vulnerable path $P_{\mu_t}(p_k@v_m, p_w@v_r, \cdots, p_u@v_n)$ involves a unique vulnerability μ_t.

4.2.2 Identifying pivotal blocking chains

The detailed process of identifying blocking chains is described in Algorithm 1. For each vulnerable path $P_{\mu_t}(p_k@v_m, \cdots, p_w@v_r, p_e@v_f, \cdots, p_u@v_n) \in \mathcal{P}$, starting from the vulnerable package p_u, PLUMBER iteratively calculates each package's safe versions for vulnerability μ_t, until a blocking package p_w whose safe version set $p_w.S_{\mu_t} = \emptyset$ is found. Then, PLUMBER considers $B_{\mu_t}(p_w@v_r, p_e@v_f, \cdots, p_u@v_n)$ as a blocking chain.

Specifically, the safe versions of each package are resolved as follows:

- For vulnerable package p_u, PLUMBER records its safe version set as $p_u.S_{\mu_t} = PublishedV(V_i, p_u) - ver_t$, where $PublishedV(V_i, p_u)$ denotes a collection of p_u's published versions in the *npm* snapshot s_i and ver_t records the p_u's vulnerable versions affected by μ_t (Lines 1-3).
- For other package p_a ($a \neq u$) on the vulnerable path, PLUMBER uses function $ResolvedSV$ to calculate its safe version set. Suppose that p_b is the direct dependency of

p_a on the vulnerable path. $ResolvedSV$ returns the p_a's versions that depend on a safe version $v \in p_b.S_{\mu_t}$, or p_a's versions that deprecate package p_b (Lines 5-7).

Finally, PLUMBER ranks the identified blocking chains based on the number of vulnerable paths passing through them. The top ranking blocking chains are considered as pivotal ones should be remediated to enable the propagation of vulnerability fixes into a large amount of packages.

4.3 Deriving remediation schemes

Leveraging our empirical findings, PLUMBER aims to derive remediation schemes for the top K pivotal blocking chains. By characterizing these blocking chains, it reports which packages should perform remediation operations and how should they update dependencies to boost the propagation of vulnerability fixes in the whole ecosystem.

4.3.1 Characterizing blocking chains

Our empirical findings reveal that the blocking chains remediated using the same strategy share common characteristics. To facilitate deriving remediation schemes, PLUMBER characterizes the top K pivotal blocking chains according to (1) *whether the packages are active*, (2) *whether the packages are specified with open version constraints*, and (3) *whether the packages need to upgrade their direct dependencies' major versions to introduce the concerned vulnerability fixes.* The activity degree of packages determines which package on the blocking chain can update their dependencies. The open version constraints specified for packages can help automatically propagate vulnerability fixes. Since upgrading dependencies' major versions takes more efforts on changing and testing code, PLUMBER considers this factor when deriving remediation schemes to avoid bringing more burdens to package developers. Specifically, PLUMBER classifies the top pivotal blocking chains into three categories:

- ***Blocking chains to be remediated via Strategy A***: For a blocking chain $B_{\mu_t}(p_w@v_r, p_e@v_f, \cdots, p_u@v_n)$, if blocking package p_w is actively maintained (i.e., publishing releases in the past year), PLUMBER arouses p_w's awareness to upgrade its direct dependency p_e to a safe version $v \in p_e.S_{\mu_t}$, in order to introduce the vulnerability fix.
- ***Blocking chains to be remediated via Strategy B***: For a blocking chain $B_{\mu_t}(p_w@v_r, \cdots, p_g@v_h,\ p_e@v_f,\ p_c@v_d, \cdots, p_u@v_n)$, PLUMBER remediates it using *Strategy B*, if the blocking chain satisfies:
  - *a*. Blocking package p_w is inactively maintained;
  - *b*. There exists at least one intermediate package $p_e@v_f$ that is specified with open version constraint $c(p_g@v_h, p_e)$ ($p_e@v_f$ is in between the blocking package $p_w@v_r$ and vulnerable package $p_u@v_n$);
  - *c*. Upgrading intermediate package p_e's direct dependency p_c from version v_d to a safe version $v \in p_c.S_{\mu_t}$ needs not to cross p_c's major version.

  In this case, PLUMBER suggests the intermediate package $p_e@v_f$ to release a new version that can be resolved by constraint $c(p_g@v_h, p_e)$, in which it upgrades package p_c to a safe version $v \in p_c.S_{\mu_t}$. As such, the vulnerability fix can be propagated into blocking package $p_w@v_r$, remediating the blocking chain.
- ***Blocking chains to be remediated via Strategy C***: For a blocking chain $B_{\mu_t}(p_w@v_r, \cdots, p_u@v_n)$, PLUMBER suggests developers to migrate the blocking package p_w, if the blocking chain satisfies:
  - *a*. Blocking package p_w is inactively maintained;
  - *b*. There are no intermediate packages that are specified with open version constraints;
  - *c*. Although there exists intermediate packages that are specified with open version constraints, upgrading their concerned direct dependencies to the safe versions has to cross major version train.

  Based on the collected package migration records (in Section 4.1.2), it generates remediation schemes. If there exists a record $p_w \Rightarrow p_m$ and package p_m does not introduce other vulnerabilities, PLUMBER suggests the packages depending on blocking package p_w to perform the migration to package p_m.

To summarize, our empirical findings indicate that remediation costs token by the three strategies typically follow: *Strategy A* < *Strategy B* < *Strategy C*. As such, for the blocking chains induced by actively maintained packages, PLUMBER highly suggests remediation *Strategy A*. For the blocking chains induced by inactive blocking packages, in the cases that the intermediate packages (i.e., packages in between the blocking and vulnerable packages) are specified with open version constraints, PLUMBER suggests remediation *Strategy B* if they can backport to their lower version trains. Otherwise, PLUMBER adopts remediation *Strategy C* to migrate the inactive blocking chains.

4.3.2 Generating remediation reports

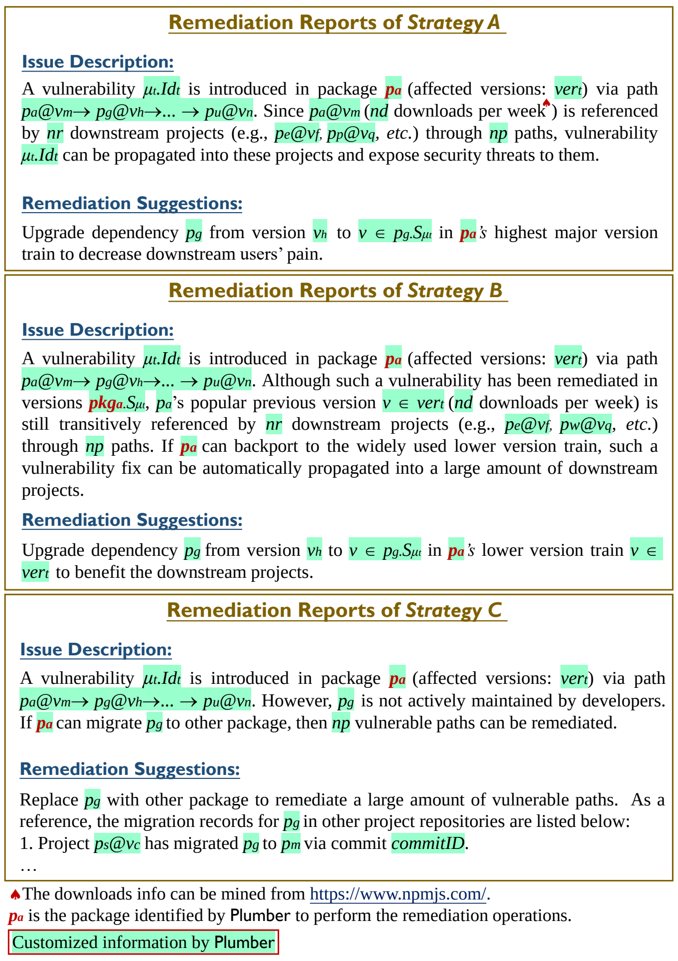
Based on the characteristics of blocking chains, PLUMBER identifies which packages should perform the remediation operations and then customizes reports with applicable solutions. Figure 12 shows the templates of remediation reports generated by PLUMBER. Specifically, in each report, it provides the following customized information:

- To arouse developers' awareness, PLUMBER gives the issue description with potential vulnerability impact analysis on the *npm* ecosystem: (1) the number of affected vulnerable paths, and (2) the scale of affected downstream projects.
- To boost the propagation of vulnerability fixes, PLUMBER explicitly points out how the packages update their dependencies: (1) suggesting the package version number to perform the remediation, and (2) listing the safe versions of dependencies to be upgraded.
- To facilitate developers migrate the inactive blocking packages, PLUMBER suggests the possible target packages and provides the migration records (including the corresponding commit ID) performed by other projects as a reference.

5 EVALUATION&AN ECOSYSTEM-LEVEL STUDY

In the evaluation section, we conducted an ecosystem-level study to answer the following three research questions:

RQ4 (Effectiveness of* PLUMBER*): *Are the remedial strategies derived by Plumber consistent with those made by developers?*



Remediation Reports of *Strategy A*

Issue Description:

A vulnerability $\mu_t.Id_t$ is introduced in package p_a (affected versions: ver_t) via path $p_a@v_m \rightarrow p_g@v_h \rightarrow \ldots \rightarrow p_u@v_n$. Since $p_a@v_m$ (nd downloads per week♠) is referenced by nr downstream projects (e.g., $p_e@v_f$, $p_p@v_q$, *etc.*) through np paths, vulnerability $\mu_t.Id_t$ can be propagated into these projects and expose security threats to them.

Remediation Suggestions:

Upgrade dependency p_g from version v_h to $v \in p_g.S_{\mu_t}$ in p_a's highest major version train to decrease downstream users' pain.

Remediation Reports of *Strategy B*

Issue Description:

A vulnerability $\mu_t.Id_t$ is introduced in package p_a (affected versions: ver_t) via path $p_a@v_m \rightarrow p_g@v_h \rightarrow \ldots \rightarrow p_u@v_n$. Although such a vulnerability has been remediated in versions $pkg_a.S_{\mu_t}$, p_a's popular previous version $v \in ver_t$ (nd downloads per week) is still transitively referenced by nr downstream projects (e.g., $p_e@v_f$, $p_w@v_q$, *etc.*) through np paths. If p_a can backport to the widely used lower version train, such a vulnerability fix can be automatically propagated into a large amount of downstream projects.

Remediation Suggestions:

Upgrade dependency p_g from version v_h to $v \in p_g.S_{\mu_t}$ in p_a's lower version train $v \in ver_t$ to benefit the downstream projects.

Remediation Reports of *Strategy C*

Issue Description:

A vulnerability $\mu_t.Id_t$ is introduced in package p_a (affected versions: ver_t) via path $p_a@v_m \rightarrow p_g@v_h \rightarrow \ldots \rightarrow p_u@v_n$. However, p_g is not actively maintained by developers. If p_a can migrate p_g to other package, then np vulnerable paths can be remediated.

Remediation Suggestions:

Replace p_g with other package to remediate a large amount of vulnerable paths. As a reference, the migration records for p_g in other project repositories are listed below:
1. Project $p_s@v_c$ has migrated p_g to p_m via commit *commitID*.
…

♠The downloads info can be mined from https://www.npmjs.com/.
p_a is the package identified by Plumber to perform the remediation operations.
🟩 Customized information by Plumber

Fig. 12: Simplified templates of generated remediation reports

RQ5 (Remediation challenges): *How challenging is remediating the blocking chains in the npm ecosystem?*

RQ6 (Usefulness of* PLUMBER*): *Can* PLUMBER *boost the propagation of vulnerability fixes in the npm ecosystem and provide useful remediation strategies to developers?*

To answer RQ4, we first identified a collection of package updates performed by developers that could remediate blocking chains with significant propagation effects of vulnerability fixes as a benchmark. We applied PLUMBER to analyzing the above blocking chains and then compare our derived remediation strategies with benchmark, in order to quantify the tool's effectiveness.

To answer RQ5, for the identified 358,422 blocking chains on the recent *npm* snapshot of August 1, 2021, we classified them into different difficulty levels of remediation. Furthermore, we observe their distribution and discuss the remediation challenges.

To answer RQ6, we applied PLUMBER to generating remediation reports for the top pivotal blocking chains and evaluate the remediation effectiveness based on developers' feedback. In our generated reports, we described the detected vulnerability issues together with remediation suggestions. More importantly, to arouse developers' awareness, we explained the vulnerability impacts on the *npm* ecosystem and the benefits of their dependency updates to the propagation of vulnerability fixes.

TABLE 8: Statistics of the *npm* snapshots from July 1, 2019 to July 1, 2020

Snapshots	#Root projects	#Dependency relations
s'_1: *July 1, 2019*	168,432	1,332,859
s'_2: *September 1, 2019*	171,764	1,342,622
s'_3: *November 1, 2019*	173,552	1,353,335
s'_4: *January 1, 2020*	187,230	1,376,942
s'_5: *March 1, 2020*	192,784	1,421,438
s'_6: *May 1, 2020*	206,642	1,486,321
s'_7: *July 1, 2020*	210,538	1,521,230

TABLE 9: Statistics of our benchmark

92 Type A blocking chain instances			
#blocking packages	#vulnerable packages	#root projects	#vulnerable paths(*Avg*)
32	25	630	38.4±24.5
293 Type B blocking chain instances			
#blocking packages	#vulnerable packages	#root projects	#vulnerable paths(*Avg*)
76	35	803	72.3±26.2
362 package updates			
Dependency updates by blocking packages	*Backporting by intermediate packages*	*Migration of inactive blocking packages*	
221	42	99	

15 package updates remediate more than one blocking chains

5.1 RQ4: Effectiveness of PLUMBER

Experimental setup. To evaluate the effectiveness of PLUMBER, we constructed a benchmark by collecting a set of package updates performed by developers. These package updates not only remediated blocking chains, but also enabled significant propagation effects of vulnerability fixes. The benchmark construction process involves three steps:

- *Constructing npm snapshots:* Based on our VP model, we constructed a collection of consecutive snapshots (s'_1-s'_7) of *npm* dependency metadata from July 1, 2019 to July 1, 2020, which were not used in our empirical study. Statistics of the *npm* snapshots are listed in Table 8. The seven snapshots only contain the active root projects and the package versions directly or transitively used by them. To precisely identify the blocking chains in snapshot s'_1, we only considered the vulnerabilities that were disclosed with fixes before July 1, 2019.
- *Identifying the blocking chains that have been remediated with significant effects of propagating vulnerability fixes:* Similar as the subject selection strategy adopted in RQ3, we focused on two types of blocking chains: (1) ***Type A.*** The blocking chains existed in snapshots $s'_1 - s'_{i-1}$ ($1 < i \leq 7$), while were remediated in snapshot s'_i. All the vulnerable paths passing through *Type A* blocking chains can introduce the vulnerability fixes. (2) ***Type B.*** The blocking chains that existed in snapshots s'_1-s'_7, while the number of vulnerable paths affected by them significantly reduced during evolution. On average, the blocking chains were passed through by 56.5±23.6 vulnerable paths in s'_1 and had fallen by 52.3%±9.2% in s'_7. In our study, we consider the blocking chains as *Type B* cases if the number of their involved vulnerable paths was more than 56.5 in s'_1, and had fallen by over 52.3% in s'_7.
- *Identifying the package updates that remediate the blocking chains of Types A and B:* We identified the package updates that remediated the *Types A* and *B* blocking chains as our benchmark. Specifically, we performed two tasks below:

TABLE 10: Effectiveness of PLUMBER in deriving remediation strategies

	Consistent Remediation Strategies	*Inconsistent Remediation Strategies*			
		NU	*NB*	*NR*	*NP*
Benchmark	-	0	2	71	2,585
PLUMBER	79.8% (289/362)	41	32	0	4,679

NU: # Dependency updates by blocking packages (*Strategy A*);
NB: # Backporting by intermediate packages (*Strategy B*);
NR: # Migration of inactive blocking packages (*Strategy C*);
NP: # Vulnerable paths remediated by package updates

– For each *Type A* blocking chain, we first identified the snapshot s'_i $(1 < i \leq 7)$ in which it had been remediated. Furthermore, for each package on such a blocking chain, we collected all its versions publishing between the interval of snapshots s'_1 and s'_i. Based on the chronological order of these version releases, we iteratively replayed the version updates of each package on this blocking chain, to check which version update enabled the remediation of blocking chain (i.e., blocking package can use the vulnerability fix version).

– For each *Type B* blocking chain, we collected the vulnerable paths passing through it on snapshot s'_1 but were remediated on snapshot s'_i $(1 < i \leq 7)$. Furthermore, for each package on such remediated vulnerable paths, we collected all its versions publishing between the interval of snapshots s'_1 and s'_i. Based on the chronological order of these version releases, we iteratively replayed the version updates of each package on the selected vulnerable paths, and checked which version update enabled the remediation of vulnerable paths (i.e., root project can use the vulnerability fix version).

Table 9 shows the statistics of our collected benchmark. The 362 package updates remediated 385 blocking chains, involving diverse blocking packages and vulnerable packages. They enabled the propagation of vulnerability fixes into 1,103 root projects via 12,926 vulnerable paths. Specifically, these package updates can be divided into three categories: (1) *dependency version updates by blocking packages*; (2) *backporting by intermediate packages*; (3) *migration of inactive blocking packages*. Furthermore, we applied PLUMBER to deriving remediation strategies for the 385 blocking chains. To quantify the effectiveness our tool, we validated if the derived remediation strategies were consistent with the real package updates in benchmark.

Results. Table 10 presents our evaluation results. In total, 289 out of 362 remediation strategies (79.8%) derived by PLUMBER are consistent with those in benchmark. Among the 289 consistent remediation strategies, 221 of them (76.6%) suggest blocking packages to update dependencies, 40 of them (13.8%) suggest intermediate packages to backport to their lower versions, and 28 of them (9.7%) suggest the affected packages to migrate inactive blocking packages to other ones. Consequently, 10,341 vulnerable paths introduced vulnerability fixes via the effective package updates.

For the 73 blocking chains, PLUMBER derived inconsistent remediation strategies with those in benchmark. Via balancing the remediation costs and propagation effects of vulnerability fixes, PLUMBER suggested 41 of them to be remediated based on *Strategy A* and 32 of them to be resolved using *Strategy B*. However, since our derived 71 remediation strategies either need to upgrade dependencies crossing

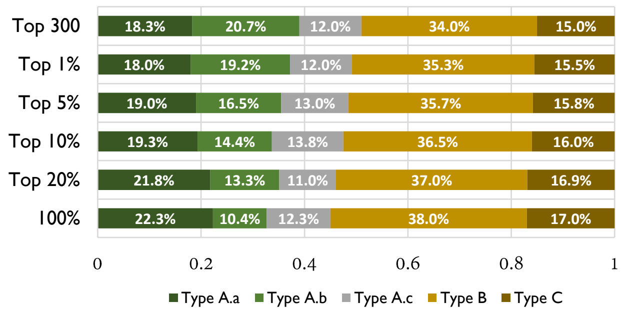


Fig. 13: Distribution of blocking chains with different remediation difficulty levels

major version trains or raise awareness of backporting practices, developers decided to migrate the inactive blocking packages to other well-maintained ones in our benchmark. For example, when resolving vulnerability issues, package `Reaction`'s developers tried to upgrade the vulnerable package `Meteorite` from 0.7.2 to its safe version 0.9.0, but encountered incompatibility issues. As described in issue `Meteorite#299` [50], they decided to migrate `Meteorite` to other alternative packages with high quality. In another example, the intermediate package `React-reality` on a blocking chain refused to backport to its lower versions (see discussions in issue `React-reality#1` [51]), due to a series of technical challenges. As a result, the affected projects had to migrate inactive blocking package `Configtest` [52], in order to remediate vulnerability issues. In addition, for the remaining two blocking chains, PLUMBER suggested the active blocking packages to upgrade dependencies to their safe versions (i.e., *Strategy A* with less remediation efforts). However, in our benchmark, developers eventually performed backporting practices to bring the benefit of vulnerability fixes to their downstream projects (clues could be found in issues `Mapbox#7` [53] and `Opac#1034` [54]).

*The evaluation results indicate the effectiveness of our PLUMBER in deriving remediation strategies for blocking chains. 289 out of 362 remediation strategies (**79.8%**) derived by PLUMBER are consistent with those in our benchmark. For the 73 inconsistent remediation strategies, our tool generates the suggestions via balancing the remediation costs and propagation effects of vulnerability fixes. However, due to the incompatibility issues or technical challenges, the package updates in our benchmark resolved the vulnerability issues in alternative ways.*

5.2 RQ5: Remediation Challenges

Experimental setup. To understand how challenging is remediating the blocking chains in the *npm* ecosystem, we analyzed the identified 358,422 blocking chains on the recent *npm* snapshot of August 1, 2021 and classified them into different difficulty levels of remediation. Based on the packages' characteristics on each blocking chain, leveraging our empirical findings in RQ3, we identify the blocking chains that are difficult/less difficult to be remediated, according to the following criteria:

• ***Type A. Blocking chains are of high difficulty to remediate.***
  – *a. The blocking chains that are induced by inactive blocking packages, whose involved intermediate packages are all specified with locked version constraints* (*Strategy C*).

- *b. The blocking chains that are induced by inactive blocking packages, whose involved intermediate packages are specified with open version constraints but need to upgrade dependencies' major versions to introduce vulnerability fixes (Strategy C).*
- *c. The blocking chains that are induced by active blocking packages, but blocking packages have to upgrade dependencies' major versions in order to introduce vulnerability fixes (Strategy A).*

For *Types A.a* and *A.b* blocking chains, developers cannot easily remediate them through upgrading dependencies. They are suggested to spend efforts to migrate the blocking packages. For *Type A.c* blocking chains, PLUMBER encourages the blocking packages to upgrade their dependencies, which requires more efforts to test the packages crossing their major versions.

- ***Type B. Blocking chains are of medium difficulty to remediate.***
  - *The blocking chains that are induced by inactive blocking packages, whose involved intermediate packages are specified with open version constraints but need not to upgrade dependencies' major versions to introduce vulnerability fixes (Strategy B).*

Type B blocking chains would be remediated if the involved intermediate packages can backport to their lower version trains. However, the backporting practices need package developers to spend efforts on supporting multiple version trains.

- ***Type C. Blocking chains are of low difficulty to remediate.***
  - *The blocking chains that are induced by active blocking packages, and the blocking packages need not to upgrade dependencies' major versions to introduce vulnerability fixes (Strategy A).*

Type C blocking chains can be remediated by active blocking packages via upgrading dependencies with fewer compatibility issues, which are highly recommended by PLUMBER.

Results. Figure 13 shows the distribution of blocking chains with different remediation difficulty levels. To see trends, we divided all blocking chains into six groups (overlapping) based on their influence ranking: top 300, 1%, 5%, 10%, 20% and 100%. For the top 20% pivotal blocking chains that affect most of vulnerable paths in the *npm* ecosystem, 46.1% of them (*Types A.a*, *A.b* and *A.c*) are challenging to remediate. 81,044 packages depending on *Types A.a* and *A.b* blocking chains should migrate the inactive blocking packages to avoid inducing vulnerability fixes, which bring heavy burden to developers. For the 11.0% *Types A.c* blocking chains, the blocking packages have to upgrade their dependencies' major versions to introduce vulnerability fixes, which need more testing efforts to avoid incompatibility issues.

Moreover, to remediate the 37.0% top pivotal blocking chains of *Type B*, 26,523 involved intermediate packages specified with open version constraints should backport to their lower version trains. Only 16.9% *Types C* blocking chains can be easily remediated by the active blocking packages via upgrading their dependencies to safe versions. It would take a long time for pivotal packages to realize the affects of their dependency upgrades/migrations/backporting practices on propagation of vulnerability fixes. The ecosystem-level demographics of blocking chains generated by PLUMBER can provide concrete guidelines and policies for package developers to address the above remediation challenges.

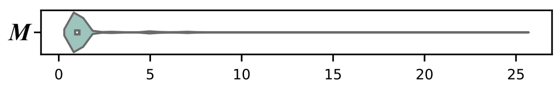


Fig. 14: Occurrences of each dependency change pair M in different projects during evolution

*For the top 20% pivotal blocking chains that affect most of vulnerable paths in the npm ecosystem, **46.1%** of them are challenging to remediate. They either need to migrate the inactive blocking packages or upgrading their the dependencies' major versions to introduce vulnerability fixes, which need more code changing and testing efforts. **37.0%** top pivotal blocking chains can be remediated via backporting practices of the involved intermediate packages. Only **16.9%** top pivotal blocking chains can be easily remediated by active blocking packages via upgrading their dependencies to safe versions with fewer incompatibility issues.*

5.3 RQ6: Usefulness of PLUMBER

Experiment setup. For the top 300 pivotal blocking chains, we deployed PLUMBER to report remediation suggestions to corresponding package developers. As shown in Figure 13, 27.0% (81 *Types A.c* and *C*) and 34.0% (102 *Type B*) blocking chains should be remediated using *Strategies A* and *B*, respectively. For the remaining 39.0% blocking chains (117 *Types A.a* and *A.b*), based on *Strategy C*, PLUMBER should warn developers to migrate the involved 56 inactive blocking packages.

As of September 1 2021, PLUMBER located 48,325 dependency change pairs <p_u, p_t> (i.e., a project deprecated dependency p_u and added a new dependency p_t across two consecutive *npm* snapshots). Figure 14 shows the occurrences of each dependency change pair M in different projects during evolution (on average M = 2.15±3.50). For 1,081 out of 48,325 (2.2%) dependency change pairs <p_u, p_t>, PLUMBER found migration keywords in the commit messages on GitHub (e.g., "replace p_u") and identified the addition of package p_t in the corresponding code commits as well. The 1,081 migration records appear at least five times in different projects during evolution ($M \geq 5$). PLUMBER considered such dependency change pairs as certain migration records. Among the 56 inactive blocking packages in the 117 *Types A.a* and *A.b* pivotal blocking chains, 43 of them (76.8%) can be migrated according to our mined records.

It is worth noting that PLUMBER only concerned the blocking chains induced by vulnerabilities with critical, high or medium levels, and merged the remediation suggestions that should be performed by the same package into one report. In total, it submitted 268 remediation reports (73 reports using *Strategy A*, 98 reports using *Strategy B* and 97 reports using *Strategy C*) to packages' issue trackers. To arouse developers' awareness, in the remediation reports, we explained the vulnerability impacts on the *npm* ecosystem and the benefits of dependency updates to the propagation of vulnerability fixes. We evaluate the usefulness of PLUMBER based on developers' feedback.

Ethical Considerations. To avoid spamming the open-source community and developers, we only remediated the

TABLE 11: Statistics of remediation reported by PLUMBER

Report Status	*Strategy A*	*Strategy B*	*Strategy C*
♠ *Vulnerabilities remediated using our strategies*	25	27	32
♣ *Vulnerabilities under remediation using our strategies*	7	5	4
★ *Remediation strategies confirmed, but inducing incompatibility issues*	9	18	0
● *Developers were unwilling to create backports*	0	7	0
♦ *False positive vulnerabilities checked by developers*	3	10	9
■ *Inaccurate info in the vulnerability DB*	1	1	0
▼ *Do not care about vulnerabilities*	7	12	3
▲ *Reports pending*	21	18	49
Confirmation rate = 47.4% (127/268)	**56.2%**(41/73)	**51.0%**(50/98)	**37.1%**(36/97)
Remediation rate = 66.1% (84/127)	**61.0%**(25/41)	**54.0%**(27/50)	**88.9%**(32/36)

More detailed info of reports is provided on **http://plumber-npm.com/**.

top 300 pivotal blocking packages and thoroughly validated the information of reports generated by PLUMBER. We guaranteed that there were no editing errors in our remediation reports and have checked the correctness of vulnerability information (e.g., unique CVE identifier, severity level and affected versions of vulnerable package). More importantly, we manually validated if the affected downstream projects could introduce the concerned vulnerability fix, combining the *npm* dependency metadata with *node-semver* rules. All the 268 remediation reports were submitted in compliance with the projects' contributing guidelines and licenses.

Results. Table 11 summarizes the statues of our remediation reports. 127 reports (47.4%) were quickly confirmed by developers, and 84 confirmed reports (66.1%) were later remediated or were under remediation using our strategies. 22 reports (8.2%) had been rejected since the relevant vulnerabilities were considered as false positives or inaccurate records in vulnerability DB through developers' validation. While in our 22 reported issues (8.2%), developers expressed that they did not care about vulnerabilities. The other reports were still pending, likely due to the less active maintenance of the projects.

Impacts on boosting the propagation of vulnerability fixes. In Table 12, *#AVP′* and *#ARP′* denote the number of vulnerable paths and root projects that can introduce vulnerability fixes after the concerned blocking chains being remediated. *#RD* denotes the download count of package versions having been remediated based on our suggestions (recorded on https://www.npmjs.com/ as of December 23, 2021). We use these three metrics to estimate the impacts of our submitted remediation reports on propagation of vulnerability fixes.

After the 84 packages published their remediated versions according to our reports, in total, the involved vulnerability fixes can be propagated into 16,403 root projects via 92,469 vulnerable paths. It is worth noting that 15,719 vulnerable paths (17.0%) are remediated based on backporting practices, while 14,425 paths (15.6%) get rid of vulnerabilities by migrating inactive blocking packages. On average, each remediated package version is receiving 72,678 downloads per week by the time of this work. The results indicate the usefulness of PLUMBER in boosting the propagation of vulnerability fixes in the ecosystem. It is thus necessary to deploy PLUMBER to continuously capture the pivotal blocking chains as well as suggest proper remediation strategies with impact analysis.

PLUMBER's generated remediation reports have boosted the propagation of vulnerability fixes into 16,403 root projects via 92,469 vulnerable paths. On average, each remediated package version received 72,678 downloads per week.

***Feedback on remediation reports generated by* PLUMBER.** Different remediation strategies achieved different confirmation rates (*Stratgy A:* 56.2%, *Stratgy B:* 51.0% and *Stratgy C:* 37.1%). Most submitted reports received developers' positive feedback. In 45 reports, developers were unwilling to remediate vulnerabilities because they encountered challenges (e.g., incompatibility issues) during the fixing process or did not care about security issues (see discussions in Section 5.4). We discuss the representative cases that reveal the usefulness of PLUMBER in five aspects:

- ***The responsive reports cover many well-known npm projects.*** 25 (*Types A.c* and *C*) and 27 (*Type B*) pivotal blocking chains were remediated by developers using *Strategies A* and *B*, respectively. 32 packages migrated the inactive blocking packages based on our remediation *Strategy C*. The responsive reports cover many well-known *npm* projects, such as `Tensorflow/tfjs#5492` [29] (*Strategy A*), `Ethers.js#1782` [30] (*Strategy B*), `Googleapi#337` [55] (*Strategy B*), `Google-gax#1062` [56] (*Strategy B*), `Google-cloud/storage#1538` [57] (*Strategy B*), `GoogleChrome/workbox#2912` [31] (*Strategy C*), and `Microsoft/just#555` [58] (*Strategy C*).
- ***Many remediation reports struck a chord with downstream packages.*** For example, `Node-sqlite3` is a blocking package, which hinders the propagation of a vulnerability fix (*CVE-2021-32803* with high severity) along vulnerable paths. Our submitted remediation report `Node-sqlite3#1493` [59] (*Strategy A*) attracted 17 developers' comments to request their dependency upgrades and was linked to 16 real vulnerability issues of the downstream projects. A downstream user of `Node-sqlite3` commented that: *"It is package developers'/maintainers' responsibilities to remove the vulnerabilities. Otherwise, such effects on other packages like a cancer in the ecosystem."*
- ***Our remediation reports indeed aroused the pivotal packages' awareness to backport to their lower version trains to benefit the whole ecosystem.*** For example, in report `Peer-id#153` [60] (*Strategy B*), we suggested `Peer-id` to backport to version 0.12.5, in order to mitigate the affects of vulnerability *CVE-2020-7720* on 450 vulnerable paths. Encouragingly, this report was quickly confirmed by developers and left a comment: *"This vulnerability had already been resolved since `Peer-id`@0.14.3. We typically don't maintain versions far back, but make an exception for this case whose weekly download rate is rather high."*
  In approach [1], Chinthanet et al. concluded that practitioners should provide developers more awareness mechanisms to enable quicker planning of dependency updates. This point of view is echoed by the above developers' feedback in our study.
- ***Many projects were willing to migrate the inactive blocking packages to remediate vulnerabilities.*** For example, in report `Image-cropper#42` [61] (*Strategy C*), we suggested `Image-cropper` to replace the inactive blocking package `Babel-cli` that introduce vulnerability *CVE-2020-28469* with a high-quality package `Webpack`. After

TABLE 12: The impacts of our remediated reports on propagating vulnerability fixes

ID	Issue Reports	Strategy	#AVP′	#ARP′	#RD	ID	Issue Reports	Strategy	#AVP′	#ARP′	#RD
1	glob-entries-plugin#19	C	576	565	1,539	138	browser-extension#463	C	143	25	1,148
2	storybook#15830	C	1,759	327	523,504	139	bst#677	C	97	61	1,808
3	ethers.js#1782	B	12,471	1,792	100,099	140	victory#1945	A	536	137	30,353
7	just#555	C	320	13	6,619	141	openVectorEditor#760	C	535	449	3,230
8	workbox#2912	C	297	31	2,425	143	amcharts4#3684	A	530	480	24,796
9	off-main-thread#46	A	2,922	2,528	596,269	147	fis3#1328	A	543	511	5,314
10	actionhero#1907	C	51	34	1,081	150	cross-fetch#112	B	1,083	704	148,135
11	assetgraph-builder#892	C	241	33	1,141	152	react-components#265	C	515	431	6,441
13	hads#65	C	37	11	1,797	154	release-it#794	B	511	487	18,580
14	node-windows#293	C	112	5	2,974	156	ibm-security#1083	C	508	440	1,383
18	swig-email-templates#50	C	71	53	2,525	157	jsonld.js#457	B	642	453	37,234
24	oc#1197	C	368	211	2,364	158	plugin-i18n-json#44	C	174	35	31,366
28	vue-cli#6632	C	163	71	2,509	163	node-jose#322	B	1,009	304	128,959
34	MQTT.js#1306	B	1,394	228	30,998	165	node-console-stamp#54	A	504	492	9,456
42	strider#1124	C	242	13	2,434	166	rendition#1361	C	502	434	2,837
44	universalviewer#808	C	234	11	2,820	169	easy-peasy#684	B	501	432	2,964
47	google-p12-pem#337	B	1,265	683	205,683	172	mammoth.js#290	A	543	334	10,773
57	react-native#31987	A	10,189	1,016	77,208	173	sequelize#13398	B	956	632	81,364
65	vf-core#1644	C	103	74	2,415	189	terminal-kit#182	B	482	442	7,620
69	pdfkit#1273	A	909	498	136,646	193	spectral#1755	B	475	455	2,559
75	rsuite#1852	C	778	613	3,969	194	auth0.js#1193	A	476	412	19,988
79	imodeljs#2066	C	36	6	1,111	195	powerapps-docs#2663	C	121	5	1,289
82	http-server#707	B	799	557	13,820	199	nodejs-storage#1538	B	783	398	72,336
86	azure-storage-node#691	A	754	625	25,460	204	apisauce#269	B	498	447	14,732
94	assetgraph#1187	C	15	13	1,701	206	xml-crypto#231	B	464	443	42,446
97	enact#2941	C	646	527	1,409	209	gax-nodejs#1062	B	678	475	128,095
102	js-libp2p-webrtc-star#374	A	741	332	1,546	211	iot-device-sdk-js#376	A	474	355	33,420
103	tedious#1297	B	606	553	51,840	213	node-sqlite3#1493	A	485	459	263,472
104	avatar-image-cropper#42	C	114	2	1,008	214	fabric#98	A	454	354	13,057
106	jupyterlab#10818	C	137	43	1,154	219	js-peer-id#153	B	2,788	290	6,986
111	syntax-highlighter#417	B	2,101	389	214,515	220	nestjs-mailgun#21	C	125	9	2,167
112	father#391	A	650	507	2,234	224	importers#136	A	598	433	4,730
116	spatial-navigation#104	C	571	490	1,731	231	DefinitelyTyped#55072	B	471	329	2,755
120	oas-kit#467	A	562	464	8,310	232	engine.io-client#676	B	2,452	668	53,578
122	lint-staged#995	B	592	398	530,750	234	z-schema#269	B	458	165	240,864
123	commitlint#2687	B	563	517	48,814	238	markdown-preview#101	B	440	313	5,378
125	patternfly-react#6153	C	561	479	1,459	239	ng2-rest#18	A	1,113	105	12,003
127	node-convict#396	B	613	540	19,981	243	celo-monorepo#8521	A	3,478	604	3,273
128	bundle-stats#1578	C	552	470	1,683	246	tfjs#5492	A	588	537	5,875
132	json-rpc-middleware#99	B	551	389	3,540	256	rollup-plugin-styles#188	A	420	420	15,707
133	rollup-plugin-postcss#386	A	549	530	149,887	260	node-xlsx#156	A	422	400	7,565
135	git-up#27	A	542	370	1,587,296	262	parse-server#7491	A	457	283	4,014

checking the historical migration record `Babel-cli` ⇒ `Webpack` (Commit#*8616b1* [62] in `Learn-webpack`) provided by PLUMBER, developers accepted our remediation suggestion.
Similar findings can be found in study [1]. Among the 5,417 projects investigated by Chinthanet et al., 1,389 of them (25.64%) decided to remove vulnerable dependencies to mitigate the secure risks. Instead of waiting for the vulnerability fix, client projects might remove the vulnerable package if they were able to find a similar one as a replacement.

- ***Developers showed great interest in the* PLUMBER *tool.*** For example, one developer left a comment in report `Xml-crypto#231` [63]: *"I noticed that you have raised similar issues in other code repositories. Are you using an automated tool? It seems that the tool can help the vulnerability patches propagate into our npm projects."* Besides, PLUMBER's remediation report `Meow#195` [64] struck a chord with the affected downstream packages. A developer supported our suggestions for `Meow` and commented that: *"Based on the detailed issue reports, there exists more than 33,500 affected downstream projects. Would you be willing to slightly compromise your principles in exchange for making a lot of downstream users happy?"*

Such feedback indicates that remediating pivotal blocking chains with proper strategies is indeed important to and welcomed by real-world *npm* developers. Encouraged by such comments, we are planning to release our tool for public use to help build a healthy *npm* ecosystem.

47.4% of our remediation reports received positive feedback from many well-known npm projects, such as `Tensorflow/tfjs`, `Ethers.js`, `Googleapi`, and `GoogleChrome/workbox`, etc. These reports indeed aroused the pivotal packages' awareness to perform dependency upgrades/backporting/migrations in order to remediate vulnerabilities, benefiting the whole ecosystem. Besides, developers showed interest in the PLUMBER *tool and recognized its usefulness in building a healthy npm ecosystem.*

5.4 Discussions

Lessons learnt from developers' feedback. Developers were unwilling to resolve our remediation reports mainly because of five cases: (1) *they had weak awareness of vulnerability affects and did not care about them*; (2) *dependency upgrades induced incompatibility issues*; (3) *they were unwilling to create backports to remediate vulnerabilities*; (4) *they only concerned the exploitable vulnerability code*; and (5) *vulnerability DB recorded inaccurate information*.

Based on developers' feedback, we can label the packages of *Cases (1), (2) and (3)* as improper ones to perform remediation. PLUMBER regards the improper packages as equivalent of inactive packages to derive more feasible remediation strategies for blocking chains. Regarding *Cases (4)* and *(5)*, the remediation ability of PLUMBER can be further improved, if *vulnerability info correction* and *vulnerability* *reachability analysis* techniques are integrated into its architecture. Several approaches have been proposed to correct vulnerability records [22] and identify the unexploitable vulnerabilities [19–21], which provide technical references for tool enhancement.

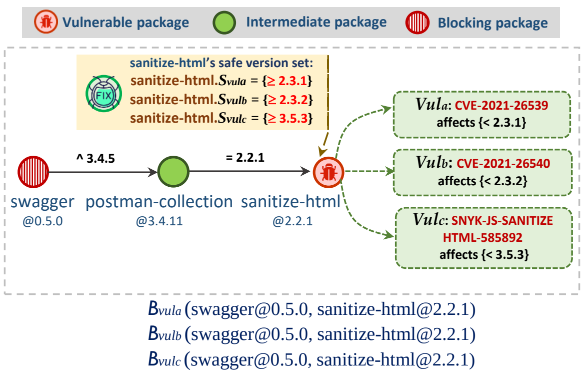
Backporting practices. PLUMBER's remediation *Strategy B* is inspired by backporting practices. Backporting requires additional efforts from package developers, as it entails transplanting important bug and vulnerability fixes to earlier versions [8]. Downstream projects can directly benefit from backported fixes without having to upgrade their dependencies to higher major or minor versions. Thus, backporting practices of packages on the pivotal blocking chains can significantly boost the propagation of vulnerability fixes in the ecosystem.

In our study, we found an interesting phenomenon that many developers were willing to support their lower major releases for a period of time, after which they have no backporting plans to avoid giving downstream packages a false sense of security. Instead, they encourage downstream users to abandon the old versions and bump to higher ones. For example, in remediation report `Angular-cli`#21608 [65], a developer express that *"`Angular-cli`'s versions prior to 10.0.0 (released two years ago) are no longer supported, since the backporting request goes against our maintenance policy: (a) 6 months of active support, during which regularly-scheduled updates and patches are released; (b) 12 months of long-term support, during which only critical fixes and security patches are released."* According to developers' feedback, PLUMBER gives higher priority to request package developers to backport to their lower versions that have been published within one year.

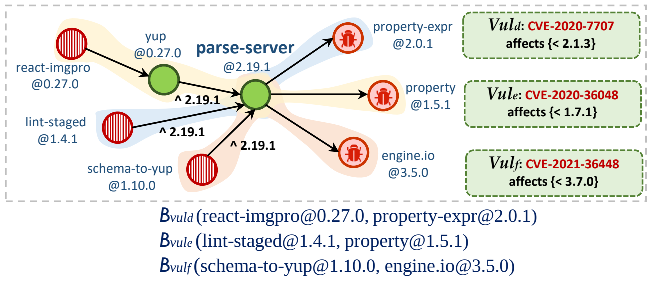
Similar empirical findings were given by Decan et al. in work [8]. Their investigation for backporting practices in open source community indicated that continued support of lower major trains might have the adverse effect of delaying the migration of dependents to the highest major release. Since lower major releases are unlikely to be maintained forever even when they benefit from backports, package producers should devise and advertise a clear phase-out strategy of lower major trains.

Noteworthy blocking chain cases. Among the top 300 blocking chains remediated in our study, we noticed two types of interesting blocking chain cases:

- *Multiple blocking chains share the same packages but involve different vulnerabilities.* On such blocking chains, each package has different safe version sets corresponding to different vulnerabilities. When deriving remediation strategies, for each package, PLUMBER identify the intersection of safe versions for all the involved vulnerabilities. Figure 15(a) shows three blocking chains passing through the same packages but package `sanitize-html` contain three vulnerabilities *CVE-2021-26539*, *CVE-2021-26540* and *SNYK-JS-SANITIZE HTML-585892*. Since each vulnerability corresponds to a safe version set for `sanitize-html`, to remediate all of them, PLUMBER suggests `postman-collection` to publish version 3.4.*, in which it upgrades `sanitize-html` from version 2.2.1 to 3.5.3 (the lowest version satisfying the intersection of three safe versions). However, to avoid inducing incompatibility issues, in report `postman-collection`#1215 [66], developers finally decided to remediate two vulnerabilities


Vulnerable package
Intermediate package
Blocking package
sanitize-html's safe version set:
sanitize-html.S_{vula} = {≥ 2.3.1}
sanitize-html.S_{vulb} = {≥ 2.3.2}
sanitize-html.S_{vulc} = {≥ 3.5.3}
FIX
^ 3.4.5
= 2.2.1
swagger @0.5.0
postman-collection @3.4.11
sanitize-html @2.2.1
Vula: CVE-2021-26539 affects {< 2.3.1}
Vulb: CVE-2021-26540 affects {< 2.3.2}
Vulc: SNYK-JS-SANITIZE HTML-585892 affects {< 3.5.3}
B_{vula} (swagger@0.5.0, sanitize-html@2.2.1)
B_{vulb} (swagger@0.5.0, sanitize-html@2.2.1)
B_{vulc} (swagger@0.5.0, sanitize-html@2.2.1)

(a) Multiple blocking chains share the same package but involve different vulnerabilities


react-imgpro @0.27.0
yup @0.27.0
parse-server @2.19.1
lint-staged @1.4.1
schema-to-yup @1.10.0
^ 2.19.1
^ 2.19.1
^ 2.19.1
property-expr @2.0.1
property @1.5.1
engine.io @3.5.0
Vuld: CVE-2020-7707 affects {< 2.1.3}
Vule: CVE-2020-36048 affects {< 1.7.1}
Vulf: CVE-2021-36448 affects {< 3.7.0}
B_{vuld} (react-imgpro@0.27.0, property-expr@2.0.1)
B_{vule} (lint-staged@1.4.1, property@1.5.1)
B_{vulf} (schema-to-yup@1.10.0, engine.io@3.5.0)

(b) Multiple blocking chains intersect at an active package

Fig. 15: Illustrative examples of noteworthy blocking chains

(i.e., *CVE-2021-26539* and *CVE-2021-26540*) by upgrading `sanitize-html` to 2.3.2 rather than acrossing major versions.

- *Multiple blocking chains intersect at an active package.* In the cases that multiple blocking chains intersect at an active package, PLUMBER suggests the intersection to perform remediation by upgrading multiple dependencies. Figure 15(b) shows an illustrative example. Three blocking chains intersect at `parse-server`@2.19.1, on which `parse-server` are all specified with the open version constraint ^ 2.19.1. In report `parse-server`#7491 [67], PLUMBER suggested `parse-server` to publish a version 2.19.* (2.19.* could be referenced by all the three blocking packages), in which it upgraded three dependencies `property-expr`, `property` and `engine.io` to the corresponding safe versions. Encouragingly, `parse-server`'s backporting remediated the three blocking chains together, allowing three vulnerability fixes to be propagated into 283 root projects via 457 dependency paths.

Responsibility for remediating pivotal blocking chains. After deriving remediation strategies for pivotal blocking chains, PLUMBER sends alert notifications to the issue trackers of certain packages that are feasible to update dependencies. The customized issue reports with applicable solutions can arouse developers' awareness of mitigating lags in vulnerability fix propagation. In our evaluation, package developers' feedback indicate the effectiveness of our notification mechanism. However, the remediation reports only reflect the authors' personal views rather than the organizational position. To build a long-term healthy ecosystem, PLUMBER's vision is to continuously report the identified pivotal blocking chains together with remediation strategies to the large-scale reporting entities such as CERT

(Computer Emergency Response Team)[3]. According to our validated remediation reports, the entities can alert relevant organizations or package vendors to perform the dependency updates within a guaranteed time frame, on behalf of the open-source community. Achieving acknowledgement by the large-scale reporting entities is the direction for our future efforts.

PLUMBER's *generality beyond the npm ecosystem.* Our methodology can be generalized to other programming language ecosystems if two conditions below are satisfied:

- It is feasible to collect and model the metadata of package dependency relationships and vulnerabilities in the ecosystem, which enables PLUMBER to continuously monitor their evolution and identify blocking chains.
- The programming language ecosystem should resolve package dependencies using the similar semantic version syntax as *npm* (e.g., *PyPI* and *Cargo*). The version range syntax allows flexibility, which enables projects to automatically deploy the new versions of their required packages that satisfy the specified version range (highest available version installation rule). In this manner, PLUMBER can boost the propagation of vulnerability fixes via open constraints if the new remediated package versions are released.

6 Threats to Validity

Following the structure provided by Wohlin et al. [68], we discuss the main threats to validity of our research.

Threats to external validity. The main external threat is the generality of our results to other ecosystems. In this study, we made all observations on *npm* packages. However, our analysis is applicable to other ecosystems that have similar package management systems, e.g., *PyPI* for Python and *NuGet* for the .NET. Additionally, as packages evolve over time, our experiment results are limited to the time period (August 1, 2020 - August 1, 2021) when we conducted those experiments. Therefore, the findings may not generalize well to all the projects. In the future, we would like to include more projects and involve various programming languages into our empirical study, so that our findings would be more representative.

Threats to internal validity. Threats to internal validity concern the external factors not considered in our study. The vulnerability dataset used for our case study may not cover all known vulnerabilities. To reduce the threat, we crawl vulnerability reports from three recognized vulnerability databases, i.e., *GitHub Advisory DB*, *Snyk Vulnerability DB* and *NPM Security Advisories* and filtered out the duplicated ones. The dataset covers 3,948 vulnerabilities, involving 2,289 vulnerable packages and 38,166 vulnerable package versions. Hence, this threat has minimal influence on our analysis results.

The second internal threat is the correctness and completeness of the remediation strategy taxonomy constructed by our manual inspection in RQ3. Our study involved much manual work. We understand that such a manual process is subject to errors. To reduce the threat, we followed an open coding procedure to inductively categorize the common remediation strategies.

3. A Computer Emergency Response Team (CERT) is a group of information security experts responsible for the protection against, detection of and response to an organization's cybersecurity incidents.

Threats to conclusion validity. Threats to conclusion validity concern the reliability of the tools and methods we have used in our paper. The vulnerability metadata used in our empirical study is collected through a self-developed VP model. Although tool development could have potentially introduced bugs that might affect the results presented in this paper, we tried our best to reduce the probability of bugs. Since the vulnerability information from database is usually described in plain text, collecting the metadata is an error-prone process. For each vulnerability report, our VP model needs to filter programming languages and identify its *unique identifier*, *affected package*, *affected package versions*, *fix versions* and *severity level*. To mitigate such threats, the VP model took four months for three authors of this paper who had over two years vulnerability analysis experience to implement and test. The two experienced authors conducted daily manual validation of vulnerability metadata. Disagreements were reconciled with the third author joining the discussions. The *Cohen's kappa* for their data validation is 0.92, indicating a near perfect inter-rater agreement.

7 Related Work

The work most related to our study falls into three categories: studies on *vulnerable dependencies*, *software ecosystems* and *library package updates*. This section covers a comprehensive portion of prior work on these topics and reflects on how the work compares with ours.

Vulnerable Dependencies. Several studies [2, 10–15, 23, 24] in the literature investigated vulnerabilities that come from dependencies. In approaches [23] and [24], researchers empirically studied the evolution and decay of vulnerabilities in source code, and found that due to the software code reuse and third-party libraries, most vulnerabilities are recurring. Cox et al. introduced metrics to qualify the "dependency freshness" of software projects, to understand the relations between outdated dependencies and vulnerabilities based on industry benchmark. Their investigation results shown that the projects using outdated dependencies four times as likely to have security issues as opposed to the projects that were up-to-date. Studies [2, 10, 11] investigated millions of *npm* packages to analyze how and when these vulnerabilities were discovered and fixed, and to what extent they affected other packages. Zimmermann [11] pointed out that a small proportion of individual vulnerable packages could impact large parts of the entire ecosystem. Whereas approaches [12–15] studied the evolution and impacts of vulnerable packages in specific communities, including Firefox, Docker Images, Apache HTTP Server and Tomcat, and Proprietary Systems. They reported their findings and provided guidelines for software maintainers and tool developers to improve the process of dealing with security issues.

Recent studies [19–22] were proposed to reduce the false alarms for reporting vulnerable dependencies. Since CVE reports are used to share information about software vulnerabilities and provide a baseline for security measures, their correctness is crucial for detecting and patching software vulnerabilities. Woo et al. [22] proposed a precise mechanism V0FINDER for discovering the correct origin of publicly reported software vulnerabilities, which can enhance the credibility of information provided by the CVEs. Pashchenko et al. [17, 18] presented precise methodologies for counting actually vulnerable dependencies, which combined the code-based analysis of patches with information on build, test, and update dates. Afterwards, the evaluation on 25,767 Java libraries shown that the proposed methodology can reduce the number of false alerts for vulnerable dependencies. Zapata et al. [19] assessed the security risks of having vulnerabilities in dependent libraries by analyzing call graphs of the vulnerable functions. They manually validated 60 projects that depend on vulnerabilities, and found that 73.3% of them were actually safe because they did not make use of the vulnerable functionality of their dependencies. In approaches [20, 21], Ponta et al. provided a fine-grained assessment approach using static and dynamic analysis to determine whether the located vulnerable code would be reachable.

Since vulnerabilities can easily be propagated through code clones, the vulnerable code clones are increasing in conjunction with the open-source software, potentially contaminating many projects [69, 70]. Approaches [71, 72] are proposed to efficiently and accurately detect vulnerable code clones in large software projects. Kim et al. [71] developed a scalable tool VUDDY, leveraging function-level granularity and a length-filtering technique to reduce the number of signature comparisons. Such an efficient design enables VUDDY to preprocess a billion lines of code in 14 hour and 17 minutes, after which it requires a few seconds to identify code clones. Besides, they also presented a security-aware abstraction technique, which extends the scope of VUDDY to identify variants of known vulnerabilities with high accuracy. The evaluation results demonstrated its effectiveness in detecting zero-day vulnerabilities in widely used software projects. Jiang et al. [72] proposed a scalable binary code clone detection framework QUICKBCC for vulnerability scanning. The framework was built on the idea of extracting semantics from vulnerable binaries both before and after security patches, and comparing them to target binaries. In order to improve performance, they created a signature based on the changes between the pre- and post-patched binaries, and implemented a filtering process when comparing the signatures to the target binaries. QUICKBCC outperformed other approaches in terms of performance when detecting well known vulnerabilities with acceptable level of accuracy.

The study most relevant to our work is done by Chinthanet et al [1]. The researchers conducted an empirical investigation to identify lags that may occur between client project-side fixing releases and package-side fixing releases. Through a preliminary study on 231 package-side fixing releases, they observed that nearly 85.72% of them bundled commits that were unrelated to a vulnerability fix. Even if the package-side fixing release was quick, the stale clients require additional efforts to introduce the fixes. This work lays the groundwork for mitigating propagation lags in an ecosystem. Compared with all the work mentioned above, our study is different in terms of the research scope and study method. In this paper, we introduced the concepts of blocking packages and blocking chains, and focused on how to remediate them for facilitating the propagation of vulnerability fixes. In addition to a thorough empirical study to characterize the blocking chains, we also propose a technique to automatically monitor and generate remediation strategies for pivotal packages.

Software Ecosystems. Software ecosystem research has been rapidly growing in the past years. Several studies compared different ecosystems. Decan et al. [73, 74] conducted a quantitative empirical analysis of the similarities and differences between the evolution of package dependency networks for seven package ecosystems: *Cargo* for Rust, *CPAN* for Perl, *CRAN* for R, *npm* for JavaScript, *NuGet* for the .NET, *Packagist* for PHP, and *RubyGems* for Ruby. They introduced novel metrics to capture the growth, changeability, resusability and fragility of these dependency networks, and use these metrics to analyse and compare their evolution. Approaches [75–77] pointed out the fragility of software ecosystems and gave insights on the challenges software developers face. They examined the *Eclipse*, *CPAN* and *npm* ecosystems, focusing on what practices cause API breakages. They found that developers in fact struggle with dependency updates, even if their projects were affected by vulnerable packages.

Approaches [16, 78–81] specifically focused on the *npm* ecosystem. In work [78, 79], researchers conducted empirical investigations with millions of *npm* packages to understand (i) the widespread phenomena of micro-packages, (ii) the size of dependencies inherited by a micro-package and (iii) the costs (ie., fetch, install, load times) of using a micro-package. Kula et al. [78] suggested developers to be aware of how sensitive their third-party dependencies were to critical changes in the software ecosystem. Abdalkareem et al. [79] suggested that *npm* developers should be careful about the selection of packages and how to keep them updated. Jafari et al. [80] defined a set of dependency management issues (i.e., dependency smells) in the *npm* ecosystem. They then conducted surveys with practitioners, to identify and quantify seven dependency smells with varying degrees of popularity and investigate why smells are introduced. Wittern et al. [81] investigated the evolution of *npm* using metrics such as dependencies between packages, download count, and usage count in JavaScript applications. Their findings helped understand the evolution of *npm*, design better package recommendation engines, and can help developers understand how their packages are being used. Liu et al. [16] proposed a knowledge graph-based dependency resolution, which resolved the inner dependency relations of dependency trees, and investigated the security threats from vulnerabilities in dependency trees at a large scale. To the best of our knowledge, our work is the first attempt to diagnose and monitor the lags in propagation of vulnerability fixes in the *npm* ecosystem.

Library/Package Updates. Recent studies [25–28, 76, 82–91] focused on upgrading dependencies. Kula et al. [25] analyzed 850,000 library migrations in Maven ecosystem, and found that the projects heavily depended on these libraries, and nearly 81.5% projects had outdated libraries. Based on interviews conducted with developers, the study pointed out that 69% of participants tended to be not aware of their vulnerable dependencies. Mirhosseini et al. [26] studied about the pull request notifications to update dependencies. Their results showed that pull requests and badge notifications could reduce lags, however, developers were often overwhelmed by lots of notifications. Approaches [27, 28] studied how prevalent is semantic versioning adopted by software projects and its impacts on dependency upgrades. Dietrich et al. [27] investigated over 70 million dependencies across 17 different package managers. They did not find evidence that projects switch to semantic versioning on a large scale. Raemaekers et al. [28] performed an empirical study on potential rework caused by breaking changes in library releases and found that breaking changes had a significant impact on client libraries.

As outlined in work [76, 82–85], dependency management should include quantitative cost-benefit analysis for assisting developers to make confident decisions in updating dependency versions. Approaches [86–89] studied the API evolution in the ecosystem and characterized the impacts of evolution on dependency upgrades. To support automated upgrades, Foo et al. [90] and McCamant et al. [91] proposed static analysis techniques for automatically and efficiently checking if a library update introduced API incompatibility issues. The above work can guide further research into better practices for automated dependency management. Our work focus on how to update pivotal packages' version constraints based on semantic versioning mechanism for vulnerability remediation. Our study complemented the findings of prior work, with the similar goal of encouraging developers to update dependencies.

8 Conclusions and Future Work

In this paper, we empirically studied the scale of packages that block the propagation of vulnerability fixes in *npm* ecosystem. By digging into the evolution history of blocking chains, we characterize their common remediation patterns, which can shed light on mitigating fix propagation lags. We refined our empirical findings into a technique, to derive effective remediation strategies for pivotal packages. The evaluation demonstrated the effectiveness of our efforts as a tool implementation PLUMBER in boosting the propagation of vulnerability fixes in the ecosystem.

In the future, we plan to improve the performance of PLUMBER in four aspects: (1) *Identifying unexploitable vulnerabilities.* To effectively reduce the false alerts, PLUMBER should integrate more precise program analysis techniques in order to identify the vulnerable dependencies that could not be exploited. (2) *Compatibility analysis.* To avoid inducing incompatibility issues, PLUMBER should identify proper safe versions of packages without API breaking changes, when deriving remediation strategies. (3) *Correcting vulnerability information.* Since the vulnerability database may provide inexact information (e.g., affected versions of vulnerable package), PLUMBER should correct the vulnerability information based on effective data mining techniques. (4) *Generalizing* PLUMBER *to other ecosystems.* PLUMBER should model and collect the metadata from other ecosystems that resolving dependencies using similar semantic version syntax as *npm* (e.g., *PyPI* and *Cargo*), in order to generalize our technique to other programming language communities.

Acknowledgment

The authors express their thanks to anonymous reviewers for their constructive comments. This work is supported by National Grand R&D Plan (Grant No. 2020AAA0103504), the National Natural Science Foundation of China (Grant Nos. 61932021, 62141210, 61902056), the Major Key Project of PCL (Grant No. PCL2021A06), the Hong Kong RGC/GRF grant 16207120, MSRA grant, ITF grant (MHP/055/19, PiH/255/21), Research Grants Council (RGC) Research Impact Fund under Grant R5034-18, the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant No. BK20202001), Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01), and 111 Project (B16009). The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the Release, Adoption, and Propagation of npm Vulnerability Fixes," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–28, 2021.
[2] A. Decan, T. Mens, and E. Constantinou, "On the Impact of Security Vulnerabilities in the npm Package Dependency Network," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 181–191.
[3] X. Sun, T. Zhou, R. Wang, Y. Duan, L. Bo, and J. Chang, "Experience Report: Investigating Bug Fixes in Machine Learning Frameworks/Libraries," *Frontiers of Computer Science*, vol. 15, no. 6, pp. 1–16, 2021.
[4] J. Hu, L. Huang, T. Sun, Y. Fan, W. Hu, and H. Zhong, "Proactive Planning of Bandwidth Resource Using Simulation-based What-if Predictions for Web Services in the Cloud," *Frontiers of Computer Science*, vol. 15, no. 1, pp. 1–28, 2021.
[5] "CVE-2021-28918," https://nvd.nist.gov/vuln/detail/CVE-2021-28918, 2021, accessed: 2021-08-01.
[6] "npm Audit," https://docs.npmjs.com/cli/v6/commands/npm-audit, 2021, accessed: 2021-08-01.
[7] "Dependabot," https://dependabot.com/, 2021, accessed: 2021-08-01.
[8] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the Past–Analysing Backporting Practices in Package Dependency Networks," *IEEE Transactions on Software Engineering*, 2021.
[9] "Mqtt.js#1306," https://github.com/mqttjs/MQTT.js/issues/1306, 2021, accessed: 2021-08-01.
[10] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying Open-source License Violation and 1-day Security Risk at Large Scale," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2169–2185.
[11] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 995–1010.
[12] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-life Vulnerabilities: A Study on Firefox Evolution, Its Vulnerabilities, and Fixes," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2011, pp. 195–208.
[13] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "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, 2019, pp. 619–623.
[14] V. Piantadosi, S. Scalabrino, and R. Oliveto, "Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache Http Server and Spache Tomcat," in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 68–78.
[15] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking Known Security Vulnerabilities in Proprietary Software Systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 516–519.

[16] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the npm Ecosystem," *44th International Conference on Software Engineering (ICSE)*, 2022.

[17] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable Open Source Dependencies: Counting Those That Matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2018, pp. 1–10.

[18] I. Pashchenko, H. Plate, E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A Methodology for Counting Actually Vulnerable Dependencies," *IEEE Transactions on Software Engineering*, vol. 48, pp. 1592–1609, 2020.

[19] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm Javascript Packages," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 559–563.

[20] H. Plate, S. E. Ponta, and A. Sabetta, "Impact Assessment for Vulnerabilities in Open-source Software Libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 411–420.

[21] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[22] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities," in *30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 3041–3058.

[23] M. Di Penta, L. Cerulo, and L. Aversano, "The Life and Death of Statically Detected Vulnerabilities: An Empirical Study," *Information and Software Technology*, vol. 51, no. 10, pp. 1469–1484, 2009.

[24] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of Recurring Software Vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010, pp. 447–456.

[25] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do Developers Update Their Library Dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[26] S. Mirhosseini and C. Parnin, "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 84–94.

[27] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency Versioning in the Wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 349–359.

[28] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning and Impact of Breaking Changes in the Maven Repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[29] "Tensorflow/tfjs," https://github.com/tensorflow/tfjs, 2021, accessed: 2021-08-01.

[30] "Ethers.js," https://github.com/ethers-io/ethers.js, 2021, accessed: 2021-08-01.

[31] "Googlechrome/workbox," https://github.com/GoogleChrome/workbox, 2021, accessed: 2021-08-01.

[32] "Github Advisory DB," https://github.com/advisories, 2021, accessed: 2021-08-01.

[33] "Snyk Vulnerability DB," https://snyk.io/vuln/, 2021, accessed: 2021-08-01.

[34] "Npm Security Advisories," https://github.com/npm/npm/security/advisories, 2021, accessed: 2021-08-01.

[35] "CVSS Score," https://www.first.org/cvss/, 2021, accessed: 2021-08-01.

[36] "node-semver," https://github.com/npm/node-semver, 2021, accessed: 2021-08-01.

[37] "Ethers.js#1439," https://github.com/ethers-io/ethers.js/issues/1439, 2021, accessed: 2021-08-01.

[38] "npm Public Registry," https://docs.npmjs.com/cli/v7/using-npm/registry, 2021, accessed: 2021-08-01.

[39] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[40] "Truffle#1147," https://github.com/trufflesuite/truffle/issues/1147, 2021, accessed: 2021-08-01.

[41] "Swap-core," https://github.com/pancakeswap/pancake-swap-core, 2021, accessed: 2021-08-01.

[42] "semver," https://github.com/advisories, 2021, accessed: 2021-08-01.

[43] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A Comprehensive Study on Challenges in Deploying Deep Learning Based Software," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 750–762.

[44] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of Real Faults in Deep Learning Systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1110–1121.

[45] J. W. Creswell, *Qualitative Inquiry and Research Design: Choosing Among Five Approaches (3rd Edition)*. SAGE Publications, Inc., 2013.

[46] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–56, 2021.

[47] H. He, R. He, H. Gu, and M. Zhou, "A Large-scale Empirical Study on Java Library Migrations: Prevalence, Trends, and Rationales," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 478–490.

[48] "CVE-2021-23624," https://security.snyk.io/vuln/SNYK-JS-DOTTY-1577292, 2021, accessed: 2021-08-01.

[49] "CVE-2021-25912," https://security.snyk.io/vuln/SNYK-JS-DOTTY-1069933, 2021, accessed: 2021-08-01.

[50] "Meteorite#299," https://github.com/oortcloud/meteorite/issues/299, 2021, accessed: 2021-08-01.

[51] "React-reality#1," https://github.com/rhdeck/react-reality/issues/1, 2021, accessed: 2021-08-01.

[52] "Configtest," https://github.com/codeconsole/configtest, 2021, accessed: 2021-08-01.

[53] "Mapbox#7," https://github.com/mapbox/github-release-tools/issues/7, 2021, accessed: 2021-08-01.

[54] "Opac#1034," https://github.com/scieloorg/opac/issues/1034, 2021, accessed: 2021-08-01.

[55] "Googleapis," https://github.com/googleapis/googleapis, 2021, accessed: 2021-08-01.

[56] "Google-gax," https://github.com/googleapis/gax-nodejs, 2021, accessed: 2021-08-01.

[57] "Google-cloud/storage," https://github.com/googleapis/nodejs-storage, 2021, accessed: 2021-08-01.

[58] "Microsoft/just," https://github.com/microsoft/just, 2021, accessed: 2021-08-01.

[59] "Node-sqlite3#1493," https://github.com/mapbox/node-sqlite3/issues/1493, 2021, accessed: 2021-08-01.

[60] "Peer-id#153," https://github.com/libp2p/js-peer-id/issues/153, 2021, accessed: 2021-08-01.

[61] "Image-cropper#42," https://github.com/likeconan/react-avatar-image-cropper/issues/42, 2021, accessed: 2021-08-01.

[62] "Commit#8616b1 in learn-webpack," https://github.com/AlNuN/learn-webpack/commit/837c45edfadbac033fa7f1cb88eb420d098616b1, 2021, accessed: 2021-08-01.

[63] "Xml-crypto#231," https://github.com/yaronn/xml-crypto/issues/231, 2021, accessed: 2021-08-01.

[64] "Meow#195," https://github.com/sindresorhus/meow/issues/195, 2021, accessed: 2021-08-01.

[65] "Angular-cli#21608," https://github.com/angular/angular-cli/issues/21608, 2021, accessed: 2021-08-01.

[66] "Xml-crypto#1215," https://github.com/postmanlabs/postman-collection/issues/1215, 2021, accessed: 2021-08-01.

[67] "Cssnano#7491," https://github.com/parse-community/parse-server/issues/7491, 2021, accessed: 2021-08-01.

[68] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, no. 7, pp. 733–743, 1986.

[69] W. Cai, F. He, X. Lv, and Y. Cheng, "A Semi-transparent Selective Undo Algorithm for Multi-user Collaborative Editors," *Frontiers of Computer Science*, vol. 15, no. 5, pp. 1–17, 2021.

[70] S. Khoshnevis, "A Search-based Identification of Variable Microservices for Enterprise SaaS," *Frontiers of Computer Science*, vol. 17, no. 3, pp. 1–16, 2023.

[71] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A Scalable Approach for Vulnerable Code Clone Discovery," in *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017, pp. 595–614.

[72] H. Jang, K. Yang, G. Lee, Y. Na, J. D. Seideman, S. Luo, H. Lee, and S. Dietrich, "QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2021, pp. 66–82.

[73] A. Decan, T. Mens, and P. Grosjean, "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[74] A. Decan, T. Mens, and M. Claes, "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 2–12.

[75] C. Bogart, C. Kästner, and J. Herbsleb, "When It Breaks, It Breaks: How Ecosystem Developers Reason About the Stability of Dependencies," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 2015, pp. 86–89.

[76] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[77] C. Xu, Y. Qin, P. Yu, C. Cao, and J. Lv, "Theories and Techniques for Growing Software: Paradigm and Beyond," *SCIENTIA SINICA Informationis*, vol. 50, pp. 1595–1611, 2020.

[78] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "On the Impact of Micro-packages: An Empirical Study of the npm Javascript Ecosystem," *arXiv preprint arXiv:1709.04638*, 2017.

[79] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *Proceedings of the 2017 11th joint meeting on Foundations of Software Engineering*, 2017, pp. 385–395.

[80] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency Smells in Javascript Projects," *IEEE Transactions on Software Engineering*, 2021.

[81] E. Wittern, P. Suter, and S. Rajagopalan, "A Look at the Dynamics of the Javascript Package Ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016, pp. 351–361.

[82] S. Raemaekers, A. Van Deursen, and J. Visser, "Measuring Software Library Stability Through Historical Version Analysis," in *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 378–387.

[83] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Liu, and Y. Wu, "An Empirical Study of Usages, Updates and Risks of Third-party Libraries in Java Projects," *arXiv preprint arXiv:2002.11028*, 2020.

[84] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An Empirical Study of Third-party Library Updatability on Android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2187–2200.

[85] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "Analyzing ad Library Updates in Android Apps," *IEEE Software*, vol. 33, no. 2, pp. 74–80, 2016.

[86] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How Do Developers React to API Evolution? The Pharo Ecosystem Case," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 251–260.

[87] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the Reaction to Deprecation of Clients of 4+ 1 Popular Java APIs and the JDK," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2158–2197, 2018.

[88] J. Henkel and A. Diwan, "Catchup! Capturing and Replaying Refactorings to Support API Evolution," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.

[89] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache Community Upgrades Dependencies: An Evolutionary Study," *Empirical Software Engineering*, pp. 1275–1317, 2015.

[90] D. Foo, H. Chua, J. Yeo, M. Y. Ang, and A. Sharma, "Efficient Static Checking of Library Updates," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 791–796.

[91] S. McCamant and M. D. Ernst, "Predicting Problems Caused by Component Upgrades," in *Proceedings of the 9th European software engineering conference held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2003, pp. 287–296.

Ying Wang received her doctoral degree in software engineering from Northeastern University, China, in 2019. She is currently an associate professor at the Software College, Northeastern University, China. Her research interests include program analysis, dependency management, and software ecosystems. Her research work has been regularly published in top conferences and journals in the research communities of software engineering, including ICSE, ESEC/FSE, ASE, and TSE and has received ICSE 2021 ACM SIGSOFT Distinguished Paper Award. She received Outstanding Doctoral Dissertation Award of Liaoning province (2021), and Nominees Award for Outstanding Doctoral Dissertation of China Computer Federation (CCF) in 2020. She joined Microsoft Research Asia StarTrack Program (2020). More information about her can be found at: https://wangying-neu.github.io/.

Peng Sun is currently a Master student at the Software College, Northeastern University, under the supervision of Prof. Zhiliang Zhu. His main research interests include program analysis, dependency management, and software testing.

Lin Pei is currently a Master student at the Software College, Northeastern University, under the supervision of Prof. Hai Yu. His main research interests include program analysis, dependency management, and software testing.

Yue Yu is an associate professor in the College of Computer at National University of Defense Technology (NUDT). He received his Ph.D. degree in Computer Science from NUDT in 2016. He has won Outstanding Ph.D. Thesis Award from Hunan Province. His research findings have been published on ICSE, FSE, ASE, TSE, MSR, IST, ICSME, ICDM and ESEM. His current research interests include software engineering, data mining and computer-supported cooperative work.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the MIDDLEWARE 2013 Doctoral Symposium, FSE 2014 SEES Symposium, and COMPSAC 2017 SETA Symposium. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He is an IEEE/ACM senior member.

Shing-Chi Cheung received his Bachelor's degree in Electrical and Electronic Engineering from the University of Hong Kong, and his PhD degree in Computing from the Imperial College London. After doctoral graduation, he joined the Hong Kong University of Science and Technology (HKUST) where he is a professor of Computer Science and Engineering. He founded the CASTLE research group at HKUST and co-founded in 2006 the International Workshop on Automation of Software Testing (AST). He serves on the editorial board of Science of Computer Programming (SCP) and Journal of Computer Science and Technology (JCST). He was an editorial board member of the IEEE Transactions on Software Engineering (TSE, 2006-9) and Information and Software Technology (IST, 2012-5). He participates actively in the program and organizing committees of major international software engineering conferences. He chaired the 19th Asia-Pacific Software Engineering Conference (APSEC) in 1996, 1997 and 2012. He was the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He is an extended member of the ACM SIGSOFT executive committee. He owns four patents in China and the United States. His research interests lie in boosting the quality of software applications using program analysis, testing, debugging, synthesis, repository mining, and artificial intelligence. Target applications include Android apps, open-source software, deep learning systems, smart contracts and spreadsheets. He is an IEEE Fellow, a distinguished member of the ACM and a fellow of the British Computer Society.

Hai Yu received a B.E. degree in Electronic Engineering from Jilin University, China, in 1993 and a PhD degree in Computer Software and Theory from Northeastern University, China, in 2006. He is currently an Associate Professor of Software Engineering at the Northeastern University, China. His research interests include complex networks, chaotic encryption, software testing, software refactoring, and software architecture. At present, he serves as an Associate Editor for the International Journal of Bifurcation and Chaos, Guest Editor for Entropy, and Guest Editor for the Journal of Applied Analysis and Computation. In addition, he was a Lead Guest Editor for Mathematical Problems in Engineering during 2013. Moreover, he has served different roles at several international conferences, such as Associate Chair for the 7^{th} IWCFTA in 2014, Program committee Chair for the 4^{th} IWCFTA in 2010, Chair of the Best Paper Award Committee at the 9^{th} International Conference for Young Computer Scientists in 2008, and Program committee member for the $3^{rd} - 13^{th}$ IWCFTA and the 5^{th} Asia Pacific Workshop on Chaos Control and Synchronization.

Zhiliang Zhu received an M.S. degree in Computer Applications and a PhD degree in Computer Science from Northeastern University, China. He is currently a full professor at Software College, Northeastern University, China. His main research interests include software engineering, complexity software systems, and applications of complex network theorie. He has authored and co-authored over 130 international journal papers and 100 conference papers. In addition, he has published five books, including *Introduction to Communication* and *Program Designing of Visual Basic .NET*, etc. He is also the recipient of nine academic awards at national, ministerial, and provincial levels. Prof. Zhu has served in different capacities at many international journals and conferences. Currently, he serves as Co-Chair of the $1^{st} - 13^{th}$ International Workshop on Chaos-Fractals Theories and Applications. He is a senior member of the Chinese Institute of Electronics and the Teaching Guiding Committee for Software Engineering under the Ministry of Education.