

# On The Lag of Library Vulnerability Updates

## An Investigation into the Repackage and Delivery of a Security Fix Within The npm JavaScript Ecosystem

Bodin Chinthanet · Raula Gaikovina  
Kula · Takashi Ishio · Akinori Ihara ·  
Kenichi Matsumoto

Received: date / Accepted: date

**Abstract** Vulnerabilities in third-party libraries is a growing concern for the software developer, not only because it poses risks to the software client itself, but to the entire ecosystem. To mitigate these risks, developers are strongly recommended to update their dependencies. Recent studies show that affected developers are not likely to respond to the vulnerability threat. In this paper, we introduce the lags of updates as a causes of the slow response to the vulnerability threat within the ecosystem. To understand these lags, we use both qualitative and quantitative approaches to conduct an empirical study on how 188 fixes were repackaged and delivered across over eight hundred thousand releases of npm software packages hosted on GitHub. We report two types of lags. *Lags in repackage* occur as a vulnerability fix is more likely to be bundled with other non-related updates with about 83.33% of commits are not related. Dependency freshness has an effect on *lags in delivery*, with the lineage freshness, downstream dependencies and severity of the vulnerability all impacting lags. Moreover, lags exist in the adoption of the fix during the delivery phase. We find that clients are more likely to wait to adopt the minor fix release rather than the quicker patch fix. The identification of these two lags opens up different avenues on how to facilitate quicker fix repackage and delivery across its ecosystem.

**Keywords** Node.js, npm, Library Vulnerabilities, JavaScript

---

Bodin Chinthanet, Raula Gaikovina Kula, Takashi Ishio, Kenichi Matsumoto  
Nara Institute of Science and Technology, Japan  
E-mail: {bodin.chinthanet.ay1,raula-k,ishio,matumoto}@is.naist.jp

Akinori Ihara  
Wakayama University, Japan  
E-mail: ihara@wakayama-u.ac.jp

## 1 Introduction

Vulnerabilities in third-party libraries are a growing concern for the software developer. In a 2018 report, over *four million vulnerabilities* were alerted in over 500 thousand GitHub repositories (GitHub, 2018a). The risk of vulnerabilities is not only restricted to their clients of these libraries (also known as packages), but it also extends to the entire ecosystem. Examples include the ShellShock (Bennett, 2014) and Heartbleed (Synopsys, 2014) bug which did have a widespread damage to their ecosystem of users. With Heartbleed, the ubiquity of OpenSSL, which is in servers used in over 66 percent of sites on the Internet, with estimates about 20 percent of the secure sites on the Web, according to a recent report from McAfee<sup>1</sup>.

It is well-known that developers struggle to update their vulnerable packages. A study shows that affected developers are not likely to respond to a vulnerability threat as the updating process involves much planning, such as managing the triage and allocating developers to address the threat (Kula et al, 2018). Complementary studies depict developers having slow reactions to updating (Bavota et al, 2015; Bogart et al, 2016; Ihara et al, 2017). In particular, up till now, much empirical studies have focused on the lag of the updates, with respect to the npm JavaScript ecosystem. For instance, Kikas et al (2017) investigated the spread of vulnerabilities over package dependency networks, while other studies focused on the updates of vulnerabilities (Decan et al, 2017, 2018) and at what versions are the vulnerabilities being adopted. Bogart et al (2016) showed through developer interviews that the struggle with updates can be due to management and process factors. (Cox et al, 2015) showed that the dependency freshness is important as outdated dependencies are more likely to have security issues. Furthermore, developers use security practices to aid in the (i) prevention, (ii) detection, and (iii) response to vulnerabilities in deployed products (Williams et al, 2018). Recent research has focused on the two security practices on prevention and the detection of vulnerabilities in the ecosystem (Hejderup, 2015; Howard and Leblanc, 2001; Munaiah et al, 2017; Nguyen et al, 2016; Pashchenko et al, 2018; Ponta et al, 2018). Yet, there have been only a few works focusing on the response to vulnerabilities.

The research gap that this paper addresses is a fine-grained tracking of the vulnerability fix, to see how fixes are repackaged and delivered throughout the ecosystem. Delivery includes how the fix propagates downstream to the rest of the ecosystem. Moreover, we analyze whether or not factors such as dependency freshness does have an impact on the repackage and delivery process. Complementary to related work, we conduct a quantitative study on the lag of updates i.e., repackage and delivery of the fix within the npm JavaScript ecosystem. In an empirical study of 188 known vulnerability fixes with over 882 thousand package releases of npm packages hosted on GitHub, lags are

<sup>1</sup> from an article at <https://www.sanfordbrown.edu/Student-Life/blog/October-2014/Heartbleed-Impact-So-Far>

tracked in two ways. First, lags in the repackage phase are identified. For each vulnerability, we then manually identify and track vulnerability reports as listed on [snyk.io](https://snyk.io). Second, we identify lags in the delivery phase. The study analyzes how the fixes are delivered and adopted by the clients.

Results indicate lags in the repackage, as vulnerability fixes are more likely to be bundled with non-related updates i.e., 83.33% of commits were not related to the fix, instead of being released in a special update. For lags in the delivery, dependency freshness has an effect, with other downstream dependencies to the vulnerable package i.e., downstream dependencies and the severity of the vulnerability and the severity i.e., severity having an impact. Finally, although repackaged fixes are more likely to be released as package patches, there are lags in the adoption as clients are more likely to adopt the minor releases. Our contributions are two-fold. The first contribution is a fine-grained definition of metrics and concepts of how a vulnerability is propagated throughout an ecosystem, defined as the repackage and delivery process. The second contribution is an empirical study to confirm the lags that are barriers to library updating, especially on how to facilitate faster fix delivery throughout the package ecosystem.

The remainder of the paper is organized as follows: Section 2 describes the basic concepts of package updating and the lag of updates for a vulnerability fix. Section 3 motivates our research questions, with Section 4 describing the approaches. Section 5 presents results of the study. We then discuss implications of our results and challenges for the future in Section 6, with Section 7 showing threats to validity of the study. Section 8 shows the related works. Finally, Section 9 concludes the study.

We have prepared a replication package of our dataset, which is available at <https://github.com/NAIST-SE/Lag-of-Library-Vulnerability-Updates>.

## 2 Repackage and Delivery of a Vulnerability Fix

In this section, we propose vulnerability updates as a repackage and delivery process for when a vulnerability fix propagates across the software ecosystem. Concretely, a real-world example illustrates this process.

### 2.1 Vulnerability Fix Process

The repackage and delivery process consists of three phases, which are described in detail below:

*Phase 1: Vulnerability Awareness.* Developers of the vulnerable package become aware of the vulnerability report from sources e.g., GitHub issue (Bill, 2012), Snyk.io (Snyk, 2017). Figure 1 is an example of vulnerability information and how they are linked to the vulnerable GitHub repository. Figure 2

Vulnerability DB > npm > socket.io

## Insecure Randomness

Affecting **socket.io** package, versions <0.9.7

**CVSS SCORE**  
5.3  
MEDIUM SEVERITY

**CREDIT**  
Martin Thomson

**CVE**  
CVE-2017-16031

**CWE**  
CWE-330

**SNYK ID**  
npm:socket.io:20120323

**DISCLOSED**  
22 Mar, 2012

**PUBLISHED**  
13 Feb, 2017

**Overview**

socket.io is a node.js realtime framework server. Affected versions of the package are vulnerable to Insecure Randomness due to the cryptographically insecure Math.random function which can produce predictable values and should not be used in security-sensitive context.

**Remediation**

Upgrade socket.io to version 0.9.7 or higher.


**References**

- GitHub Changelog
- GitHub PR
- GitHub Issue
- GitHub Commit
- Google Group Forum

Fig. 1: A vulnerability report of package `socket.io` reported in `snyk.io` due to insecure randomness. The references are linkages to the GitHub repository issue, pull request and commit at <https://github.com/socketio/socket.io/issues/856>

## ID generation vulnerability #856

**Closed** martinthomson opened this issue on Apr 27, 2012 · 0 comments

 martinthomson commented on Apr 27, 2012

The generation of new IDs for new sockets is vulnerable to prediction attacks. Potentially, though highly unlikely, collisions are also possible.

Mentioned in the comments of #497.

The `Manager.generateId()` method generates identifiers using the default `Math.random()` pseudorandom number generator, which produces predictable output. Knowledge of the `socket.io` identifier is sufficient information for a client to receive information from a `socket.io` server. With the current implementation, an attacker that is able to predict subsequent identifiers can make a request and gain potentially private information.

Fig. 2: A vulnerability report for package `socket.io` reported in GitHub issue due to the insecure randomness vulnerability.

## Fix for ID generation vulnerability #856 #857

**Merged** rauchg merged 4 commits into `socketio:master` from `martinthomson:bug/856` on Apr 27, 2012

Conversation 1 Commits 4 Checks 0 Files changed 1



martinthomson commented on Apr 27, 2012

Contributor + 😊 ...

Using `crypto.randomBytes` to generate a session ID. 96 bits of randomness, 24 bits of sequence number. Using the URL safe base64 encoding [RFC 4648](#) to represent the value.

Backward compatibility for node 0.4 uses an insecure generation method with `Math.random()`. I could fix this too, but I don't believe that this version of node is still popular enough to justify the effort.

Fig. 3: A screenshot of the pull request that fixes the insecure randomness vulnerability in the `socket.io` package.

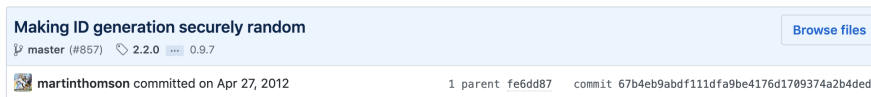


Fig. 4: A screenshot of the commit that fixes the insecure randomness vulnerability for the `socket.io` package.

is an example of a vulnerability issue that is reported in GitHub. Phase 1 is triggered when developers become aware of vulnerabilities in this report. The end of Phase 1 is triggered by developers responding to the threat. Developers discuss with vulnerability reporters to identify which code changes are required. Developers are then assigned to create the fix. Note that assigned developers can also be reporters.

*Phase 2: Vulnerability Fix Repackage.* Phase 2 is concerned with repackaged fixes in a new release. This phase is separated into three steps; (i) fixing process, (ii) reviewing process, and (iii) releasing process. The fixing process starts after developers have identified and corrected the vulnerable code. Figure 3 and Figure 4 are examples of commit that related to the fixes. The fix reviewing process is then performed to confirm correctness and coding style. If the fix is accepted, it will be repackaged and merged to the repository i.e., master branch, similar to Figure 3. On the other hand, subsequent discussions may be required to improve fix correctness. The end of Phase 2 is marked by a new version release, which is incremented and should follow semantic versioning (semver) rules (Preston-Werner, 2009) for npm packages.

*Phase 3: Vulnerability Fix Delivery.* Phase 3 essentially allows the fix to propagate to downstream clients. Downstream clients refer to clients that both

Table 1: Metrics to measure the Lags in Repackage.

Metric	Description	Rationale
Commits-In-Fix	# of commits for vulnerability fix in release	The more Commits-In-Fix, the more effort spent to fix the vulnerability
Commits-In-Release	# of commits in the release	The more Commits-In-Release, the more changes repackage in the release
Fix-to-Release-ratio	$\frac{Commits-In-Fix}{Commits-In-Release}$	A high ratio means that the release is more likely to be the fix specific release
Other-to-Release-ratio	$1 - \frac{Commits-In-Fix}{Commits-In-Release}$	A high ratio means that the release is more likely to be the non-fix release



Fig. 5: Metrics to measure the Lags in Repackage.

directly and indirectly (transitive) depend on a package. This phase starts after the fixes have been adopted by clients, thus patching the vulnerability. In some cases, the semver allows the package manager to automatically update the library, especially if the constraints are met. However, there are cases where the fixes cannot be automatically adopted due to incompatible code. Hence, developers require additional migration effort to facilitate successful adoption. The end of Phase 3 is when the adoption is successful. Projects with downstream clients will then return to Phase 2 for repacking the fixes. Phase 2 and Phase 3 will continue to propagate throughout the ecosystem until the vulnerability threat has been eliminated.

## 2.2 Lags in Repackage (Phase 2)

Table 1 and Figure 5 show the metrics that we use to investigate the lag of updates in the vulnerability fix repackage phase. We introduce three metrics to describe the repacked fix. As shown in Figure 5, we semantic versioning (semver) rules (Preston-Werner, 2009) of `VMajor.Minor.Patch` to distinguish between major (V1.0.0), minor (V1.1.0) and patch (V1.0.1) releases.

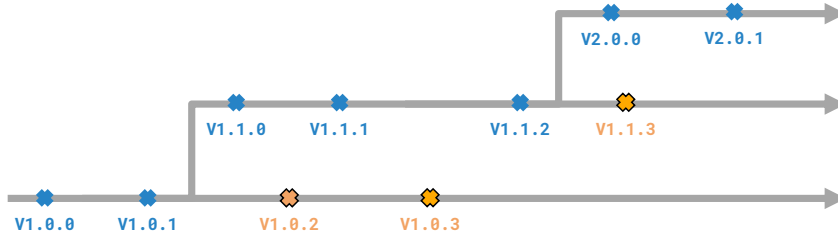


Fig. 6: Example of a Latest Lineage (LL) and a Supported Lineage (SL) metric to measure the freshness vulnerable package.

*Repackaged fix* For the repackaging metrics, we count how much of fixes get repackaged in each release commit i.e., Commits-In-Fix. The metrics include counting the fix and release commits as the fix is being delivered downstream the clients. The final metric is then the proportion of fix commits over the release commits. The ratio should tell us whether the patches are specifically for the fix or include other types of maintenance. Together with the semantic versioning, our assumption is that projects tend to be reliant on the release schedule rather than getting the fixes out as soon as possible.

### 2.3 Lags in Delivery (Phase 3)

Lags in the delivery usually refer to the release and adoption of the vulnerability fixes. In the study, we consider different factors related to the dependency freshness to analyze these lags.

*Dependency freshness of delivered fix* We consider three different lenses when we analyze the dependency freshness of the fix and how it spreads throughout the ecosystem:

1. *Lineage Freshness of Fix*: Figure 6 depicts the classification used to detect the freshness of the delivered fixes. As shown in the Figure, we classify fixed package as either being in the (1) Latest Lineage (LL) - the client has adopted any version of the package that has the latest updates or (2) Supported Lineage (SL) - the client has adopted a version that is not as fresh and is outdated. In the example, V1.0.2, V1.0.3 and V1.1.3 are SL versions. We assume that fresher fixes will have less lags in their adoption.
2. *Downstream Dependencies*: refers to the transitive dependency distance between a vulnerable library and any downstream clients i.e., 1, 2, 3 and  $\geq 4$ . For each vulnerable library, we transverse to identify all downstream clients within the ecosystem. In related works, it is referred to as transitive for Kikas et al (2017) or indirect dependencies for Hejderup (2015). Our

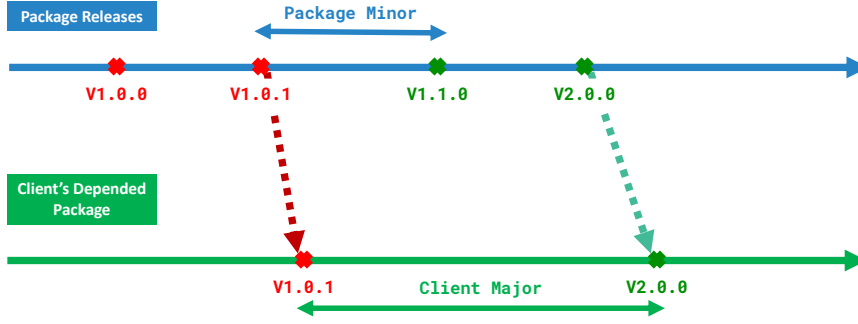


Fig. 7: Example of a lag in the delivery, caused by a delay in the adoption of package releases by version releases.

assumption is that the further away a threat, the longer it will take the fix to be adopted by a client.

3. *Severity of vulnerability* ( $H$ ,  $M$ ,  $L$ ): refers to severity of vulnerability i.e.,  $H$  = high,  $M$  = medium,  $L$  = low, as indicated in the report (See Figure 1 for an example of a medium severity vulnerability). We assume that high threats will experience less lags.

*Adoption of package release* As shown in Figure 7, the delivered fixes are classified into the following (i) *package major*: developers have released a version that may have incompatible API changes, (ii) *package minor*: developers have released a version that should have backward-compatible manner, (iii) *package patch*: developers have released a version that should have backward-compatible fixes. Note that pre-releases or special releases are not considered in this study. Our assumption is understand if fixes tend to be repackaged in a certain release type. In this example in Figure 7, we need that a vulnerable package minor was first adopted by a client major. Later, also the fix was released in the next package minor (V1.1.0), it only adopted by the client major (V2.0.0), with a package major (V2.0.0).

### 3 Research Questions

The research questions focus on the goal of understanding the response to the vulnerability, specifically what lags occur during the delivery of the fix into the ecosystem. The following three research questions are formulated:



### 3.1 ( $RQ_1$ ) What lags occur during the repackage of a fix?

We compliment the recent of works that have studied the threat of the vulnerability (Decan et al, 2018; Hejderup, 2015). Both works showed how the vulnerability threat can affect the ecosystem network. In our work, we manually explore and check how the fix is repackaged at the commit level, using our defined metrics to identify lags at this phase.

### 3.2 ( $RQ_2$ ) What effect does the dependency freshness have in reducing lags during the delivery of a fix?

( $RQ_2$ ) investigates whether dependency freshness of the vulnerability fix impacts the lag during the repackage phase. First introduced by Cox et al (2015), we explore whether the dependency freshness i.e., Lineage freshness, downstream dependencies and severity, of the fix package impacts lags in the fix. Furthermore, Decan et al (2018) showed that freshness of the updated dependency reduces the technical lag. We analyze the freshness through the lens of (a) in-direct downstream clients as Kikas et al (2017) revealed that vulnerabilities not only affect a single package but also any other downstream clients and (b) severity as Decan et al (2018) showed how the severity vulnerability may impact the lag.

### 3.3 ( $RQ_3$ ) What lags occur during the adoption of a fix?

( $RQ_3$ ) investigates what fixed versions tend to get adopted by clients. In light of the incident with the package `event-stream` in 2018, we know that downstream clients incurred lags of at least a week due to the inactivity of developers (Sparling, 2018). In this case, the knowledge of which package version was safe to adopt and what was actually adopted, could provide insights into lags during the delivery.

## 4 Approach

In this section, we present our data collection and research approaches in order to answer our three research questions. First, the collection of the dataset is presented. Then in subsequent sections, we discuss in detail each approach, data processing and analysis of the results.

### 4.1 Data Collection

The dataset is acquired from two different sources, (i) library packages in the software ecosystem and (ii) vulnerability reports of packages. We selected the npm JavaScript ecosystem (NPM, 2010) as it hosts one of the largest collection

of packages and has been the target of recent studies (Abdalkareem et al, 2017; Decan et al, 2017, 2018; Hejderup, 2015; Kikas et al, 2017). To extract ecosystem information of dependencies, similarly to Wittern et al (2016), we use the npm registry to download an offline copy of the npm ecosystem. The `package.json` meta-file is extracted to retrieve packages information in the repository, including the semantic versioning information for each package. For the vulnerability reports, we use the `snyk.io` website to extract vulnerabilities that were originally reported in the CVE and CWE databases. At the time of study, `snyk.io` had a comprehensive list of vulnerabilities that are linked to npm packages. Additionally, `snyk.io` also provides GitHub references for more detailed information e.g., submitted issues, fix commits, pull request. To match the GitHub repositories to the vulnerability, we first only selected vulnerability that already released a fix. Then, we used the package name and references to match the report with the downloaded repository automatically.

As shown in Table 2, we collected vulnerability reports that disclosed from 9 April 2009 to 16 November 2016 with 792 reports. For package repositories from GitHub, we used repository snapshot from 17 December 2017 with 154,425 repositories. Note that two sets of data have a one year lag since we want to capture more than 90% of the released fixes according to the study of Decan et al (2018). After filtering the vulnerability reports that do not release any fixes, we mapped 42,980 repositories (including all downstream clients) with 188 vulnerability reports.

We prepare three different datasets for each research question. The prepared dataset for  $(RQ_1)$  involves a manual investigation of the vulnerability reports. Since the approach requires a link, we only include the report that has at least one GitHub Pull request or Commit link. Thus, our final dataset included 131 vulnerability reports. The prepared Dataset for  $(RQ_2)$  involves investigation of the fix being published and adopted. Thus, from the 131 vulnerability reports used in  $(RQ_1)$ , we filter out reports that do not report a fix. We then classify the vulnerable packages according to the labeled package and client versions. The prepared dataset for  $(RQ_3)$  involves extracting all downstream clients that adopted the package fix. Preparation of  $(RQ_3)$  consists of two steps. First, we searched for all downstream clients that depended on the vulnerable library listed in 188 reports. We then filtered out vulnerabilities that did not have a fix released. In the end,  $(RQ_3)$  consists of 136 vulnerability reports with 74 different vulnerable packages and a total of 19,583 different downstream clients.

#### 4.2 Research Method for $(RQ_1)$

The approach to answer  $(RQ_1)$  involves the investigation of the repackaging and delivery of vulnerability fix process. Since all downloaded repositories are from GitHub, we investigated Issue, Commit and Pull Requests that are mentioned in the vulnerability report. The first step is the extraction of these linkages i.e., explicit links to the Issue, Commit or Pull Request, that are listed in

Table 2: Summary of collected dataset information.

npm JavaScript Ecosystem Information	
Repository Snapshot	17 Dec 2017
# Extracted Package GitHub Repositories	154,425
# Matched Package GitHub Repositories	42,980
# Total Dependencies (with downstream)	83,427
# Total Packages Releases	882,222
npm Vulnerability Report Information	
Disclosures Period	9 Apr. 2009~16 Nov. 2016
# Extracted Reports	792
# Matched Reports	188
High severity	47
Medium severity	128
Low severity	13
$(RQ_1)$ Dataset	
# Vul. Reports (with fix references)	131
# Vul. Reports (without fix references)	57
$(RQ_2)$ Dataset	
# Vul. Reports (with fix released)	136
# Vulnerable Packages (with fix released)	74
# Downstream Clients	19,583
$(RQ_3)$ Dataset	
# Vul. Reports (follow semver)	75
# Vulnerable Packages (follow semver)	43
# Direct Clients (follow semver)	5,133
# Filtered Clients (break semver)	7,141

the `snyk.io` vulnerability report. The links extracted are captured into three types: Issue i.e., Figure 2, Commit i.e., Figure 4, and Pull request i.e., Figure 3. Based on the links, we are able to identify the vulnerable version and is used to search for the vulnerable code commit.

To validate that developers knowingly applied the fix, two authors manually confirmed that comments in the commit log or pull request i.e., one author found the evidence and the other validated. By using this method, we were also able to discern whether fixes were applied specifically to fix the vulnerability, or were bundled with other changes. Using this method, we were able to distinguish between commits that fixed the vulnerability. Figure 8 is the GitHub comparing changes tool (Sencha Labs, 2018) that we used to confirm the set of commits that released in a fix update. By using the tool and checking the comments, we then were able to manually labeled commits as either (1) fix commits or (2) other commits. The information about the fix commits and other commits is used for the following analysis to answer the  $(RQ_1)$ .

Table 1 shows the metrics that we use to investigate the lag of update. We use the metrics that are discussed in Section 2.3. There are two kinds of dimension in our metrics, including the repackaged fix metrics i.e., Fix-

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



Fig. 8: An example of the tool used to compare differences between two releases in terms of commits.

to-Release ratio. Second, we consider the labeled release version i.e., package major, package minor and package patch, to understand how developers release the fix. Furthermore, we then manually evaluate the size of the fix change. All metrics are presented using a bar chart and a count of the number of delivered fixes by version. To validate our results, we present concrete examples from the results.

### 4.3 Research Method for ( $RQ_2$ )

Our approach to answer ( $RQ_2$ ) is to identify lags in the fix delivery through the analysis of dependency freshness as proposed in this paper (i.e., lineage freshness, downstream dependencies and severity of vulnerability). For the lineage freshness, we avoided duplicated clients while traversing the dependency tree by considering only one of the duplicated clients that more closer to the vulnerable package. This is because **npm** allows the same downstream clients to appear multiple times in the dependency tree depend on the requirement of each client. Therefore, we adopted a breadth-first search (BFS) approach for traversing to each downstream client, since we assume that vulnerable clients that are closer to the vulnerable package are more likely to update. As for the client dependency scope, we only considered an exact version and a range of version. To identify the version from a range, we chose the oldest version available in the range as it is the first adoptable client version. Furthermore, we confirm the fix by semi-automatically extracting the commit history to identify and gather meta-data of the vulnerable version and the first available fix version of clients e.g., semantic version, release date, downstream dependency. Severity is taken from the vulnerability report as shown in Figure 1.

Identifying the fix has to meet specific conditions. First it should be the version of the package depended by the client, Second, it must be released before the upper bound of reported vulnerable version. Hence, the condition for identifying the first available fix is *"the version of the package depended by*

Table 3: Summary number of filtered clients grouped by their change type.

SemVer Update Patterns	# clients
Major only	97
Minor only	320
Patch only	2,588
No change	4,136

*the client must not included in the vulnerable range disclosed in the vulnerable report".*

We use visualizations and statistics to draw our conclusions for ( $RQ_2$ ). We use box-plots and report statistics for visualizing dependency freshness trends by the lineage, downstream dependents and severity. In order to statistically validate the differences in the results, we apply Kruskal-Wallis non-parametric statistical test (Kruskal and Wallis, 1952). We test the hypothesis that ‘*clients with fresher dependencies are likely to experience less lags*’. We investigate the effect size using Cliff’s  $\delta$ , which is a non-parametric effect size measure (Romano et al, 2006). Effect size are analyzed as follows: (1)  $|\delta| < 0.147$  as Negligible, (2)  $0.147 \leq |\delta| < 0.33$  as Small, (3)  $0.33 \leq |\delta| < 0.474$  as Medium, or (4)  $0.474 \leq |\delta|$  as Large. To analyze Cliff’s  $\delta$ , we use the `cliffsDelta` package<sup>2</sup>.

#### 4.4 Research Method for ( $RQ_3$ )

Our approach to answer ( $RQ_3$ ) is an investigation of lags in the fix delivery through the analysis of fix adoption in clients. In order to track the adoption, we first extract any clients that depend on the vulnerable package including the version of dependency similar to ( $RQ_2$ ). We then filter those packages that did not follow semantic versioning. This is performed by a semi-automatic checking the version history of packages to identify patterns of releases. For example, we remove packages that only release major versions. Table 3 shows the filtering out packages that did not follow semantic versioning. Our key assumption is that clients that utilized the semantic versioning i.e., releasing major, minor, and patch versions, are more likely to follow the semantic version. As a result, 7,141 packages were filtered out from the dataset. This pre-processing helps to remove any bias in the data. In the end, our final dataset for ( $RQ_3$ ) consists of 75 vulnerability reports, 43 vulnerable libraries, and 5,133 different clients.

Our analysis involves understanding how the delivered fix is adopted by clients as shown in Figure 7. We use visualizations and statistics to draw conclusions about lags in the delivery phase. We use box-plots and report the median values for visualizing trends how long it takes for clients to adopt the delivered fixes. In order to statistically validate our results, we apply Pearson’s chi-squared test ( $\chi^2$ ) (Pearson, 1900) with the null hypothesis ‘*the package release type and the client adoption type are independent*’. To support the

<sup>2</sup> <https://github.com/neilernst/cliffsDelta>

Table 4: Shows the distribution of investigated delivered fixes in  $(RQ_1)$ .

Package Version Label	# of Delivered Fixes
Package Major	9 (6.87%)
Package Minor	47 (35.88%)
Package Patch	75 (57.25%)
	131

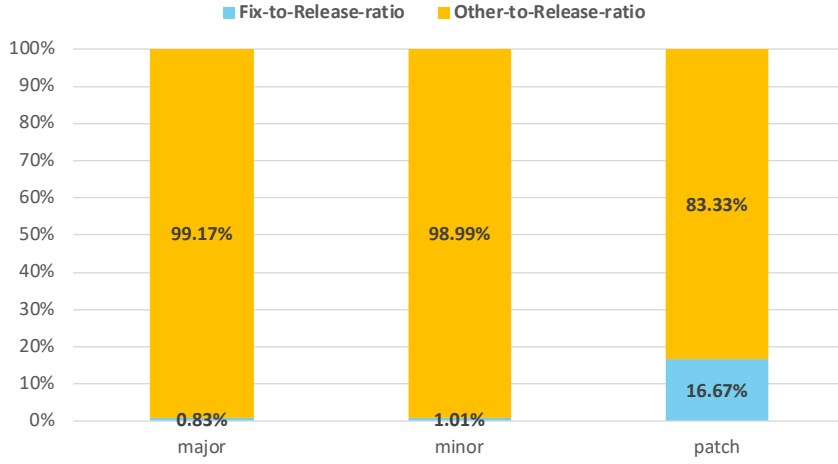


Fig. 9: As shown in the figure, a fix is usually accompanied with other commits in the repackaged release.

hypothesis, we investigate the effect size using Cramér's  $V$  ( $\phi'$ ), which is a measure of association between two nominal categories (Cramér, 1946). According to Cohen (1988), effect size is analyzed as follows: (1)  $\phi' < 0.07$  as Negligible, (2)  $0.07 \leq \phi' < 0.20$  as Small, (3)  $0.20 \leq \phi' < 0.35$  as Medium, or (4)  $0.35 \geq \phi'$  as Large. To analyze Cramér's  $V$ , we use the `researchpy` package<sup>3</sup>.

## 5 Results

In this section, we present the results for  $(RQ_1)$ ,  $(RQ_2)$ , and  $(RQ_3)$ . The results include key findings and then finally answers each research question.

<sup>3</sup> <https://pypi.org/project/researchpy/>

### 5.1 ( $RQ_1$ ) What lags occur during the repackaging of a fix?

Table 4 and Figure 9 show two findings: that (i) vulnerability fixes are often repackaged and released as patches and (ii) vulnerability fixes are repackaged as a minor part of the update.

To support the first finding, Table 4 shows that out of the 131 vulnerability reports that were evaluated, 57.25% of vulnerability fixes are repackaged in a patch version release. On the other hand, we find that 6.87% and 35.88% of fixes were released as either the major update and a minor update. This evidence indicates that vulnerability fixes have a lag in the repackaging, regardless whether the fix is simple or not. One example is the `faye` patch fix<sup>4,5</sup>. In this case, the patch fixes insecure default values. Under closer manual inspection, we find that there is only one commit that updates the default value of variables<sup>6</sup>. Yet, the patch itself includes the other fixes<sup>7</sup>. In this case, the other commits were fixing mutable variable usage inside a closure, which had no relation to the fix.

Figure 9 shows that fixes are usually repackaged in a patch and are small in size. The results show that the median average Fix-to-Release-ratio per release is 0.83%, 1.01%, and 16.67% package major, minor, and patch update. We find that the fix itself is small. For instance, we find that the average size of the fix in terms of the line of code is eight lines. One example is the `npm`<sup>8</sup> package. The vulnerability that attacked the package was due to a symlink attack and was caused by a predictable tmp folder name. The actual fix to mitigate the vulnerability took seven lines of code<sup>9</sup>. Based on our results, we now return to answer the research question ( $RQ_1$ ) *What lags occur during the repackaging of a fix?*:

Repackaged vulnerability fixes constitute for a small portion of commits and are released as a patch. We find that (i) the vulnerability fixes are often released as patch fix and (ii) 83.33% of commits in the repackaged fix are not related to the vulnerability fix.

### 5.2 ( $RQ_2$ ) What effect does the dependency freshness have in reducing lags during the delivery of a fix?

Table 5, Table 6, and Figure 10 show the effect of dependency freshness for lags in delivery. Instead, we find that (i) clients further downstream are more likely to adopt the fix slower when vulnerabilities affect the latest lineage, and

<sup>4</sup> <https://snyk.io/vuln/npm:faye:20121107>

<sup>5</sup> <https://github.com/faye/faye/compare/0.8.8...0.8.9>

<sup>6</sup> <https://github.com/faye/faye/commit/e407e08c68dd885896552b59ce65503be85030ad>

<sup>7</sup> <https://github.com/faye/faye/commit/23e8d5e489a6480e47c62e403f5ebbcddbc14f5d>

<sup>8</sup> <https://snyk.io/vuln/npm:npm:20130708>

<sup>9</sup> <https://github.com/npm/npm/commit/f4d31693>

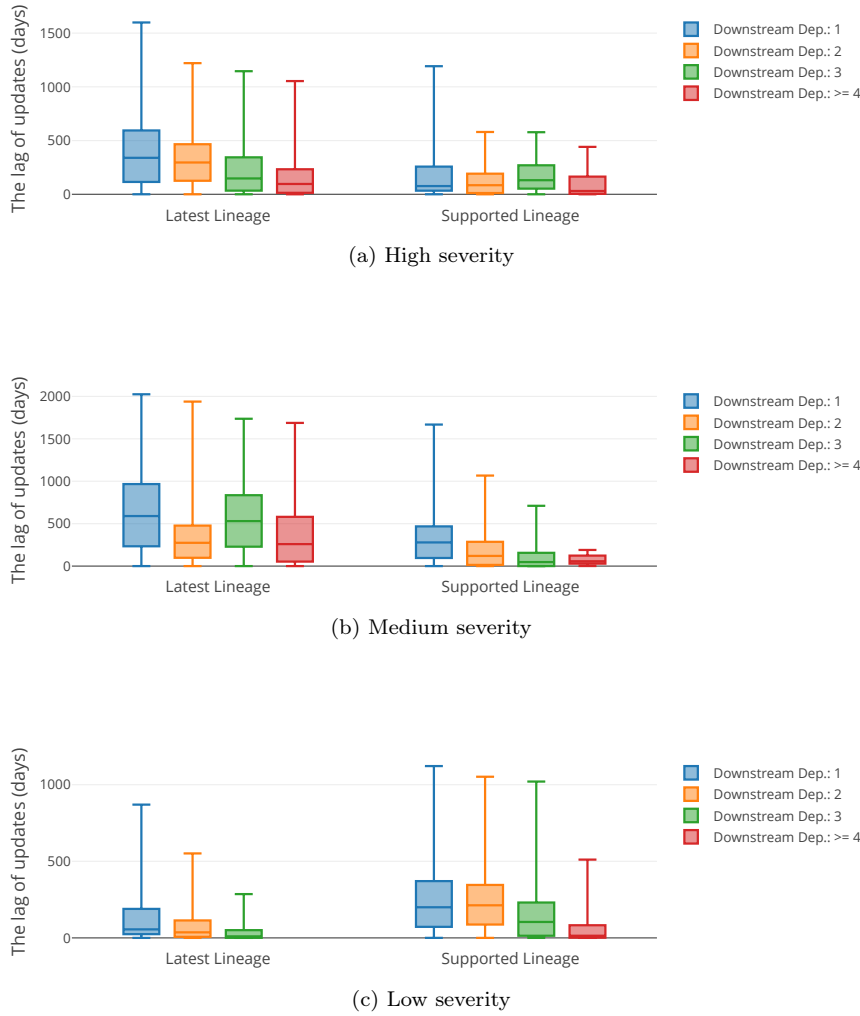


Fig. 10: Trends indicate that freshness does not have an impact on whether there are lags in adoption of fixes.

(ii) fixes with a medium vulnerability suffers the most lags. Figure 10 shows that there are visual differences in the lags between the Latest Lineage (LL) and the Supported Lineage (SL), but lineage freshness does not impact the amount of lags when the severity is taken into account.

In fact, the results show interesting trends in terms of the how much lag is reduced as the fix propagates throughout the ecosystem i.e., measured by



the downstream dependencies, and how the lowest severity has an a reduced lag compared to medium severity. This result indicates that adoption time decreases i.e., fewer lags of updates, for clients that are further downstream as shown in Table 5. Table 6 shows that downstream clients adopt the fix slower i.e., more lags of updates, when a vulnerability affects the latest lineage freshness, especially for high and medium severity. In the case of low severity, downstream clients quickly adopt fixes i.e., fewer lags of updates, when the vulnerability affects the latest lineage.

For the statistical evaluation, our hypothesis on whether ‘*clients with fresher dependencies are likely to experience less lags*’ was supported. Table 6 shows that the difference between the Latest Lineage and the Supported Lineage is statistically significant i.e., Downstream Dep. 1-3 for medium and low severity and 1-2 for high severity. However, only clients that affected by low severity vulnerability accept our hypothesis. It also shows that the effect sizes of the

Table 5: Summary statistics show that lags are decreasing as downstream dependencies increases.

	Downstream Dep.	<i>n</i>	Max	Mean	Median	SD
H	1	2916	1599.11	326.89	257.94	286.69
	2	7553	1220.52	302.84	293.23	207.76
	3	2205	1145.58	193.18	131.94	187.77
	4	457	1053.70	136.14	79.98	152.99
	5	89	558.97	167.29	115.88	163.91
	6	7	399.81	103.95	55.12	129.11
M	1	16681	2025.16	546.26	474.41	416.44
	2	15614	1938.84	312.13	269.31	262.85
	3	14354	1736.58	535.04	531.12	349.95
	4	5108	1687.70	347.23	259.08	330.37
	5	1322	1454.57	238.65	121.72	275.71
	6	217	780.60	220.80	168.72	209.30
	7	7	427.94	168.08	3.74	193.49
L	1	1581	1121.80	194.88	126.13	187.00
	2	4184	1052.17	219.85	204.73	169.11
	3	519	1020.77	134.21	84.39	138.58
	4	67	511.13	68.44	14.09	98.55

Table 6: The comparison of the lag of updates between clients that use the latest lineage and the supported lineage of libraries (LL = median of the latest lineage, SL = median of the supported lineage, \* = statistically significant), (Medium, Large) = effect size.

Downstream Dep.	H	M	L
1	LL > SL * (Medium)	LL > SL * (Medium)	LL < SL * (Medium)
2	LL > SL * (Large)	LL > SL * (Medium)	LL < SL * (Large)
3	LL > SL	LL > SL * (Large)	LL < SL * (Medium)
≥ 4	LL > SL	LL > SL	-

Table 7: Summary Statistics that show the semantic version releases of both the vulnerability fix and the adoption by the client.

Delivered Fix (# fixes)	Adoption	Mean (%)	Median (%)	SD (%)
package major (3)	client major	61.24	87.88	43.42
	client minor	0.00	0.00	0.00
	client patch	0.00	0.00	0.00
	client removed	38.76	12.12	43.42
package minor (26)	client major	22.99	5.77	29.03
	client minor	53.54	60.93	32.13
	client patch	6.30	0.00	15.83
	client removed	17.18	15.38	17.31
package patch (46)	client major	34.28	22.14	36.40
	client minor	36.14	25.70	35.39
	client patch	21.30	4.22	31.09
	client removed	8.28	4.17	11.24

Table 8: Contingency table of package delivery by client adoption

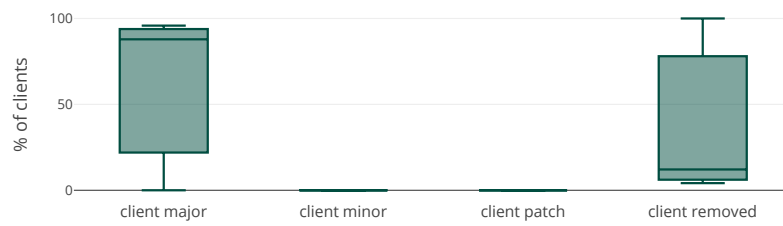
	Client major	Client minor	Client patch	Client removed	All
Package major	93	0	0	10	103
Package minor	82	292	27	90	491
Package patch	365	767	280	171	1583
All	540	1059	307	271	2177

differences are noticeable i.e., medium to large. Based on our results, we now return to answer the research question ( $RQ_2$ ) *What effect does the dependency freshness have in reducing lags during the delivery of a fix?*:

Dependency freshness has an effect on lags in the fix delivery. We find that (i) downstream clients affected by low severity vulnerabilities tends to suffer fewer lags and (ii) fixes with a medium vulnerability suffers the most lags. In cases of high and medium severity, the latest lineage tends to suffer more lags. However, the supported lineage tends to suffer more lags when clients affected by low severity.

### 5.3 ( $RQ_3$ ) What lags occur during the adoption of a fix?

Table 7, Table 8, and Figure 11 show two findings related to the lag: (i) the repackaged fixes tend to be released as patches, however, packages are waiting minor releases before adoption. In response to the first finding, Table 7 shows that the majority of repackaged fixes are released as patches. This is actually 46 out of the 75 delivered fixes that were analyzed. Putting together the results



(a) Package major adopted by clients



(b) Package minor adopted by clients



(c) Package patch adopted by clients

Fig. 11: The adoption distribution of package (a) major, (b) minor, and (c) patch by clients.

Table 9: Results of statistical test for ( $RQ_3$ )

Statistic	Value
Pearson’s chi-squared test ( $\chi^2$ )	327.4104
p-value	$1.09 \times 10^{-67}$
Cramér’s V ( $\phi'$ )	0.2742

of ( $RQ_1$ ), we can say that although the fix is coupled with other commits, it is more likely to be released as a patch. Through manually analysis of three cases from the package major release, we find that waiting for dependent new features or a complex discussion on the vulnerability fix may lead to a fix being released in a major version. From the example of `connect`<sup>10</sup>, we find that developers repackaged a lot of updates that were related to the vulnerability fix, testing, documentation before releasing in a new major version<sup>11</sup>. In the example of `knockout`<sup>12</sup>, developers planned to create a new feature of the package, so they made an update to support more secured processing in the package<sup>13</sup>. From the example of `ws`<sup>14</sup>, developers have a long discussion about the fix of vulnerability since it might affect the performance of the package. In the end, they decided to merged the fix for the major release<sup>15</sup>.

For the second finding, show that clients tend to adopt fixes according to the same semantic versioning as the packages. The results provides two implications. First, it implies that clients are more likely to release the package as a minor release rather than a patch release. As shown in Table 7, client patches adoptions are not a popular option (4.22%) since clients tend to adopt the fix in major (22.14%) and minor (25.70%) releases. Secondly, we find that fixes released as patches are not as popular. Surprisingly, this shows that clients are unlikely to adopt releases based on very specific fixes, even if they are security related. We find that in cases of package major (87.88%) and package minor (60.93%) are more likely to be adopted compared to a patch release. One interesting observation is that for the three packages that released their fix in a major release, the client was more likely to release their fix as a major revision or simply remove the dependency altogether.

For the statistical evaluation of our findings we shown in Table 9 that our hypothesis is rejected due to the result of Pearson chi-squared test. Thus, there is *an association between the package release type and the client adoption*. From the Cramér’s V effect size ( $\phi'$ ), we got a value of 0.272 which show the medium level of association. Based on our results, we now return to answer research question ( $RQ_3$ ) *What lags occur during the adoption of a fix?*:

<sup>10</sup> <https://snyk.io/vuln/npm:connect:20120107>

<sup>11</sup> <https://github.com/senchalabs/connect/compare/1.8.4...2.0.0>

<sup>12</sup> <https://snyk.io/vuln/npm:knockout:20130701>

<sup>13</sup> <https://github.com/knockout/knockout/pull/1022>

<sup>14</sup> <https://snyk.io/vuln/npm:ws:20160920>

<sup>15</sup> <https://github.com/websockets/ws/pull/832>

Although repackaged fixes are more likely to be released as package patches, there is a lag in the adoption as clients are more likely to adopt the minor releases.

## 6 Implications

This section discusses three main implications based on our results in  $(RQ_1)$ ,  $(RQ_2)$ , and  $(RQ_3)$ .

**1) *The release cycle impacts lags when releasing and adopting vulnerability fixes.*** According to the results of  $(RQ_1)$ , vulnerability fixes are minor with less than 16.67% of commits in new update related to the vulnerability fix. We suspect that vulnerability fix repackaging is a cause for lags. Hence, developers of the vulnerable packages are recommended to release fixes as soon as they have applied the fix, if not they should highlight these fixes when packaging the fix. Additionally, from ten randomly selected vulnerabilities, we found that discussions between developers did not include explicit mention of the vulnerability i.e., GitHub Issue, Commit, and Pull request. Since developers repackaged the fix with other updates, developers may have been unaware.

The implication for both researchers and practitioners is that the release cycle of both the library being fixed and its client plays an important role in how fast the updates will propagate the ecosystem.

**2) *Highlighted security fixes tend to get adopted quicker.*** According to  $(RQ_1)$  and  $(RQ_3)$ , vulnerability fixes are released as a patch i.e., 57.25% of the fixes. However, clients are more likely to adopt the fixes by updating their library major and minor versions i.e., 19.17% and 22.67%, than the patch version i.e., 4.88%. Security fixes need to be highlighted in the fix, as a possible reason is for failure to update because client developers are more interested in major features that are highlighted in an update. Recently, some open source communities start to make tools to highlight the vulnerability problems in the software ecosystem. GitHub ([GitHub, 2017](#)) made a new function for notifying the new vulnerability from the dependency list of clients. Also, npm ([NPM, 2018](#)) made a new command for listing the vulnerability information in downloaded dependencies of clients. Therefore as stated in  $(RQ_1)$ , explicit patches identification and highlighting of the security vulnerability is needed to speed up delivery.

The implication for both researchers and practitioners is that a client is more likely to adopt a fix, if they are aware that it is highlighted.

**3) Regardless of dependency freshness, fixes suffer lags due to planning the adoption.** From our first finding of  $(RQ_2)$ , clients that are further down the transitive dependency tend to adopt quicker than those that are closer to vulnerable package. A possible reason is developers of downstream clients have more time to plan and prepare their projects before fixes are available to adopt. In terms of security, according to the official documentation of npm (NPM, 2016), when a security threat is identified, the following severity policy is put into action:

- P0: Drop everything and fix!
- P1: High severity, schedule work within 7 days.
- P2: Medium severity, schedule work within 30 days.
- P3: Low severity, fix within 180 days.

Surprisingly, our results show evidence that low severity fixes are adopted just as quick as high severity fixes. A possible reason for the quicker adopting of low severity could be because the fix is easier to integrate into the application.

The implication for both researchers and practitioners is that package developers in npm seem to require additional time before updating their dependencies.

## 7 Threats to Validity

Key threats to the study were classified as either internal and external threats.

**Internal Validity.** Internal threats to validity refer to the concerns of the our study setup. We discuss three threats. The first threat is the correctness of tools and techniques used in this study. We use the listed dependencies and version number as defined in the `package.json` meta-file. The threat is that sometimes some dependencies are not listed or semantic version is invalid and vice-versa. However, we applied a filter to remove clients that do not follow the semantic versioning. We believe that the threats of invalid data are minimal.

The second threat is the tools used to collect and implement our defined terminology. We used python libraries *numpy*, *scipy*, *gitpython*, and *semantic-version*. To mitigate this, we carefully confirmed our results by manually validating the results for  $(RQ_1)$  and  $(RQ_2)$ , then also manually validating some of the results with the listed dependencies and dependent statistics on the npm website. For the existing tool for suggesting the fixed version like `npm-audit`, we found that this tool was inappropriate to use in this work due to its limitation. The first limitation is that `package-lock.json` file is required for analyzing repositories. We found that only 2.27% of repositories in this work have that file. The second one is that the tool always uses the latest information from the server for the suggestion. Unlike our work, we analyzed the data available in the historical snapshot.

As the correctness of dependency relations is dependent on getting all dependencies, the final internal threat is the validity of our collected data. In

this study, our ecosystem is made up of packages and clients that were either or affected directly or transitively by at least a single vulnerability. We also consider only normal dependencies listed in the `package.json` file to make sure that the packages are used by clients not only for testing. Hence, other types of dependencies including: (1) `devDependencies`, (2) `peerDependencies`, (3) `bundledDependencies`, and (4) `optionalDependencies` are ignored in this study. We are confident that the result of ( $RQ_2$ ) are not affected by invalid data.

**External Validity.** External threats to validity refer to the generalization concerns of the study’s results. The main external threat is the generality of other results to other ecosystems. In this study, we focused solely on the npm JavaScript ecosystem. However, our analysis is applicable to other ecosystems that have similar package management system e.g., PyPI for Python, Maven for Java. Immediate future plans include studying the lags in other ecosystems. Another threat is the sample size of the analyzed data. In this study, we analyzed only 188 matched vulnerability reports from 792 extracted reports on *snyk.io*. This small size of sample data might not be able to represent the population. However, we are confident of the data quality and reduced bias. This is because we followed strict methods to validate by two authors for ( $RQ_1$ ) and ( $RQ_3$ ) data.

**Construct Validity.** Construct threats to validity refer to the selection of factors in the experiment. The key threat is that there may be other factors apart from the three factors of dependency freshness, severity of vulnerability, and downstream dependencies. We believe the bias are minimal and validate our selection of these factors since they were discussed in prior studies related to dependency updating and security vulnerability impact i.e., measuring of dependency freshness from Cox et al (2015), impact of a vulnerability to transitive dependencies from Kikas et al (2017), responding to a vulnerability from NPM (2016).

## 8 Related Work

Complementary the related work introduced throughout the paper, in this section, we discuss some key related work with respect to updating software libraries, malware and vulnerabilities, and mining-related studies.

### 8.1 On Updating Libraries

These studies relate to the migration of libraries to the latest versions of libraries. With new libraries and newer versions of existing libraries continuously being released, managing a system’s library dependencies is a concern on its own. As outlined in Bogart et al (2016); Raemaekers et al (2012); Teyton

et al (2012), dependency management includes making cost-benefit decisions related to keeping or updating dependencies on outdated libraries. Additionally, Bavota et al (2015); Hora et al (2015); Ihara et al (2017); Robbes et al (2012); Sawant et al (2016) showed that updating libraries and their APIs are slow and lagging. Kula et al (2018) found that such update decisions are not only influenced by whether or not security vulnerabilities have been fixed and important features have been improved, but also by the amount of work required to accommodate changes in the API of a newer library version. Recently, Abdalkareem et al (2017); Decan et al (2017); Mirhosseini and Parnin (2017) also studied dependency issues at the ecosystem level.

**Our work** focuses on the lag of the fix instead of vulnerability itself. Our work complements also the findings of prior work, with the similar goal of encouraging developers to update.

## 8.2 On Malware and Vulnerabilities

These studies relate to security vulnerability within the software ecosystem from various aspects. Recent studies try to look at the impact of vulnerability within the ecosystem by looking at dependencies usage (Decan et al, 2018; Hejderup, 2015; Kikas et al, 2017; Lauinger et al, 2017; Linares-Vásquez et al, 2017). These results show that vulnerability able to affect the wide range of clients in the ecosystem. There is a study about the relationship between bugs and vulnerabilities, to conclude that the relationship is weak (Munaiah et al, 2017). In order to increase the awareness of developers to the security vulnerability, some studies try to create a tool to detect and alert vulnerability when it disclosed (Cadariu et al, 2015; GitHub, 2018b). There is a study about addressing the over-estimation problem for reporting the vulnerable dependencies in open source communities and the impact of vulnerable libraries usage in the industry (Pashchenko et al, 2018). Additionally, some studies try to predict the vulnerability of software system by analyzing source code (Alhazmi et al, 2007; Chowdhury and Zulkernine, 2011; Shin and Williams, 2008).

**Our work** takes a look at the vulnerable code and its fix at the commit level, instead of the release level in prior studies. Furthermore, we define our dependency freshness metrics based on the transitive dependencies, lineage and severity, thus it is at the ecosystem level. Prior work instead only analyzes the direct dependencies.

## 8.3 Mining-related studies

These studies relate to the mining technique in the software repository and software ecosystem. The first step of software repository mining is data collection and extraction. Researchers need to have the data sources and know about which part of the data can use in their work. In the case of the npm package repository, we can extract the information of packages from `package.json`



meta-file (Mirhosseini and Parnin, 2017; Wittern et al, 2016). In the case of the security vulnerability, we can collect data from Common Weakness Enumeration (CWE) (Mitre Corporation, 2018b) and Common Vulnerabilities and Exposures (CVE) (Mitre Corporation, 2018a) database (Alhazmi et al, 2007; Cadariu et al, 2015; Chowdhury and Zulkernine, 2011; Lauinger et al, 2017; Linares-Vásquez et al, 2017; Munaiah et al, 2017). To study the issues within the software ecosystem, we also define the traversal of the downstream clients by using the dependency list of clients. These studies introduce some technique to model the dependency graph (Bavota et al, 2015; Hejderup, 2015; Kikas et al, 2017).

*Our work* uses similar mining techniques to extract the dependencies as well as construct the ecosystem. In our work, we manually extract and investigate the commits to understand the contents of the fix.

## 9 Conclusion

Developers in the software ecosystem now have over a thousand libraries at their fingertips, attributed by large networks of dependent packages that provide the functionality to any client application. In this study, we use the case study of npm JavaScript ecosystem with the vulnerability report from snyk.io to understand how clients respond to the fix and how fast it is spread. Based on our results, we conclude that:

- lags in repackage occur as vulnerability fixes are more likely to be bundled with other non-related updates i.e., about 16.7% of commits are related to the fix.
- lags in delivery are caused by clients that are more likely to adopt the minor fix than adopt the patch fix.
- other factors such as downstream dependencies and severity do have an impact on lags.

The identification of these two lags and their implications opens up different avenues for both researchers and practitioners to investigate. We envision our work to bring insights that work towards reducing lags. Potential future avenues for researchers includes a developer survey to better understanding the reason for developers, creation of tools highlight vulnerability fixes automatically as they propagate through the ecosystem and research for prioritizing the vulnerabilities, to name a few.

## Acknowledgment

This work was supported by the Japan Society for Promotion of Science (JSPS) KAKENHI Grant Numbers 18H04094, JP15H02683, and 17H00731.

## References

- Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp 385–395
- Alhazmi OH, Malaiya YK, Ray I (2007) Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security* 26(3):219–228
- Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Software Engineering (EMSE)* 20(5):1275–1317
- Bennett JT (2014) Shellshock in the Wild - Shellshock in the Wild. <https://www.fireeye.com/blog/threat-research/2014/09/shellshock-in-the-wild.html>, (Accessed on 07/29/2018)
- Bill (2012) Static middleware crashes node server when URL contains a trailing backlash - Issue #452 - senchalabs/connect. <https://github.com/senchalabs/connect/issues/452>, (Accessed on 02/14/2019)
- Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In: Proceedings of the 24th International Symposium on the Foundations of Software Engineering (FSE), pp 109–120
- Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 516–519
- Chowdhury I, Zulkernine M (2011) Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture (JSA)* 57(3):294–313
- Cohen J (1988) *Statistical Power Analysis for the Behavioral Sciences*. Routledge
- Cox J, Bouwers E, Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), pp 109–118
- Cramér H (1946) *Mathematical Methods of Statistics*. Princeton University Press
- Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in OSS packaging ecosystems. In: Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 2–12
- Decan A, Mens T, Constantinou E (2018) On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories (MSR), pp 181–191
- GitHub (2017) About security alerts for vulnerable dependencies. <https://help.github.com/articles/>

- [about-security-alerts-for-vulnerable-dependencies/](#), (Accessed on 02/14/2019)
- GitHub (2018a) How security alerts are keeping your code safer. <https://github.blog/2018-03-21-security-alerting-a-first-look-at-community-responses/>, (Accessed on 02/14/2019)
- GitHub (2018b) Viewing and updating vulnerable dependencies in your repository. <https://help.github.com/articles/viewing-and-updating-vulnerable-dependencies-in-your-repository/>, (Accessed on 02/14/2019)
- Hejderup J (2015) In Dependencies We Trust: How vulnerable are dependencies in software modules? Master's thesis, Delft University of Technology
- Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How Do Developers React to API Evolution? The Pharo Ecosystem Case. In: Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME), pp 251–260
- Howard M, Leblanc D (2001) Writing Secure Code. Redmond, WA, USA
- Ihara A, Fujibayashi D, Suwa H, Kula RG, Matsumoto K (2017) Understanding When to Adopt a Library: A Case Study on ASF Projects. In: Proceedings of the 13th International Conference on Open Source Systems (OSS), pp 128–138
- Kikas R, Gousios G, Dumas M, Pfahl D (2017) Structure and Evolution of Package Dependency Networks. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp 102–112
- Kruskal WH, Wallis WA (1952) Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association* 47(260):583–621
- Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? *Empirical Software Engineering (EMSE)* 23(1):384–417
- Lauinger T, Chaabane A, Arshad S, Robertson W, Wilson C, Kirda E (2017) Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In: Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)
- Linares-Vásquez M, Bavota G, Escobar-Velásquez C (2017) An Empirical Study on Android-related Vulnerabilities. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR), pp 2–13
- Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 84–94
- Mitre Corporation (2018a) CVE - Common Vulnerabilities and Exposures (CVE). <https://cve.mitre.org/>, (Accessed on 02/14/2019)
- Mitre Corporation (2018b) CWE - Common Weakness Enumeration. <https://cwe.mitre.org/>, (Accessed on 02/14/2019)
- Munaiah N, Camilo F, Wigham W, Meneely A, Nagappan M (2017) Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. Em-

- pirical Software Engineering (EMSE) 22(3):1305–1347
- Nguyen VH, Dashevskiy S, Massacci F (2016) An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering (EMSE)* 21(6):2268–2297
- NPM (2010) npm. <https://www.npmjs.com/>, (Accessed on 02/14/2019)
- NPM (2016) Responding to Security Threats and Critical Updates. <https://www.npmjs.com/policies/security#responding-to-security-threats-and-critical-updates>, (Accessed on 02/14/2019)
- NPM (2018) Auditing package dependencies for security vulnerabilities. <https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities>, (Accessed on 02/14/2019)
- Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F (2018) Vulnerable Open Source Dependencies: Counting Those That Matter. In: *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp 42:1–42:10
- Pearson K (1900) X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50(302):157–175
- Ponta SE, Plate H, Sabetta A (2018) Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software. In: *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*, pp 58–68
- Preston-Werner T (2009) Semantic Versioning 2.0.0. <https://semver.org/>, (Accessed on 02/14/2019)
- Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pp 378–387
- Robbes R, Lungu M, Röthlisberger D (2012) How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem. In: *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, pp 56:1–56:11
- Romano J, Kromrey JD, Coraggio J, Skowronek J, Devine L (2006) Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen’s d indices the most appropriate choices. In: *annual meeting of the Southern Association for Institutional Research, Citeseer*
- Sawant AA, Robbes R, Bacchelli A (2016) On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In: *Proceedings of the 32th International Conference on Software Maintenance and Evolution (ICSME)*, pp 400–410
- Sencha Labs (2018) Comparing 1.8.4...2.0.0 - senchalabs/connect. <https://github.com/senchalabs/connect/compare/1.8.4...2.0.0>, (Accessed on 02/14/2019)

- Shin Y, Williams L (2008) An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 315–317
- Snyk (2017) Denial of Service (DoS) - npm:connect. <https://snyk.io/vuln/npm:connect:20120107>, (Accessed on 02/14/2019)
- Sparling A (2018) I don't know what to say. - Issue #116 - dominictarr/event-stream. <https://github.com/dominictarr/event-stream/issues/116>, (Accessed on 02/14/2019)
- Synopsys (2014) Heartbleed Bug. <http://heartbleed.com/>, (Accessed on 07/29/2018)
- Teyton C, Falleri JR, Blanc X (2012) Mining Library Migration Graphs. In: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), pp 289–298
- Williams L, McGraw G, Migue S (2018) Engineering Security Vulnerability Prevention, Detection, and Response. *IEEE Software* 35(5):76–80
- Wittern E, Suter P, Rajagopalan S (2016) A Look at the Dynamics of the JavaScript Package Ecosystem. In: Proceedings of the 13th International Conference on Mining Software Repositories (MSR), pp 351–361