Which Is Better For Reducing Outdated and Vulnerable Dependencies: Pinning or Floating?

Imranur Rahman, Jill Marley, William Enck, Laurie Williams
North Carolina State University
{irahman3, jahmad5, whenck, lawilli3}@ncsu.edu

***Abstract*—Developers consistently use version constraints to specify acceptable versions of the dependencies for their project. *Pinning* dependencies can reduce the likelihood of breaking changes, but comes with a cost of manually managing the replacement of outdated and vulnerable dependencies. On the other hand, *floating* can be used to automatically get bug fixes and security fixes, but comes with the risk of breaking changes. Security practitioners advocate *pinning* dependencies to prevent against software supply chain attacks, e.g., malicious package updates. However, since *pinning* is the tightest version constraint, *pinning* is the most likely to result in outdated dependencies. Nevertheless, how the likelihood of becoming outdated or vulnerable dependencies changes across version constraint types is unknown. *The goal of this study is to aid developers in making an informed dependency version constraint choice by empirically evaluating the likelihood of dependencies becoming outdated or vulnerable across version constraint types at scale.* In this study, we first identify the trends in dependency version constraint usage and the patterns of version constraint type changes made by developers in the npm, PyPI, and Cargo ecosystems. We then modeled the dependency state transitions using survival analysis and estimated how the likelihood of becoming outdated or vulnerable changes when using *pinning* as opposed to the rest of the version constraint types. We observe that among outdated and vulnerable dependencies, the most commonly used version constraint type is *floating-minor*, with *pinning* being the next most common. We also find that *floating-major* is the least likely to result in outdated and *floating-minor* is the least likely to result in vulnerable dependencies. Based on our findings, we recommend that developers use any kind of *floating* constraint with lockfiles to balance the tradeoffs of *pinning* and *floating*.**

***Index Terms*—Dependency management, Version constraint, Software supply chain security, Empirical software engineering**

I. Introduction

In open-source software (OSS) ecosystems, developers can specify *version constraints* for a given dependency on which version to depend [1]. *Pinning* and *floating* are two types of specifying *version constraints* for a dependency. For example, developers can depend *only* on one dependency version (e.g., "1. 2. 0" of a dependency), which is called *pinning*. Alternatively, they can specify to depend on *version ranges* of dependency (e.g., "∧1. 2. 0" matches $1.2.0 <= \text{version} < 2.0.0$ of the dependency), which is called *floating*.

In practice, developers have to balance the trade-off between pinning and floating. With *pinning*, developers can have a deterministic build of their project, which minimizes the likelihood of *breaking changes* [2] occurring from the dependencies' published versions. However, developers lose the automatic updates with bug fixes, feature improvements, and security fixes when *pinning* is used. Suppose a *pinned* dependency is found to be vulnerable according to a security advisory. In this case, developers have to manually update the vulnerable pinned version to a fixed version of the dependency, ideally after auditing the impact of changes. Until the fixed dependency version is adopted, the project might be exploited in the wild. *Pinning* can also lead to packages having outdated dependencies as soon as the upstream developers release a new dependency version. Having outdated dependencies is risky since they may contain bugs or vulnerabilities [3].

On the other hand, *floating* can auto-update the dependency version, including the security fixes. However, auto-update can introduce breaking changes [4], [5] if the upstream developers do not follow semantic versioning (SemVer) appropriately [6]. For example, developers can choose to *float* the patch version while keeping major and minor versions fixed for a dependency. However, if the upstream developers remove a method in a patch release, automatically adopting this release may cause breaking changes to the downstream packages [7]. *Floating* can also have negative security consequences. For example, an attacker may compromise an OSS package and release a malicious version that is auto-adopted by downstream packages [8]–[10]. In that case, the attacker might be able to exploit the downstream packages.

Developers consistently choose version constraint types while building and releasing their packages' versions, but may not realize the impact of their choices. Conventional wisdom suggests *pinning* should lead to more outdated dependencies than the rest of the version constraint types since *pinning* is the tightest version constraint type. However, it remains unknown if *pinning* leads to more vulnerable dependencies in-the-wild. In addition, He et al. [11] conducted a study to quantify the cost of maintenance and security benefits using *pinning* and *floating*. We build upon their work with an empirical study on which version constraints are more likely to result in outdated or vulnerable dependencies. Developers do not know *how much more likely* pinning is to result in outdated or vulnerable dependencies compared to the rest of the version constraint types.

The goal of this study is to aid developers in making an informed dependency version constraint choice by empirically evaluating the likelihood of dependencies becoming outdated or vulnerable across version constraint types at scale. Our in-the-wild analysis can help developers make an informed choice about the version constraint type, considering the security

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00229

and maintenance implications of their selection. To reach our goal, we design the following research questions: **RQ1:** *What is the frequency distribution of different version constraint types in the npm, PyPI, and Cargo ecosystems?* **RQ2:** *How frequently do developers change the version constraint type for dependencies, and how does the change result in dependencies being outdated or vulnerable?* **RQ3:** *How do pinning and floating affect the dependencies' time to become vulnerable? How do pinning and floating affect the time it takes for dependencies to become outdated?*

To answer **RQ1**, we use package release data from npm, PyPI, and Cargo, and security advisories to analyze developers' declared version constraints for dependencies. To answer **RQ2**, we consider four transitions in the state of dependencies: (1) updated to outdated; (2) outdated to updated; (3) remediated to vulnerable; and (4) vulnerable to remediated. We analyze how frequently developers intentionally change the version constraint type, which then leads to the four transitions. To answer **RQ3**, we model the dependency state transitions using survival analysis to figure out which version constraint type is more likely to result in outdated or vulnerable dependencies. We use the Cox proportional hazards model to model the <package, dependency> relationships and the dependency state transitions to estimate the relative impact of version constraint type on dependency state.

In summary, our study contributes: (1) an empirical study of dependency version constraints and their prevalence in npm, PyPI, and Cargo; (2) a quantitative analysis of dependency version constraints changes to show trends made by developers in version constraint type; (3) a statistical survival analysis to show the relative likelihood of becoming outdated or vulnerable dependencies across version constraint types. We share our replication package in a Zenodo repository [12].

II. Background and Definitions

A. Definitions

An open source software (OSS) **package** is a collection of code that provides certain functionality to its users and that is often available in software ecosystem registries (e.g., npm, PyPI) for reuse, modification, and sharing freely [13]. When the package is reused by a project or another package, we call the former package a direct **dependency**. For example, in Listing 1, postcss is the package of interest. postcss depends on several other packages, such as commander, webpack, and lodash, and these packages are thus considered dependencies in this context. Based on the **version constraint** specified by the package for the dependencies (e.g., in Listing 1), the package management software applies a *dependency resolution algorithm* to decide the exact versions of dependencies to be used in the project allowed by **version constraints**, and we refer to this process as **dependency resolution**.

When the package uses a version of the dependency that is not the latest available dependency version, we call it an **outdated dependency**. When the used dependency version is deemed vulnerable according to a security advisory and a fixed dependency version for the advisory is available, we call it a **vulnerable dependency**. We do not consider unfixed vulnerabilities in our study, since without a fixed version from upstream, there is no easy way for the downstream developers to mitigate the vulnerability.

```
{ "name": "postcss",
  "version": "0.1.2",
  "dependencies": {
    "commander": "<4.0.0 >1.2.3", \\ fixed-ranging
    "webpack": ">=1.0.0", \\ floating-major
    "meow": "5.x.x", \\ floating-minor
    "jiti": "~3.1.1", \\ floating-patch
    "config": "<9.4.0", \\ at-most
    "dotenv": "5.0.1", \\ pinning
    "postcss-core": "^1.2.0 || ^2.0.1", \\ or-
    expression
    "cheerio": "!1.2.1", \\ not-expression
    "mocha": ">2.5.0 !2.15 !2.16", \\ complex-
    expression
    "lodash": "git+ssh://git@github.com:lodash/
    lodash.git#v.4.11", \\ unclassified
  }, "devDependencies": {...}, ...
}
```

Listing 1. An example package.json file inspired by postcss to illustrate different types of version constraints available in npm, PyPI, and Cargo. More details are present in Section II and in Figure 1.

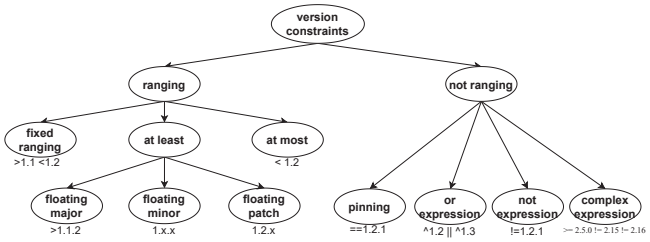

Fig. 1. Taxonomy of version constraint types with examples.

B. Version Constraint Types

We define a fine-grained categorization of version constraint types below, using the example in Listing 1. The relative restrictiveness of the constraint types is illustrated in Fig. 2.

1 Ranging. If the version constraint contains any kind of version ranges, we call the constraint *ranging*.

1.1 Fixed ranging. If the version constraint has a fixed version range with upper and lower bounds, we call the version constraint *fixed ranging*. For example, postcss specified $< 4.0.0 > 1.2.3$ for commander in Listing 1 and this version constraint will resolve to $[> 1.2.3, < 4.0.0]$.

1.2 At least. If the version constraint specifies the lower bound and the upper bound is implicit with an operator, we call this version constraint *at least*. *Floating-major*, *floating-minor*, and *floating-patch* are the three subcategories of *at-least*.

1.2.1 Floating-patch. If the version constraint of a dependency is specified in a way that the package gets an automatic update when the dependency releases a patch version, we call the constraint *floating-patch*. For example, postcss specified $\sim 3.1.1$ for jiti in Listing 1, and $\sim 3.1.1$ will resolve to $[>= 3.1.1, < 3.2.0]$.

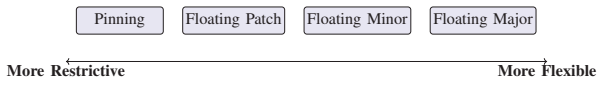

Fig. 2. Spectrum of version constraint types.

1.2.2 Floating-minor. If the version constraint of a dependency is specified in such a way that the package gets an automatic update when there is a new minor or patch released by the dependency, we call the constraint *floating-minor*. For example, postcss specified $5.x.x$ for meow in Listing 1, and $5.x.x$ will resolve to $[>= 5.0.0, < 6.0.0]$.

1.2.3 Floating-major. If the version constraint of a dependency is specified in such a way that the package gets an automatic update when there is any new release by the dependency, we call the constraint *floating-major*. For example, postcss specified $>=$ 1.0.0 for webpack in Listing 1, and $>= 1.0.0$ will resolve to any version greater than or equal to 1.0.0.

1.3 At most. If the version constraint specifies the upper bound but no lower bound, we call this version constraint type *at most*. For example, postcss specified $< 3.4.0$ for config and $< 3.4.0$ will resolve to any version less than 3.4.0.

2 Not ranging. If the version constraint does not contain any kind of ranging, we call that *not ranging*.

2.1 Pinning. If the version constraint of a dependency is fixed to one single version, we call it *pinning*. For example, postcss specified 5.0.1 version constraint for dotenv in Listing 1, and the version constraint will resolve to only version 5.0.1.

2.2 OR expression. If the version constraint contains a logical OR operator, we call this *or expression*. In Listing 1, postcss specified an *or expression* "$\wedge 1.2.0 \; || \wedge 2.0.1$" for postcss-core.

2.3 NOT expression. If the version constraint is specified with the logical NOT operator and without any other operator, we call this *not expression*. For example, postcss specified "!1.2.1" for cheerio in Listing 1 and the version constraint will resolve to any version of cheerio except 1.2.1.

2.4 Complex expression. If the version constraint contains multiple boolean expressions and multiple operators, or a combination of multiple above types, we call this *complex expression*. For example, postcss specified a boolean expression "$> 2.5.0$!2.15 !2.16" for mocha in Listing 1.

2.5 Unclassified. If the version constraint cannot be placed into any one of the above categories, we call the constraint *unclassified*. Examples of *unclassified* category could be referring to a version from other repositories (e.g., from GitHub). For example, postcss specified $git + ssh : //git@github.com : lodash/lodash.git\#v.4.11$ for lodash in Listing 1.

III. Methodology

A. Data Collection

1) Package Metadata: We collected the package dependency relationship information and package version release information from deps.dev [14] for the npm, PyPI, and Cargo ecosystems on 2024-08-20. We chose the three ecosystems for diversity: npm is the largest (5.12M npm vs 715k Maven packages), PyPI is the oldest (PyPI was introduced in 2003 vs Maven Central in 2005), and Cargo is the newest among the major software ecosystems. Package dependency relationship information contains ecosystem, package name, version metadata, e.g., package.json file, and the used direct dependencies along with specified version constraint. Deps.dev has been used in similar prior studies to collect package metadata in software ecosystems [15]–[19]. After collecting the data, we had $2,603,314$ npm, $274,720$ PyPI, and $122,069$ Cargo packages before applying any inclusion-exclusion criteria. To ensure construct quality, we manually inspected a sample of 50 packages, confirming version and dependency metadata against public package registries. We found deps.dev data to be accurate and consistent with registry data.

2) Security Advisories: We collected security advisories (CVE data) from the CISA Open Source Vulnerabilities (OSV) database, osv.dev [20] for npm, PyPI, and Cargo packages on 2024-09-12. We chose OSV since OSV pulls data from multiple vulnerability feeds, e.g., GitHub Security Advisories [21], PyPA, GoVulDB [22], for multiple ecosystems and provides the data in a unified OSV format, which facilitates cross-ecosystem analysis. We have $2,192$ npm, $3,767$ PyPI, and 989 Cargo security advisories, in total $6,948$. After collecting the data, we converted it to an SQL table with *advisory identifier*, *ecosystem*, *vulnerable package name*, *version where the vulnerability was introduced*, and *version where the vulnerability was fixed*. If the vulnerability contains multiple vulnerable version ranges and multiple corresponding fixed versions, we separate the vulnerability into multiple SQL rows where each row corresponds to one SemVer vulnerable version range and one fixed version to facilitate our analysis.

3) Package SourceRank: We examined one characteristic for packages: the SourceRank score, which is available from libraries.io. Libraries.io is used by other researchers as a data source [23]–[25]. The Package SourceRank score indicates the package quality, popularity, and community metrics calculated in libraries.io dataset [26], [27]. SourceRank depends on several factors, including the presence of a README file, license, adherence to SemVer, recent updates, and the number of contributors. We downloaded this data on 2025-01-11 to use as an additional inclusion criterion for **RQ3**.

4) Package Inclusion Criteria: We begin with an initial dataset of $3,000,103$ ($2,603,314$ npm, $274,720$ PyPI, and $122,069$ Cargo) packages collected from deps.dev. Our first step is to apply three inclusion criteria: ***(1)*** the package must be at least two years old, which is operationalized by the time difference between the first and last version release; ***(2)*** the package must have at least one residual activity (e.g., one version release) in the last two years; and ***(3)*** the package needs to have at least one dependency. Our criteria are inspired by Miller et al. [28]'s criteria of abandoned packages since we want to exclude the abandoned packages from our analysis. Moreover, "two years" is a commonly used criterion to measure whether a package is maintained or not [29], [30]. However, our package selection criteria may miss packages that are less than two years old or have had no activity in

TABLE I
RUNNING EXAMPLE OF HEXO PACKAGE WITH ONE OF IT'S DEPNDENCY MOMENT.

row	pkg	pkg version	dep	dep constraint	constraint type	dep version	dep highest rel.	**Interval start**	**Interval end**	***updated***	***remediated***
	...	...	...	...	...	...	...	...	...	...	...
116	hexo	3.0.1	moment	2.9.0	pinning	2.9.0	2.9.0	2015-04-06	2015-04-09	true	true
117	hexo	3.0.1	moment	2.9.0	pinning	2.9.0	2.10.2	2015-04-09	2015-05-13	false	true
118	hexo	3.0.1	moment	2.9.0	pinning	2.9.0	2.10.3	2015-05-13	2015-05-20	false	true
119	hexo	3.1.0	moment	$\sim$ 2.10.3	floating-patch	2.10.3	2.10.3	2015-05-20	2015-05-20	true	true
	...	...	...	...	...	...	...	...	...	...	...
122	hexo	3.1.1	moment	$\sim$ 2.10.3	floating-patch	2.10.6	2.10.6	2015-07-28	2016-01-02	true	true
123	hexo	3.1.1	moment	$\sim$ 2.10.3	floating-patch	2.10.6	2.11.0	2016-01-02	2016-01-09	false	true
124	hexo	3.1.1	moment	$\sim$ 2.10.3	floating-patch	2.10.6	2.11.1	2016-01-09	2016-02-03	false	true
125	hexo	3.1.1	moment	$\sim$ 2.10.3	floating-patch	2.10.6	2.11.2	2016-02-03	2016-02-28	false	false
126	hexo	3.2.0	moment	$\sim$ 2.11.2	floating-patch	2.11.2	2.11.2	2016-02-28	2016-03-07	true	true
...	...	...	...	...	...	...	...	...	...	...	...

the last two years (e.g., feature complete packages [31]). The number of packages after these inclusion criteria is 163,207 (117,129 npm, 42,777 PyPI, and 3,301 Cargo packages). Among these packages, 22,513 (17,263 npm, 5,158 PyPI, and 92 Cargo) packages had at least one vulnerable dependency.

B. Dependency Resolution And Data Preparation

After collecting the data and applying inclusion criteria, we use the methodology presented by Rahman et al. [30] to initially prepare our data for further analysis. We split each <package, dependency> relationship into multiple intervals based on the version release by the package or by the dependency. By definition, no new version of the package or the dependency is released during each interval.

An example of these constructed intervals ('interval start' and 'interval end') is shown in Table I. The dependency resolution at each interval takes only the dependency versions available at the beginning of the interval to ensure the dependency resolution accounts for the historical version releases by the dependency. We use deps.dev [14] to conduct the dependency resolution with this additional constraint and populate the 'dep version' column of Table I. As identified in Pinckney et al. [32], npm has a time-travelling feature (`--before` argument) which can be used to resolve historical dependency resolution. However, PyPI or Cargo does not have this feature. To the best of our knowledge, deps.dev is the only service with a dataset that resolves historical dependencies (time-travelling feature) for npm, PyPI, and Cargo. Historical dependency resolution for multiple ecosystems is a challenging task, and there is no ground truth for verifying the accuracy of historical dependency resolution by deps.dev. We manually conducted the historical dependency resolution for 20 packages' versions and their dependencies and created a ground truth. We then compared our resolution with the resolution provided by deps.dev, and we found that the data from deps.dev matched our ground truth. In addition, dependency constraints with SemVer version qualifiers (e.g., pre-releases, build metadata) are typically not fetched through dependency resolution using version ranges, unless explicitly specified. So, we exclude these SemVer version qualifiers from our dependency resolution method.

We then fill up the boolean columns 'updated' and 'remediated' to mark if the package uses an outdated or vulnerable dependency by the following conditions. The resolved version of the dependency is considered *outdated* ('updated'=false) if the resolved dependency version is not the highest SemVer version of the dependency present at the start of the interval. Similarly, the resolved version of the dependency is considered *vulnerable* ('remediated'=false) if the resolved version is within the vulnerable version range for some vulnerability and a fixed version is available at the beginning of the interval.

C. ***RQ1***: *What is the frequency distribution of different version constraint types in the npm, PyPI, and Cargo ecosystems?*

This RQ consists of two parts: (a) What version constraint types typically occur in general?; and (b) What version constraint types occur in outdated or vulnerable dependencies?

To answer (a), we analyzed the package metadata for all version releases of packages in our dataset. We classified the used version constraint ('dep constraint') for every package version into one of the version constraint types described in Section II-B using regular expressions and store that information as another column of the SQL table. With SQL query, we aggregated the used version constraint type in three granularities: number of unique packages using the type, number of unique <package, dependency> using the type, and number of unique <package, package version, dependency> using the type.

To answer (b), which is to understand how dependency-version constraint types relate to outdated or vulnerable dependencies, we analyzed the time intervals during which packages in our dataset depended on outdated or vulnerable versions. First, we filtered the time intervals with 'updated'=false on prepared data and calculated another column 'time duration' by subtracting 'interval start' from 'interval end' column. The result was that for each of the rows, we had the used version constraint type and the associated time duration. We aggregated the time durations and computed the relative time durations for each version constraint type. By quantifying the weighted time spent with outdated dependencies across constraint types, we aim to identify which types are most associated with packages' outdatedness. We followed the same

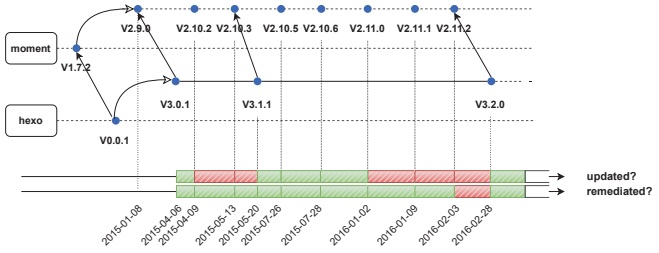

Fig. 3. Illustration of dependency state transition using hexo's dependency relationship with moment. Green time quantum indicates 'yes', and red time quantum indicates 'no' in both 'updated?' and 'remediated?' axis.

steps with 'remediated'=false for quantifying which types are most associated with packages' vulnerability.

D. *RQ2: How frequently do developers change the version constraint type for dependencies, and how does the change result in dependencies being outdated or vulnerable?*

This RQ explores how changes in version constraints affect whether a dependency becomes outdated or vulnerable. First (**RQ2**a), we want to understand how frequently developers modify the *type of version constraint*. Then (**RQ2**b), we analyze the trends in these version constraint type changes.

RQ2a. To understand what *changes* in version constraint and/or version constraint types lead to a package's dependencies being outdated or vulnerable (or vice versa), we analyze four types of state transitions between consecutive time intervals: (1) updated to outdated; (2) outdated to updated; (3) remediated to vulnerable; and (4) vulnerable to remediated.

We illustrate the state transition with the running example in Fig. 3. According to our time intervals model, such transitions can occur due to one of the following two events: (i) the dependency releases a new version, which the package may or may not adopt, and (ii) the package releases a new version, which may or may not include changes to dependency constraints.

The above two events can be categorized into four cases based on whether the version constraint type was changed:

ⓐ The dependency released a new version. For example, if the package depends on a pinned version of a dependency (e.g., 1.2.0) and the dependency releases a new version 1.2.1, the package is considered outdated with the release of 1.2.1.

ⓑ The package released a new version with the *same dependency version constraint and same constraint type*.

ⓒ The package released a new version with a *different dependency version constraint but retained the same constraint type* (e.g., remained "pinned" or "floating-minor").

ⓓ The package released a new version with a *different dependency version constraint and a changed constraint type* (e.g., switched from "pinned" to "floating-minor").

Among these four scenarios, ⓐ and ⓑ are the cases without developers' intervention. Developers changed the version constraint in ⓒ, and developers changed both the version constraint and version constraint type ⓓ. So, ⓒ and ⓓ combined represent the cases where developers intervened, and that intervention resulted in a dependency state transition.

RQ2b. We further analyze the type ⓓ cases, where the package *changes its dependency constraint type*, since these shifts offer insights into how changes made by developers result in updated or remediated dependencies. We examine the prevalence of such ⓓ cases across all four transition types. For each of the dependency state transitions, we filter out the two consecutive rows for a <package, dependency> pair where the package has a different version constraint ('dep version' column) and a different version constraint type ('constraint type' column) in the second row. We store what changed in such consecutive rows (e.g., *pinned* to *floating-patch*) for the dependency state transition, and aggregate the type change pattern for each of the dependency transitions to compute the prevalence. These trends reveal how constraint type changes result in dependency freshness and security, directly informing the impact of version constraint types.

E. *RQ3: How do dependencies' version constraint type affect the dependencies' time to become vulnerable? How do dependencies' version constraint type affect the dependencies' time to become outdated?*

Survival Analysis. To answer this RQ, we utilized survival analysis, a statistical methodology exploring time-to-event data. The expected duration of software projects, or time to termination or delivery of a project, is a popular application of survival analysis in software engineering [33], [34]. We are interested in observing the time to become vulnerable and the time to become outdated.

Model Selection. The Cox proportional hazards model is a survival method utilized to examine the relationship between the time until an event occurs and a set of explanatory variables [35]. The use of this model enables researchers to estimate the relative risk between groups, which is expressed in hazard ratios. The hazard function, also known as the hazard rate, is the conditional probability of an individual experiencing the event of interest, given the event has not yet occurred up to that point in time [36]. The hazard ratio is the ratio of the hazard rate for the treatment group to that of the baseline group [37].

In our study, we are interested in evaluating the relationship between dependencies' version constraint types and the hazards of becoming outdated or vulnerable. We constructed two models: one with time to vulnerability as the event and one with time to outdated as the event. The covariates in our models were the version constraint types. We chose *pinning* as the baseline group as *pinning* is the most restrictive type, allowing for a straightforward interpretation of the relative hazard ratios. Because several dependencies transitioned between constraint types (for example, 85 dependencies transition from *floating-major* to *pinning*), we used the time-varying Cox proportional hazards model (from Python's lifelines [38] package), which enables covariates to change their values over time [39].

Dataset construction We filtered our dataset for the top 1000 packages of each ecosystem (a total of 3000 packages across the three ecosystems) based on SourceRank scores

TABLE II
VERSION CONSTRAINT DIVERSITY BY VERSION CONSTRAINT TYPE AND ECOSYSTEM

Version Constraint Type	Cargo			npm			PyPI		
	Unique Packages	Unique (pkg, dep)	Unique (pkg, pkgver, dep)	Unique Packages	Unique (pkg, dep)	Unique (pkg, pkgver, dep)	Unique Packages	Unique (pkg, dep)	Unique (pkg, pkgver, dep)
floating-minor	**3,268 (89.12%)**	**6,674 (92.22%)**	**99,877 (95.29%)**	**105,826 (62.63%)**	**614,392 (71.86%)**	**34,092,953 (66.72%)**	249 (0.47%)	487 (0.25%)	5,749 (0.13%)
pinning	141 (3.85%)	192 (2.65%)	2,986 (2.85%)	41,216 (24.39%)	180,851 (21.15%)	15,384,917 (30.11%)	9,016 (17.06%)	47,036 (24.33%)	1,286,540 (29.40%)
floating-patch	111 (3.03%)	173 (2.39%)	1,302 (1.24%)	9,533 (5.64%)	28,500 (3.33%)	1,100,620 (2.15%)	3,375 (6.38%)	11,551 (5.98%)	237,391 (5.43%)
fixed-ranging	125 (3.41%)	176 (2.43%)	535 (0.51%)	795 (0.47%)	1,383 (0.16%)	75,018 (0.15%)	12,823 (24.26%)	48,638 (25.16%)	1,265,353 (28.92%)
at-most	1 (0.03%)	1 (0.01%)	1 (0.00%)	136 (0.08%)	187 (0.02%)	3534 (0.01%)	3,291 (6.23%)	5,523 (2.86%)	74,123 (1.69%)
complex-expression	3 (0.08%)	3 (0.04%)	3 (0.00%)	19 (0.01%)	29 (0.00%)	626 (0.00%)	1,773 (3.35%)	3,551 (1.84%)	49,113 (1.12%)
or-expression	0 (0.00%)	0 (0.00%)	0 (0.00%)	1,225 (0.73%)	1,859 (0.22%)	40,702 (0.08%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
floating-major	18 (0.49%)	18 (0.25%)	113 (0.11%)	10,220 (6.05%)	18,757 (2.19%)	402,644 (0.79%)	**22,087 (41.78%)**	**76,232 (39.43%)**	**1,453,788 (33.23%)**
not-expression	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	246 (0.47%)	296 (0.15%)	3,449 (0.08%)
total	3,667	7,237	104,817	168,970	854,958	51,101,014	52,860	193,314	4,375,506

obtained from the libraries.io dataset. After filtering, our initial survival dataset consisted of 16,223 unique dependencies from the <ecosystem, package, dependency> relationship, with over 1.5M rows of data. Each row of data consisted of the dependency ID, an interval start time, an interval end time, and the version constraint type. We removed any dependency where the interval starting time was the same as the interval ending time, as these rows may cause a spurious result. Our final survival dataset consisted of 16,094 unique dependencies and contained data points for both changed constraints and unchanged constraints. The dependent variables in our models were whether or not the dependency was outdated and whether or not the dependency was vulnerable.

TABLE III
OCCURRENCE PERCENTAGE OF VERSION CONSTRAINT TYPES IN OUTDATED AND IN VULNERABLE DEPENDENCIES.

Version Constraint Type	Cargo		npm		PyPI	
	Outdated (%)	Vulnerable (%)	Outdated (%)	Vulnerable (%)	Outdated (%)	Vulnerable (%)
floating-minor	**95.55**	**99.09**	**67.74**	**56.20**	0.16	0.07
pinning	2.64	0.04	27.79	37.73	**56.02**	**65.83**
floating-patch	1.33	0.81	4.27	5.93	8.14	7.37
fixed-ranging	0.45	0.06	0.08	0.07	29.57	22.73
at-most	0.02	-	0.02	0.05	4.56	3.55
complex-expression	0.00	-	0.00	0.00	0.59	0.45
or-expression	-	-	0.10	0.03	-	-
floating-major	-	-	-	0.00	0.95	0.00
not-expression	-	-	-	-	0.02	0.00

IV. RESULTS

A. *RQ1: What is the frequency distribution of different version constraint types in the npm, PyPI, and Cargo ecosystems?*

To answer this RQ, we analyze the version constraint types used across npm, PyPI and Cargo. We divide our analysis into two parts: *(a) What version constraint types typically occur in general?* Table II shows how frequently different version constraint types appear in each ecosystem. We count the use of each version constraint type in three granularities: the number of unique packages, unique <package, dependency> pairs, and unique <package, version, dependency> tuples. We can see that *floating-minor* is the most commonly used in npm and Cargo. *Floating-minor* is used over 99k times in <package, version, dependency> granularity in Cargo, 34M times in npm, and 6k times in PyPI. *Pinning* is the second most frequently used type in Cargo (3k times), npm (15M times), and PyPI (1.2M times). However, in PyPI, *floating-major* is the most commonly used, followed by *pinning* and *fixed-ranging*, which differ from npm and Cargo. *Floating-patch*, *at-most*, and *complex-expression* appear less frequently. Ecosystem developers have certain version constraint types that are unique to that ecosystem. For example, *or-expression* occurs only in npm and *not-expression* occurs only in PyPI.
(b) What version constraint types typically occur in outdated and in vulnerable dependencies? We computed the prevalence of constraint types associated with outdated dependency versions and vulnerable dependency versions. Table III shows the distribution of version constraint types among outdated and among vulnerable dependencies. We can see that *floating-minor* and *pinning* account for the majority of time intervals with outdated and with vulnerable dependencies. *Floating-minor* represents 95% of the dependencies in Cargo but accounts for 95% outdated and 99% vulnerable dependencies (*over-represented*). Also, *floating-minor* represents 67% of the dependencies in npm, but accounts for 67% outdated and 56% vulnerable dependencies (*under-represented*) in npm. *Pinning* represents 30% of the dependencies in npm but accounts for 28% outdated (*under-represented*) and 38% vulnerable dependencies (*over-represented*). In addition, *pinning* represents 29% of the dependencies in PyPI but accounts for 56% outdated (*over-represented*) and 66% vulnerable dependencies (*over-represented*). Moreover, *fixed-ranging* represents 29% dependencies in PyPI but accounts for 30% outdated (*over-represented*) and 23% vulnerable dependencies (*under-represented*). However, *fixed-ranging* accounts for a small portion of outdatedness and vulnerability in Cargo and npm, given that *fixed-ranging* constitutes 0.51% of Cargo and 0.14% of npm dependencies. Over-representation indicates the constraint is more likely to result in outdated or vulnerable dependencies in that ecosystem. Similarly, under-representation indicates the constraint is less likely to result in outdated and vulnerable dependencies in that ecosystem.
The rest of the constraint types contribute much less to outdatedness or vulnerability, each with less than 6%. Less frequent version constraint types (from **RQ1**a), e.g., *or expression*, *not expression*, and *complex expression*, are less frequent in outdated and vulnerable dependencies as well.
Why is *floating-minor* the most used in outdated dependency? First of all, *floating-minor* is the most frequently used dependency constraint type. Since *floating-minor* is the most represented in the dependency constraints, *floating-minor* is

TABLE IV
PREVALENCE OF CONSTRAINT TYPE CHANGES LEADING TO DEPENDENCY STATE TRANSITIONS.

Transition Type	ⓐ	ⓑ	ⓒ	ⓓ
1. Updated → Outdated	246	1,016	47,795	22,887
2. Outdated → Updated	5,645	85	5,222,276	67,681
3. Remediated → Vulnerable	16	1,271	594	1,133
4. Vulnerable → Remediated	51	0	19,549	3,829
Total count	5,958	2,372	5,290,214	95,530
Column %	0.11%	0.04%	98.09%	1.77%

expected to be the most represented in both outdated and vulnerable dependencies. Second, based on our definition of outdated dependency, we compared the used dependency version in each interval with the highest available version of the dependency, which is considered the "ideal" dependency version in our case. This choice of "ideal" dependency version leads to *floating-minor* being the most used in outdated dependencies. If a package uses *floating-minor* for a dependency and the dependency releases a new major version, the package will continue to have "outdated" dependency until the package releases a new version adopting the newest major version of the dependency. Similarly, if the upstream developers maintain multiple major branches, and the downstream developers use *floating-minor* with an older major branch, the package will always be considered "outdated" for that dependency.

Why *floating-minor* is the most used in vulnerable dependency? For each row of our dataset, we examine if the used dependency version is vulnerable and whether a fixed version is available at the start of the interval; if yes, we consider the used dependency version vulnerable. Based on SemVer policy [32], [40], [41], we do not expect *floating-minor* to be the most frequently used version constraint type in vulnerable dependencies. So we hypothesize that packages get security fixes automatically when they use *floating-minor*. To verify our hypothesis, we analyzed the vulnerable version ranges (e.g., SemVer versions from vuln introduced to vuln fixed) associated with vulnerabilities in our dataset. We manually reviewed 50 randomly chosen CVEs to validate our hypothesis. We found that, as opposed to our hypothesis, *vulnerability fixes are not being propagated to all major vulnerable versions*.

In our dataset, we found 4,219 (2,465 npm, 1,560 PyPI, and 194 Cargo) CVEs and 1,553 (473 npm, 1,001 PyPI, and 79 Cargo) packages where the vulnerable versions are spread across multiple major versions, but only one major version received the fix. As a case study, we present GHSA-23wx-cgxq-vpwx where package dset's version [0.0.0 - 3.1.1] are vulnerable, and the fixed version for this vulnerability is 3.1.2. Although the package dset has versions {0.0.0, 1.0.0, 1.0.1, 2.0.0, 2.0.1, 2.1.0, 3.0.0, 3.1.0, 3.1.1, 3.1.2, ...}, the fix is not propagated to the major versions 0, 1 and 2. Not propagating the vulnerability fixes to all vulnerable major versions makes *floating-minor* no better than *pinning*. For example, if the dependents of dset use the *floating-minor* constraint, i.e., $\wedge$0.0.0 or $\wedge$1.0.0 or $\wedge$2.0.0, they will always have a vulnerable version of dset in their project since the fix was not backported. Our observation that patches are not being backported aligns with the findings of Decan et al. [42]. Pinckney et al. [32] also found that some developers release security patches with minor and major version increments.

Key Insights: Among outdated and vulnerable dependencies, the most commonly used version constraint type is *floating-minor*, with *pinning* being the next most common.

*B. **RQ2**: How frequently do developers change the version constraint type for dependencies, and how does the change result in dependencies being outdated or vulnerable?*

This RQ explores how changes in version constraints affect whether a dependency becomes outdated or vulnerable. First (**RQ2**a), we want to understand how frequently developers change the *version constraint type*. Then (**RQ2**b), we want to analyze what the trends are in such version constraint type change. For example, we want to know if switching from a strict *pinned* version to a more flexible *floating-minor* constraint is common and, if so, whether this change results in outdated and vulnerable dependencies.

RQ2a. We consider four types of dependency state transitions: (1) updated to outdated; (2) outdated to updated; (3) remediated to vulnerable; and (4) vulnerable to remediated. And each transition may be triggered by one of four change scenarios:

ⓐ The dependency releases a new version. For example, the dependency releases a new version fixing a vulnerability, but the package is using a *pinned* vulnerable version, so it will be considered as remediated to vulnerable transition.

ⓑ The package releases a new version but keeps the *same version constraint and the same constraint type*.

ⓒ The package releases a new version with a *changed version constraint* but retains the *same constraint type* (e.g., remains "pinned" or "floating-minor").

ⓓ The package releases a new version with a *changed version constraint and a changed constraint type* (e.g., switches from "pinned" to "floating-minor").

Table IV summarizes how often each type of change leads to the four transitions. The most frequent cause of outdated to updated transitions is type ⓒ (over 5M cases), where package developers adjust constraints without changing the type. However, 67,681 cases involve an actual type change (type ⓓ), where the package developers change the version constraint and also the version constraint type.

ⓐ is the least common in each of the dependency transitions. ⓐ comprises 246 cases in updated to outdated (0.37%), 5,645 cases in outdated to updated (0.11%), 16 cases in remediated to vulnerable (0.64%), and 51 cases in vulnerable to remediated (0.24%). ⓐ is the case when the dependency version release is auto-adopted by the package.

ⓑ plays a minimal role in regression transitions (to outdated and to vulnerable) but rarely used in upgrades (to updated and to remediated). ⓑ comprises 1,016 cases in updated to outdated (1.52%), 85 cases in outdated to updated (<0.01%), 1,271 cases in remediated to vulnerable (50.42%), and none in vulnerable to remediated (0.00%).

TABLE V
TOP 5 MOST FREQUENT VERSION CONSTRAINT CHANGES BY DEPENDENCY TRANSITION TYPE

Dependency Transition	From Constraint	To Constraint	Constraint Change	Frequency
Updated → Outdated	floating-minor	pinning	↑ constrained	12,853
	floating-major	fixed-ranging		2,739
	floating-major	pinning	↑ constrained	2,243
	floating-minor	floating-patch	↑ constrained	1,630
	fixed-ranging	pinning		660
Outdated → Updated	pinning	floating-minor	↓ constrained	28,591
	floating-minor	pinning	↑ constrained	9,424
	pinning	floating-major	↓ constrained	5,448
	floating-patch	floating-minor	↓ constrained	4,442
	fixed-ranging	floating-major		4,006
Remediated → Vulnerable	floating-minor	pinning	↑ constrained	541
	floating-major	pinning	↑ constrained	199
	floating-major	fixed-ranging		117
	fixed-ranging	pinning	↑ constrained	61
	floating-minor	floating-patch	↑ constrained	43
Vulnerable → Remediated	pinning	floating-minor	↓ constrained	1,154
	pinning	floating-major	↓ constrained	694
	pinning	fixed-ranging	↓ constrained	342
	fixed-ranging	floating-major		331
	floating-minor	pinning	↑ constrained	279

(c) is the most prevalent occurrence across all transitions, particularly in upgrades. (c) comprises 47,795 cases in updated to outdated (71.41%), 5,222,276 cases in outdated to updated (98.52%), 594 cases in remediated to vulnerable (23.55%), and 19,549 cases in vulnerable to remediated (78.66%). This result indicates developers changing the version constraint but keeping the version constraint type is the most prevalent.

(d) reflects deliberate restructuring (both constraint and constraint type). (d) comprises 22,887 cases in updated to outdated (34.10%), 67,681 cases in outdated to updated (1.28%), 1,133 cases in remediated to vulnerable (44.93%), and 3,829 cases in vulnerable to remediated (15.41%).

In total, (a) results in 5,958 (0.11%), (b) in 2,372 (0.04%), (c) in 5,290,214 (98.09%), and (d) in 95,530 (1.77%) dependency state transitions. Our results indicate that auto-adopting newer dependency version releases due to version constraint types is a rare case in changing dependency state.

Key Insights: Updating the version constraint while retaining the same type is the most prevalent case that leads to updated or remediated dependencies. Retaining the version constraint type indicates that developers have a preferred version constraint type for each dependency.

RQ2b. We analyze type (d) transitions to identify the trends in constraint type change. Packages releasing a new version with a changed constraint and a changed constraint type correspond to 1.8% from Table IV. Then, we analyze the frequency of each unique type-change pair. Table V presents the top 5 frequent version constraint type changes in each transition group.

We find that developers often move from *pinning* to more flexible types (e.g., *floating-minor*) when updating dependencies. Changing *pinning* to *floating-minor* occurred in 28,591 outdated to updated transitions and 1,154 vulnerable to remediated transitions, both of which are the highest in that transition group. In contrast, some developers become more restrictive in changing the version constraint. For example, changing *floating-minor* to *pinning* is the highest occurring pattern in transitions toward outdatedness or vulnerability. For vulnerable to remediated transitions, removing *pinning* constraints for *floating-minor*, *floating-major*, or *fixed-ranging* appears to help exposed packages get fixed dependency versions. By analyzing the number of occurrences for each of the cases from Table V, we can conclude that these transitions are not rare.

Under-/over-representation of each dependency type could have an impact on the way dependency constraints are present in Table V. For example, *floating-minor* and *pinning* are frequently present in the transitions since *floating-minor* and *pinning* are the top-2 most frequently used version constraint types. In addition, less frequently used version constraint types (e.g., *or expression*, *not expression*, *complex expression*) are not present in the top-5 frequent transitions in Table V since these are less represented overall in version constraints.

Key Insights: The top 3 version constraint type changes to remediate dependencies involved removing *pinning*.

C. *RQ3: How do dependencies' version constraint type affect the dependencies' time to become vulnerable? How do dependencies' version constraint type affect the dependencies' time to become outdated?*

To interpret the models, we examine the hazard ratios: the hazard of the covariate of interest divided by the hazard of the reference group, which in our case is *pinning*. A hazard ratio $>$ 1 indicates increased hazard compared to the pinning category, and a hazard ratio $<$ 1 indicates decreased hazard compared to the pinning category. A hazard ratio of 1 implies equal hazard between the specific version constraint and *pinning*.

Time to Outdatedness. As shown in Table VI, all covariates have p-values that are < 0.005, except for the *at-most* covariate, demonstrating that the effects of all covariates (minus

TABLE VI
COX TIME-VARYING MODEL RESULTS FOR DEPENDENCY OUTDATEDNESS AND VULNERABILITY

Covariate	Outdatedness				Vulnerability			
	Coef	HR (exp(coef))	se(Coef)	p-value	Coef	HR (exp(coef))	se(Coef)	p-value
floating-major	-4.87	0.01	0.04	<0.005	–	–	–	–
floating-minor	-0.82	0.44	0.01	<0.005	-0.56	0.57	0.05	<0.005
floating-patch	-0.28	0.75	0.01	<0.005	0.39	1.48	0.09	<0.005
fixed-ranging	-0.48	0.62	0.01	<0.005	0.99	2.69	0.06	<0.005
complex-expression	-1.43	0.24	0.04	<0.005	1.26	3.52	0.15	<0.005
at-most	0.00	1.00	0.02	0.96	–	–	–	–
or-expression	0.66	1.94	0.03	<0.005	–	–	–	–
not-expression	-3.28	0.04	0.22	<0.005	–	–	–	–

the at-most covariate) are statistically significant and that the observed results are unlikely due to random chance. The hazard ratio of 1 for *at-most* indicates that dependencies with the *at-most* constraint type have the same hazard (instantaneous risk of becoming outdated at any given time) as *pinning*. Statistically, there is no evidence that the at-most constraint type differs from the pinning constraint type. The *or-expression* type has a hazard ratio greater than 1, indicating that dependencies with the *or-expression* constraint type have about a 94% higher likelihood of becoming outdated compared to *pinning*. The hazard ratios for the rest of the constraint types are less than 1, implying a decreased likelihood of becoming outdated compared to *pinning*. The hazard ratios of these constraint types range from 0.01 (*floating-major*) to 0.75 (*floating-patch*), indicating that dependencies with the floating-major constraint type have a 99% reduction in the likelihood of becoming outdated, while those with the *floating-patch* constraint type have a 25% reduction in likelihood.

The decreasing hazard ratios for *floating-patch* (0.75), *floating-minor* (0.44), and *floating-major* (0.01) can be explained by the relative restrictiveness compared to *pinning* from Fig. 2. *Floating-major* has the lowest hazard ratio, which is expected since *floating-major* is the most flexible version constraint type. Packages using *floating-major* essentially keep the latest available version of that dependency. Similarly, *not-expression* has a very low hazard ratio (0.04) due to the design of *not-expression*. Packages use *not-expression* to avoid a certain version of the dependency (likely because of an incompatibility issue, or breaking changes). In doing so, the dependency resolution algorithm can pick the latest dependency version if the latest version is not the one excluded, and so *not-expression* is essentially a modified form of *floating-major*. However, the hazard ratio for *or-expression* is difficult to reason with, since it is not known why developers would use *or-expression*, and also *or-expression* is npm-specific.

Key Insights: Version constraint type matters in becoming outdated dependencies. *Floating-major* is the least likely and *or-expression* is the most likely to become outdated.

Time to Vulnerable. We removed four covariates from our vulnerability model: *or-expression*, *not-expression*, *at-most*, and *floating-major*. First, we removed *floating-major* and *or-expression* as none of the dependencies with these version constraint types became vulnerable, as shown in Table III. We also removed *not-expression* as only one dependency experienced vulnerability with *not-expression*. These covariates had an event rate of 0% (or close to 0% for the *not-expression*) and would not provide useful information to the hazard estimation function. Next, we removed *at-most*, as the covariate caused convergence issues in our model. The disproportionately high number of events occurring in the early time period (first 100 days) created a quasi-separation issue, which prevented a stable hazard estimation [43].

Floating-minor, *floating-patch*, *fixed-ranging*, and *complex-expression* are the four covariates included in the model. As shown in Table VI, all four covariates have p-values < 0.005, illustrating that the observed results are statistically significant and not due to random chance. *Floating-minor* is the only covariate with a hazard ratio < 1. Dependencies using the *floating-minor* have a 43% reduction in the likelihood of becoming vulnerable. The hazard of vulnerability is 1.5, 2.7, and 3.5 times higher for *floating-patch*, *fixed-ranging*, and *complex-expression*, respectively.

Since *floating-patch* and *floating-minor* are more flexible than *pinning* (Fig 2), we expected both to have hazard ratios < 1. However, *floating-patch*'s 1.48 hazard ratio indicates *floating-patch* is 48% more likely to result in vulnerability. Not backporting vulnerability fixes to all major versions (our finding from **RQ1**) can explain the 1.48 hazard ratio of *floating-patch*. With the same reasoning, *floating-minor* should also have > 1 hazard ratio. In contrast to our expectation, we found that *floating-minor* is 43% less likely to result in vulnerability than *pinning*. From Table II, we know that *floating-minor* is the most used version constraint in npm and Cargo. The 0.57 hazard ratio of *floating-minor* could be explained by the disproportionate use of *floating-minor* in survival dataset. On the other hand, *fixed-ranging* (2.69) and *complex-expression* (3.52) have relatively higher hazard ratios for experiencing vulnerability. Developers using *fixed-ranging* may have found a suitable range that is compatible with their use, and so they may not update dependencies frequently.

Key Insights: Version constraint type matters in becoming vulnerable dependencies. *Floating-minor* is the least expected, and *complex-expression* is the most expected to become vulnerable.

V. Discussion and Implications

A. Implications For Developers

Use Automated Dependency Update. Enabling automatic dependency updates can eradicate the tedious manual task of finding vulnerable dependencies and updating to a fixed version. The use of a package management system (e.g., npm), dependency management tools (e.g., Dependabot), or software composition analysis tools could help developers by notifying them of outdated or vulnerable dependencies. In our survival analysis, we found that *pinning* is more likely to result in outdated dependencies than any *floating*. Thus, we recommend to avoid *pinning*. At the very least *floating-patch* should be used to balance the pros and cons of getting security updates and leaving the scope of breaking changes. *Floating-minor* should be used for upstream packages whose developers are found to be generally following the SemVer policy or for *feature-complete* [31] packages so that bug and security fixes are adopted automatically. *Floating-major* can be used for the packages for which internal breaking changes are less expected. *Fixed-ranging* or *at-most* should be used when package developers are aware of a breaking change in a certain dependency's version and can specify the version constraint to avoid the versions introducing breaking changes.
Use Lockfiles. As an alternative to *pinning*, lockfiles can account for specific versions (or even cryptographic hashes) of a dependency that is tested to work properly without breaking changes. Having a lockfile can reduce the fear of breaking changes with *floating*. If breaking changes occur, developers can iterate over the dependency versions released between the version specified in the lockfile and the highest allowed version to pinpoint the version that introduced the breaking changes to facilitate effective bug and vulnerability triaging. Additionally, lockfiles should be periodically updated to reduce the manual labor needed for the bug or vulnerability triaging phase.

B. Implications for Researchers

In this study, we found that *floating-minor* and *pinning* are the most frequently used constraint types. From our survival analysis, we found that *floating-minor* is better than *pinning*, but *floating-patch* is worse than *pinning* for vulnerability. The implication of these results is that floating vs pinning is a non-trivial decision for developers, and future research could explore this direction further. We also found that updating the version constraint while keeping the version constraint type is the most prevalent change leading to updated/remediated dependencies. Keeping the version constraint type the same could indicate that developers have a preferred constraint type for a given dependency. Future research is needed to reveal if such a preference exists and why. In addition, we do not know the developers' intent in changing the version constraint types, leading to updated/remediated dependencies or vice versa. Developers may not care about outdatedness/vulnerability, and having updated/remediated dependencies could be a byproduct of entirely different decisions or design choices. Future research could help uncover developers' intentions leading to the changes in version constraints.

C. Implications for Tool Builders

Our work highlights that developers may have a preference for version constraint type for a given dependency. Tool builders can help developers who want to add a dependency by showing which version constraint type is mostly used for the given dependency by its dependents. In addition, tool builders can help find bad behaviors in SemVer compliance of packages (e.g., packages that do not propagate vulnerability fixes to all major vulnerable version ranges). A dependency's level of adherence to SemVer can help developers in selecting the appropriate version constraint type for the given dependency.

D. Positioning Our Work In The Context Of Existing Research

He et al. [11] conducted a study that closely aligns with ours, as both examine the *pinning* vs *floating* debate. He et al. [11] conducted a simulation-based analysis, whereas we conducted an empirical analysis and survival modeling. In addition, He et al. [11] used only npm packages, whereas we expanded the scope to include npm, PyPI, and Cargo packages. Despite these differences, both studies complement each other and agree that pinning direct dependencies does not offer significant security advantages. He et al. [11] concluded that pinning direct dependencies does not provide any security benefit in an ecosystem where floating is the mainstream practice. Similarly, our survival analysis reveals that pinning tends to result in more outdated dependencies than floating. Both He et al. [11] and our study recommend avoiding direct pinning and advocating the use of lockfiles to manage dependencies effectively. While the methodologies differ, our study supports He et al. [11]'s conclusions. Given the rise of *pinning* dependencies [11], future research could explore alternate strategies for maintaining up-to-date dependencies for large ecosystems.

VI. Related Work

Dependency Versioning Practices. Dietrich et al. [1] examined 70M dependency version constraints across 17 package managers and, through a developer survey, found that developers struggle to find the right balance between using pinned dependencies and floating dependencies. Raemaekers et al. [7] conducted an empirical study on semantic versioning versus breaking changes and found that adherence to semantic versioning principles has increased only slightly over time. Decan et al. [44] analyzed the SemVer compliance in package dependencies of Cargo, npm, Packagist, and Rubygems. The authors have found that the "wisdom of the crowd" principle is applicable when declaring dependency constraints; if other dependents of the required dependency use SemVer-compliant constraints, it is likely that the dependency is respecting the SemVer policy. Pinckney et al. [32] conducted a large-scale analysis of SemVer practices in npm and found that developers' imperfect use of SemVer version constraints often blocks the propagation of security patches to downstream projects. He et al. [11] conducted a counterfactual analysis and simulation to study the security and maintenance impact of version constraints in the npm ecosystem. They found that

pinning direct dependencies increases the cost of maintaining vulnerable and outdated dependencies and increases the risk of exposure to malicious package updates in larger dependency graphs. While the prior studies in this direction focus on dependency versioning practices in-the-wild, our study focuses on the impact of dependency version constraint types in becoming outdated and vulnerable dependencies.

Outdated Dependencies. Kula et al. [45] analyzed the dependency adoption lag in the Maven ecosystem and found that developers are more likely to adopt the latest version of the dependency for the newly added dependencies than for updating the existing dependencies. In another study, Kula et al. [46] studied the extent to which developers update their library dependencies and found that 81.5% of projects in GitHub have outdated dependencies. Cox et al. [47] introduced the concept of "dependency freshness" to quantify the up-to-date state of software dependencies. They conducted an empirical study with "dependency freshness" in GitHub and found that only 16.7% of dependencies are up-to-date. Derr et al. [48] identified the reasons behind developers not updating their outdated dependencies through a survey and found that developers do not update dependencies due to the fear of breaking changes, lack of knowledge, and lack of motivation. Wang et al. [49] conducted an empirical study of dependency updated intensity and delay in OSS packages and found that 50% of OSS packages have at least half of their dependencies outdated. Huang et al. [50] analyzed usage and updates of Java packages and found that more than half of the projects have a lag of more than 500 days from the latest dependency version. While the prior studies on outdated dependencies analyzed the trend in usage and dependency adoption lag through various metrics, no study analyzed the impact of the version constraint type used in outdated dependencies.

Vulnerable Dependencies. Pashchenko et al. [51] conducted a study to analyze vulnerable open source dependencies in SAP software and found that 81% of the vulnerable dependencies could be fixed by updating to a newer dependency version. In a follow-up study, Pashchenko et al. [52] proposed a methodology, Vuln4Real, to reliably measure the extent of vulnerable dependencies and applied that to the top 500 OSS Maven dependencies from SAP. Kumar et al. [53] studied the impact of vulnerable dependencies in open-source software and found that for most programming languages, a critical vulnerability persists on average for over a year. Mir et al. [54] investigated vulnerable dependencies and their reachability on the Maven ecosystem and found that 32% of all projects are affected by vulnerable dependencies. Zheng et al. [55] conducted an empirical study on vulnerabilities and vulnerable packages in Rust, finding that it takes more than two years for vulnerabilities to appear on public vulnerability databases and that one-third of the vulnerabilities have no fixed commit before their disclosure. They also found that memory safety and concurrency issues account for two-thirds of the vulnerabilities, and the vulnerable code contains statistically significantly more unsafe functions and blocks than the rest of the code. We learned from the prior studies that have empirically evaluated vulnerabilities and vulnerable dependencies, and explored the relationship between version constraint types used by package developers and having vulnerable dependencies.

VII. Threats To Validity

External Validity. A limitation concerns the generalizability of our results to other ecosystems beyond those analyzed. Each ecosystem has its own policies, practices, and developer communities that may influence dependency management in ways not fully captured in this study. Despite this, we believe the insights derived from version constraint usage and their impact on outdated and vulnerable dependencies are broadly relevant to other ecosystems.

Internal Validity. We use vulnerability data from the OSV.dev database [20], a widely adopted source for open-source security advisories. Nonetheless, OSV may not capture every published advisory. Vulnerabilities not present in this dataset could lead to underreporting. Furthermore, we do not assess whether reported vulnerabilities are reachable or exploitable in practice, as discussed in the context of VEX (Vulnerability Exploitability eXchange) reports [56]. In our study, all vulnerabilities are treated equally, regardless of severity or CVSS (Common Vulnerability Scoring System) score.

Additionally, our analysis focuses exclusively on runtime dependencies, which packages have direct control over. We exclude dev and optional dependencies from our measurements. We also excluded SemVer version qualifiers (e.g., prereleases, build metadata) from our dependency resolution and our analysis, since versions with qualifiers are not fetched through typical dependency resolution (e.g, with `npm install`). Finally, while some dependencies may be more critical than others, we treat all dependencies equally. Assigning importance weights is outside the scope of this study, but it represents a meaningful direction for future work.

VIII. Conclusion and Future Work

In this study, we empirically analyzed the version constraint types used in-the-wild and also the patterns of version constraint type changes made by the developers. We utilized survival analysis to find the relative hazard risk of *pinning* to result in outdated and vulnerable dependencies in comparison to the rest of the version constraint types. Our study shows that *floating-major* is the least likely to result in outdated and *floating-minor* is the most likely to result in vulnerable dependencies. We recommend that developers avoid *pinning* and use a hybrid strategy with *floating* and lockfiles. For future researchers, our study could be extended to create a used version constraint type-based metric to measure the risk of a package before developers use that package as a dependency.

IX. Acknowledgment

This work was supported and funded by the National Science Foundation Grant No. 2207008. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] 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)*, May 2019, pp. 349–359, iSSN: 2574-3864. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8816809?casa_token=KYtKkGU9INEAAAAA:inn0e5XiBymxMDQ-m4nXcdomVIXGorYWwbIsXyh2nipOu--OchJ3OYpq7Fon_n-1EiuLm3g03A

[2] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency Smells in JavaScript Projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, Oct. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9519532

[3] A. Zerouali, T. Mens, J. Gonzalez-Barahona, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories — and its application to npm," *Journal of Software: Evolution and Process*, vol. 31, no. 8, p. e2157, 2019, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2157. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2157

[4] 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)*, Nov. 2015, pp. 86–89. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7426643

[5] 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 Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, pp. 42:1–42:56, Jul. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3447245

[6] 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, Jul. 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121216300243

[7] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2014, pp. 215–224. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6975655

[8] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *IEEE Symposium on Security and Privacy (SP)*. arXiv, 2023, arXiv:2204.04008 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2204.04008

[9] N. Zahan, P. Kanakiya, B. Hambleton, S. Shohan, and L. Williams, "OpenSSF Scorecard: On the Path Toward Ecosystem-Wide Automated Security Metrics," *IEEE Security & Privacy*, vol. 21, no. 6, pp. 76–88, Nov. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10163720/

[10] "OSSF Scorecard: Build better security habits, one test at a time," https://scorecard.dev/, last accessed: 31-May-2025.

[11] H. He, B. Vasilescu, and C. Kästner, "Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks," in *Proceedings of the ACM on Software Engineering, Volume 2, Number FSE, Article FSE013 (July 2025)*, Feb. 2025, arXiv:2502.06662 [cs]. [Online]. Available: http://arxiv.org/abs/2502.06662

[12] "Replication Package," https://doi.org/10.5281/zenodo.15559007, last accessed: 31-May-2025.

[13] Sonarcube, "What is an Open Source Package," https://www.sonarsource.com/learn/open-source-package/#:~:text=An%20open%20source%20package%20is,freely%20in%20their%20own%20projects., last accessed: 31-May-2025.

[14] "Open Source Insights: Understand your dependencies," https://deps.dev/, last accessed: 31-May-2025.

[15] Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding vulnerabilities in software supply chains," *Empirical Software Engineering*, vol. 30, no. 1, p. 20, Nov. 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10581-2

[16] J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang, and Y. Liu, "Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639230

[17] Y. Liu, D. Tiwari, C. Bogdan, and B. Baudry, "Detecting and removing bloated dependencies in CommonJS packages," May 2025, arXiv:2405.17939 [cs]. [Online]. Available: http://arxiv.org/abs/2405.17939

[18] M. Alhanahnah and Y. Boshmaf, "DepsRAG: Towards Agentic Reasoning and Planning for Software Dependency Management," Oct. 2024, arXiv:2405.20455 [cs]. [Online]. Available: http://arxiv.org/abs/2405.20455

[19] J. Akhoundali, S. R. Nouri, K. Rietveld, and O. Gadyatskaya, "MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery," in *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2024. New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 42–51. [Online]. Available: https://dl.acm.org/doi/10.1145/3663533.3664036

[20] "OSV.dev : A distributed vulnerability database for open source," https://osv.dev, last accessed: 31-May-2025.

[21] "GitHub Advisory Database," https://github.com/advisories, last accessed: 31-May-2025.

[22] "OSV: Current data sources," https://google.github.io/osv.dev/data/#current-data-sources, last accessed: 31-May-2025.

[23] H. Gu, H. He, and M. Zhou, "Self-Admitted Library Migrations in Java, JavaScript, and Python Packaging Ecosystems: A Comparative Study," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 627–638, iSSN: 2640-7574. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10123560

[24] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, "Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1741–1765, Apr. 2023, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9832512/authors#authors

[25] H. He, Y. Xu, X. Cheng, G. Liang, and M. Zhou, "MigrationAdvisor: Recommending Library Migrations from Large-Scale Open-Source Data," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2021, pp. 9–12, iSSN: 2574-1926. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9402644

[26] M. Saini, R. Verma, A. Singh, and K. K. Chahal, "Investigating diversity and impact of the popularity metrics for ranking software packages," *Journal of Software: Evolution and Process*, vol. 32, no. 9, p. e2265, 2020, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2265. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2265

[27] Y. Sun, D. German, and S. Zacchiroli, "Using the uniqueness of global identifiers to determine the provenance of Python software source code," *Empirical Software Engineering*, vol. 28, no. 5, pp. 1–35, Sep. 2023, company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 5 Publisher: Springer US. [Online]. Available: https://link.springer.com/article/10.1007/s10664-023-10317-8

[28] C. Miller, M. Jahanshahi, A. Mockus, B. Vasilescu, and C. Kastner, "Understanding the Response to Open-Source Dependency Abandonment in the npm Ecosystem," in *International Conference on Software Engineering*, 2025.

[29] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 921–933. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616262

[30] I. Rahman, R. Paramitha, N. Zahan, S. Magill, W. Enck, and L. Williams, "No Vulnerability Data, No Problem: Towards Predicting Mean Time To Remediate In Open Source Software Dependencies," Mar. 2025, arXiv:2403.17382 [cs]. [Online]. Available: http://arxiv.org/abs/2403.17382

[31] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, 2017, pp. 186–196.

[32] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A Large Scale Analysis of Semantic Versioning in NPM," in *Proceedings of the 20th International Conference on Mining Software Repositories*, 2023. [Online]. Available: https://www.jonbell.net/preprint/msr23-npm.pdf

[33] P. Sentas and L. Angelis, "Survival analysis for the duration of software projects," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, 2005, pp. 10 pp.–5.

[34] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information and Software Technology*, vol. 52, no. 9, pp. 902–922, 2010.

[35] D. V. Dawson, D. R. Blanchette, and B. L. Pihlstrom, "Application of biostatistics in dental public health," in *Burt and Eklund's Dentistry, Dental Practice, and the Community*. Elsevier, 2021, pp. 131–153.

[36] V. Bewick, L. Cheek, and J. Ball, "Statistics review 12: survival analysis," *Critical care*, vol. 8, pp. 1–6, 2004.

[37] E. W. Steyerberg and T. A. Gerds, "Concepts in cancer survival analysis: Research questions, data, and models," *Surgical oncology*, vol. 19, no. 2, p. 52, 2010.

[38] C. Davidson-Pilon, "lifelines: survival analysis in python," *Journal of Open Source Software*, vol. 4, no. 40, p. 1317, 2019. [Online]. Available: https://doi.org/10.21105/joss.01317

[39] Z. Zhang, J. Reinikainen, K. A. Adeleke, M. E. Pieterse, and C. G. Groothuis-Oudshoorn, "Time-varying covariates and coefficients in cox regression models," *Annals of translational medicine*, vol. 6, no. 7, p. 121, 2018.

[40] "Semantic Versioning 2.0," https://semver.org/, last accessed: 31-May-2025.

[41] W. Li, F. Wu, C. Fu, and F. Zhou, "A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 1604–1614, iSSN: 2643-1572. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10298458?casa_token=hP2OSK48sbUAAAAA:XNFAsxNuaPffAVeomBVGw1ZQl_mG9mhYJSenVgcusJPnRO5QHk8Cpv5YSHq6nCVHToL6vnHwcA

[42] 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*, vol. 48, no. 10, pp. 4087–4099, Oct. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9540328/

[43] R. W. Nahhas, *Introduction to Regression Methods for Public Health Using R*. CRC Press, 2024.

[44] A. Decan and T. Mens, "What Do Package Dependencies Tell Us About Semantic Versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, Jun. 2021, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8721084?casa_token=snsxTWrs_ToAAAAA:drtQbzFU1icJ0P9II1PX6eLshRSTh94VE3T1Xc6vxFOVPfP3dN3uoY4EC3mmXOpNE76BOmznnA

[45] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2015, pp. 520–524, iSSN: 1534-5351. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7081869

[46] 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, Feb. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[47] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring Dependency Freshness in Software Systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 109–118, iSSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7202955

[48] 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*. Dallas Texas USA: ACM, Oct. 2017, pp. 2187–2200. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134059

[49] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2020, pp. 35–45, iSSN: 2576-3148.

[50] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in Java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, Apr. 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10131-8

[51] 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*. Oulu Finland: ACM, Oct. 2018, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3239235.3268920

[52] ——, "Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, May 2022, conference Name: IEEE Transactions on Software Engineering.

[53] S. H. B. I. Kumar, L. R. Sampaio, A. Martin, A. Brito, and C. Fetzer, "A Comprehensive Study on the Impact of Vulnerable Dependencies on Open-Source Software," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2024, pp. 96–107, iSSN: 2332-6549. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10771210

[54] A. M. Mir, M. Keshani, and S. Proksch, "On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, arXiv:2301.07972 [cs]. [Online]. Available: http://arxiv.org/abs/2301.07972

[55] X. Zheng, Z. Wan, Y. Zhang, R. Chang, and D. Lo, "A Closer Look at the Security Risks in the Rust Ecosystem," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 34:1–34:30, Dec. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3624738

[56] CISA, "Vulnerability Exploitability eXchange (VEX) : Use Cases," https://www.cisa.gov/sites/default/files/2023-01/VEX_Use_Cases_Aprill2022.pdf, last accessed: 31-May-2025.