Mitigating Persistence of Open-Source Vulnerabilities in Maven Ecosystem

Lyuye Zhang[¶*], Chengwei Liu[*§], Sen Chen[†], Zhengzi Xu[*], Lingling Fan[‡], Lida Zhao[*], Yiran Zhang[*], Yang Liu[*]
zh0004ye@e.ntu.edu.sg, chengwei001@e.ntu.edu.sg
[¶]Continental-NTU Corporate Lab, Nanyang Technological University, Singapore
[*]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[†]College of Intelligence and Computing, Tianjin University, China
[‡]College of Cyber Science, Nankai University, China

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00058

***Abstract*—Vulnerabilities from third-party libraries (TPLs) have been unveiled to threaten the Maven ecosystem in the long term. Despite patches being released promptly after vulnerabilities are disclosed, the libraries and applications in the community still use the vulnerable versions, which makes the vulnerabilities persistent in the Maven ecosystem (e.g., the notorious Log4Shell still greatly influences the Maven ecosystem nowadays from 2021). Both academic and industrial researchers have proposed user-oriented standards and solutions to address vulnerabilities, while such solutions fail to tackle the ecosystem-wide persistent vulnerabilities because it requires a collective effort from the community to timely adopt patches without introducing breaking issues.**

To seek an ecosystem-wide solution, we first carried out an empirical study to examine the prevalence of persistent vulnerabilities in the Maven ecosystem. Then, we identified affected libraries for alerts by implementing an algorithm monitoring downstream dependents of vulnerabilities based on an up-to-date dependency graph. Based on them, we further quantitatively revealed that patches blocked by upstream libraries caused the persistence of vulnerabilities. After reviewing the drawbacks of existing countermeasures, to address them, we proposed a solution for range restoration (Ranger) to automatically restore the compatible and secure version ranges of dependencies for downstream dependents. The automatic restoration requires no manual effort from the community, and the code-centric compatibility assurance ensures smooth upgrades to patched versions. Moreover, Ranger along with the ecosystem monitoring can timely alert developers of blocking libraries and suggest flexible version ranges to rapidly unblock patch versions. By evaluation, Ranger could restore 75.64% of ranges which automatically remediated 90.32% of vulnerable downstream projects.

***Index Terms*—Open-source Software, Software Security, Java**

I. Introduction

The vulnerabilities present in widely used TPLs have garnered significant attention from communities. *log4j-core*, serving as a fundamental library, swiftly responded by releasing patch updates after the exploitation of Log4Shell [1]. Downstream users have taken prompt action to adopt these patch updates, as reported by Google [2]. Despite a year's worth of advancements, Log4Shell continues to impact numerous downstream applications and persist within the Maven ecosystem, as reported by many reports and news [3-7]. Given that over 2,000 vulnerabilities from Maven libraries have been disclosed by the National Vulnerability Database (NVD) [8], it is possible that numerous other vulnerabilities persist and pose a threat to the Maven ecosystem.

Aiming for this urgent threat, many researchers [9-15] studied the vulnerability impact within the Maven ecosystem and substantiated vulnerabilities have extensively proliferated in downstream libraries. A few of them have recognized the persistence of vulnerabilities over time and provided insights into potential solutions [9-11], [16]. Wu et al. [9], [17] revealed that the reachable vulnerabilities are more likely to be addressed. Developers' reluctance to upgrade vulnerable dependencies due to potential breaking changes has been highlighted by Pashchenko et al. [11] who also discovered that developers prioritize handling vulnerabilities in direct dependencies rather than transitive ones [18]. Moreover, Industrial standards have been proposed to promote remediation, such that OpenSSF [19] proposed the best practice guidance [20] and a tool, Scorecard [21], for developers on managing vulnerabilities in dependencies. Plumber [16] aims for persistent vulnerabilities in Node Package Manager (NPM) with limited applicability to Maven due to the rare usage of version ranges in Maven. However, because these solutions are either user-oriented aiming for individual projects or inapplicable for Maven, not all stakeholders in the ecosystem would be benefited, which barely promotes the ecosystem-wide mitigation of persistent vulnerabilities due following issues:

Issue 1: The lack of collective awareness. Effectively mitigating persistent vulnerabilities requires collective efforts from the community, particularly from developers of widely-used libraries, rather than just a few individuals. Hence, the ability to accurately locate influential libraries and swiftly arouse awareness of relevant developers is missing yet required.

Issue 2: The overreliance on human practices. Although developers may be aware of the negative consequences of vulnerabilities, they require a solid understanding of software security to effectively remediate them. Even if they possess the necessary skills, remediation practices such as upgrading, backporting, and migration are often time-consuming and

[§] Chengwei Liu is the corresponding author.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00058

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00058

require significant manual effort. As such, relying solely on human practices to eliminate vulnerabilities within the software ecosystem is not realistic.

Issue 3: The backward-incompatibility of dependencies. Maven libraries are known to have version releases violating Semantic Versioning (SemVer) [22-25]. Consequently, numerous breaking changes across upgrades may lurk within the ecosystem. To maintain stability, many developers prefer to define dependencies using single versions [26], rather than flexible version ranges, even though Maven allows for the latter. It further complicates the mitigation of vulnerabilities, as automatic security upgrades are not widely applicable, in contrast to the NPM ecosystem [27].

To address the issues outlined above, we did the followings:

- For **Issue 1**, as illustrated in Figure 1, we first studied the prevalence of persistent vulnerabilities within the Maven ecosystem and identified affected libraries for alerts. Specifically, we implemented an algorithm based on a dependency vulnerability graph we constructed to recursively identify downstream vulnerable dependents. Based on these, the impact of persistent vulnerabilities is uncovered regarding time span and affected libraries[1] in the Maven ecosystem (RQ1). Our study revealed that, upon disclosure, approximately 82.22% of vulnerabilities within the Maven ecosystem remain unresolved in over 50% of the downstream libraries. As of the date of data collection, 58.73% of these vulnerabilities still impacted more than 50% of downstream libraries. Furthermore, it is revealed that persistent vulnerabilities are caused by blocked fixes by downstream libraries, and blocking libraries can be accurately located with our algorithm (RQ2).
- For **Issue 2**, we explored the effectiveness of existing countermeasures in remediating persistent vulnerabilities in the Maven ecosystem. However, we found that these workarounds either required extensive manual effort or were susceptible to breaking changes, highlighting the need for an automatic, scalable and compatibility assurable solution (RQ3).
- For **Issue 3**, we propose a solution for range restoration (Ranger) for both clients and the Maven ecosystem to automatically restore compatible and secure version ranges for vulnerable libraries and dependents. Ranger checks all types of code-centric compatibility with state-of-the-art tools to exclude breaking versions and employs unit tests for validation. With compatible version ranges, patched versions of vulnerable libraries and dependencies could be automatically resolved for downstream users without human intervention. Ranger also continues to mitigate persistent vulnerabilities in the ecosystem by monitoring blocking libraries and providing range suggestions to relevant developers to arouse community awareness. In the evaluation, Ranger could restore $3,109$ (75.64%) ranges which automatically remediated $10,678$ (90.32%) vulnerable downstream projects (RQ4).

[1]Affected libraries refer to the libraries that have the vulnerabilities in their direct or transitive deployable dependencies

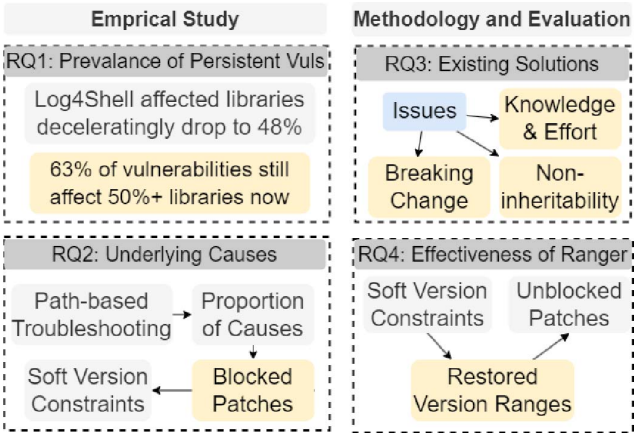

Fig. 1. Overview

The contributions we made are as follows:

- We developed Ranger to restore compatible and secure version ranges which could automatically mitigate the persistent vulnerabilities in the Maven ecosystem.
- We conducted an empirical study to substantiate the persistence of vulnerabilities and quantitatively revealed their underlying cause and the effectiveness of countermeasures.
- We implemented a monitoring system based on an up-to-date dependency graph and a search algorithm to locate the libraries that block vulnerability fixes and suggest remediation for relevant developers and downstream users.

II. Preparation for Empirical Study

To commence our study, we first briefly introduce the concept of SemVer used in Maven. Then, we constructed a dependency graph using data sourced from both the Maven Central Repository (MCR) and NVD. Based on the dependency graph, we developed a searching algorithm (ALSearch) to facilitate tracking of affected libraries throughout the course of our study.

A. Background of SemVer in Maven

Within the Maven ecosystem, most version numbers adhere to the SemVer standard [28]. This standard consists of three digits: *Major*, *Minor*, and *Patch*. Major upgrades, which change the *Major* digit, are the only type of upgrade that allow for incompatible changes. Version ranges [29] supported by Maven rely on SemVer. However, 99.21% of dependency version specifications in Maven are single versions which are called Soft Version Constraints [30] (SoftVer). The SoftVer stipulates the preferred version for a dependency so that Maven mostly resolves the preferred versions for the dependencies [26].

B. Infrastructure of Study

1) Dependency Graph for Maven

A dependency graph was constructed, including vulnerabilities, as an infrastructure for the empirical analysis. As of 01

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

Apr 2023, MCR contained $541,753$ libraries and $11,859,883$ versions, both of which were extracted from the MCR index [31] and added to the dependency graph as *Library* and *Version* vertices. $82,708,563$ dependency edges from *Version* to *Version* were extracted from the Project Object Model (POM) files including properties specifically designed to regulate the dependency resolution. We used the approximately over 2k Common Vulnerabilities and Exposures (CVE) for Maven libraries at NVD as vulnerability data. Due to the absence of well-formatted mappings between vulnerabilities and versions, $1,861$ vulnerabilities and their mappings were collected after cross-checking multiple sources from Github Advisory [32], Google Open-Source Database [33], and Snyk Vulnerability Database [34], which are available on our website [35].

2) Search Algorithm

We developed a precise Affected Library Searching Algorithm (ALSearch) that leverages the Maven dependency resolution rules to accurately track dependents of vulnerabilities based on the dependency graph. Unlike the forward resolution approach used by Maven to resolve dependencies from the root to leaf vertices, ALSearch was designed to facilitate backward tracking from vulnerability vertices to dependent vertices. As ALSearch is tailored for backward tracking, its rules have been adapted accordingly, and are outlined below:

Scope is a feature to limit the transitivity of a dependency. Out of the six scopes, only *compile* and *runtime* are inheritable and tracked by ALSearch. **Optional** dependencies are not transitive, and thus should not be tracked for dependents with $\geq$ 2 depth. **Exclusions** are used to exclude certain versions of transitive dependencies. All transitive dependencies under the *exclusions* are excluded. Hence, if the libraries with vulnerabilities are excluded by any dependent, dependents should not be tracked. **Multiple versions selection**: If a library is used with different versions in a dependency tree, Maven would prioritize the version specified first during a Breadth-First Search (BFS) resolution from direct to transitive dependencies. ALSearch considers a target library affected only if the vulnerable versions of the affected library are closer to the target library than the non-vulnerable versions.

Incorporating the above rules, for each vulnerability, ALSearch iterates over downstream libraries in a BFS manner. During each iteration, it includes two procedures to track a dependent and validate the tracked target respectively:

- **Dependents tracking**: Check if the *Version* vertex has consecutive dependency edges pointing to any vulnerable version of a library affected by a *Vulnerability* vertex. If yes, check if the properties on dependency edges adhere to the aforementioned rules. If yes, proceed to the next procedure.
- **Dependencies validation**: Resolve dependencies of the target *Version* vertex following normal Maven dependency resolution rules in a reversed direction until the version of the vulnerable library is resolved. If the resolved version is vulnerable, the target *Version* vertex is considered an affected version.

After the iteration, the affected *Version* vertex is stored with the publishing date and depth. To boost performance, the maximum depth of the call chain is initially set to 10, based on research indicating that the semantics decline after 10 successive calls [36]. Our study later also confirms that there are significantly fewer affected libraries beyond a depth of 9. We verified ALSearch by randomly selecting $1,000$ affected library versions and retrieving dependency trees of them using the *mvn deptree* command. If the library did depend on a vulnerable dependency, the library was considered affected. Only 12 (1.20%) libraries were false positives, mainly due to different OS requirements or incomplete data in MCR (discussed in Threat of Validity Section VI).

III. Empirical Study

To quantitatively assess the prevalence and underlying cause of persistent vulnerabilities in the Maven ecosystem, we conducted an empirical study to answer the following research questions:

• **RQ1**: ***How prevalent are persistent vulnerabilities in the Maven ecosystem?*** The impact of vulnerabilities over time is evaluated regarding the distribution of time spans and counts of affected libraries to demonstrate persistent vulnerabilities.

• **RQ2**: ***What are the causes of persistent vulnerabilities?*** We quantitatively uncovered the underlying factors by categorizing and analyzing 6 cases to identify the primary cause.

Dataset: the primary dataset is the dependency graph in Section II-B1. To investigate the prevalence of Log4Shell in real-world projects, besides data from MCR, an additional dataset was created by cloning Java repositories on GitHub managed by Maven (with POM files) and filtering out those with fewer than 20 stars to ensure their popularity. As of April 1, 2023, a total of 13,638 repositories were collected, and dependency trees were extracted using the Maven command *mvn deptree*. The dependency trees of 9,220 repositories were successfully extracted.

A. RQ1: Analysis of persistent Vulnerabilities

The impact of persistent vulnerabilities on downstream affected libraries is demonstrated by their long-tail prevalence. We used Log4Shell as an example to showcase the metrics we used and then evaluated all vulnerabilities to demonstrate the persistence.

1) Log4Shell Analysis

First, we retrieved the affected library and version vertices associated with release dates with ALSearch. Because usually, the latest version of a library is the release currently maintained by the developers, a library is considered affected if the latest version depends on vulnerable *log4j-core*. The downstream libraries were categorized into 3 categories (1) **Affected**: The downstream library's latest version is affected. The proportion of these libraries is denoted as P_{vul}. (2) **Patched**: The library's latest version is not affected, but at least one of its previous versions was affected. The proportion of them is called P_{patch}. (3) **Removed**: The older versions of the library were affected, and the latest version does not depend on *log4j-core* anymore.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

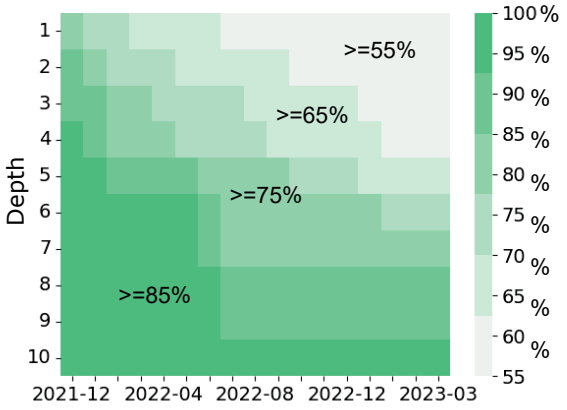

Fig. 2. Heatmap of Proportion of Affected Libraries by Log4Shell

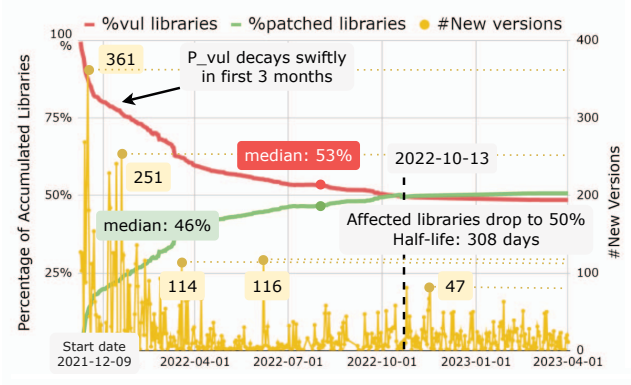

Fig. 3. Accumulated Affected and Patched Libraries for Log4Shell

The P_{vul} of Log4Shell over time is demonstrated in the heat map Figure 2. In the heatmap, the x-axis refers to the timeline from the publishing date at NVD to 01 Apr 2023 based on months, while the y-axis refers to the depth. It is shown that P_{vul} at depth 1 decays faster than at other depths. It is because those libraries serve as first-level dependents, which would be quickly aware of the vulnerable versions of *log4j-core* in their dependencies. With the depth increasing, the downstream libraries are less likely to be aware of the transitive vulnerability and less likely to execute the vulnerable code of *log4j-core*. Thus, the decaying rate decreases as the depth goes deep.

In Figure 3, P_{vul} and P_{patch} are depicted by days. The sum of P_{vul} and P_{patch} is nearly 100% because the number of the third category, **removed**, is negligible. The P_{vul} reached 50% in Oct 2022 and decayed much slowlier than before. Since P_{vul} decays in a decelerating manner, Log4Shell would remain persistent in the ecosystem without abating for a long time. Hence, we define a metric, **Half-life**, to measure the time that P_{vul} decays to 50% from its initial value. The Half-life of Log4Shell can be measured based on days as 308 days.

Although the P_{vul} decays slowly, the number of newly released affected versions decreases more quickly than P_{vul} as in the same figure at the right axis. The number of new versions gradually decreases from the peak of 361 when Log4Shell was initially exposed. It is seen that there were still new affected versions published after 15 months of exposure. We further investigated the depths of these versions and found out that 94% of them were not first-level dependents. It suggests that the upstream dependencies of these affected libraries failed to upgrade *log4j-core* in time. Note that the number of new vulnerable versions has been fluctuating because the numbers are usually small on weekends.

To assess the prevalence of Log4Shell in real-world Java projects, we searched for vulnerable versions of *log4j-core* in the dependency trees of the $9,220$ Maven projects we collected earlier. Our search revealed that 973 (10.55%) of these repositories had used *log4j-core* in their dependency trees, out of which 392 (40.28%) were using the vulnerable versions of *log4j-core*. We confirmed that none of these repositories had published vulnerable versions to MCR, which indicates that, besides libraries, end users were still using vulnerable *log4j-core* versions in their projects after 15 months of disclosure.

Finding 1: The P_{vul} of *log4j-core* decayed rapidly to 65% in the first 3 months upon disclosure. However, the decaying was decelerating, and it took 308 days to reach its *Half-life*. Log4Shell was still affecting 392 GitHub Maven projects after 15 months.

2) Other Java Vulnerability Analysis

To find out if the decelerating decaying of P_{vul} is prevalent for other vulnerabilities, we measured the P_{vul} for all collected Java vulnerabilities as illustrated in Figure 4. Based on P_{vul}, the *Half-lives* of vulnerabilities were derived. Since the exposure duration (from publishing date to data collection date 01 Apr 2023) varies greatly among vulnerabilities, we normalized *Half-life* by dividing the exposure duration. While the **New Release Span** (NRS) was calculated by days from the CVE publishing date to the last date that affected versions are released. *New Release Span* was also normalized by the same exposure duration per vulnerability. Because the number of affected libraries and versions vary greatly among vulnerabilities, the vulnerabilities with exceptionally few affected versions could bring deviations to the distributions. To ensure the representativeness of the vulnerability data set, we filtered out vulnerabilities that affected fewer than 100 versions and plotted the same normalized distributions in Figure 4 as *Filtered*. The number of filtered vulnerabilities was $1,319$.

The normalized Half-lives can be negative if the P_{vul} already drops below 50% before the vulnerability is published. Also, the normalized Half-lives can be 100% if the P_{vul} is still above 50% by the data collection date. According to Figure 4, only 17.78% of vulnerabilities have their P_{vul} dropped below 50% before the publishing of vulnerabilities, which means

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

the rest 82.22% of vulnerabilities affect 50%+ downstream libraries when they were disclosed. Even by the data collection date, 58.73% of vulnerabilities still maintain over 50% P_{vul}. Hence, it is concluded that most vulnerabilities persist and continue to affect downstream libraries, as seen in the case of Log4Shell. In Figure 4, *Filtered half-life*, denoted by light green bars, exhibits the distribution of filtered vulnerabilities. Because the numbers of filtered and pre-filtered vulnerabilities in most intervals are close to each other, it means vulnerabilities that were filtered out did not cause deviations. It is noteworthy that both ends of the distribution are higher than those in between, which means most vulnerabilities either were quickly remediated by downstream libraries or persisted in the ecosystem.

The normalized *New Release Span* is used to indicate the impact of vulnerabilities in new releases. In Figure 4, normalized *New Release Span* is depicted by yellow lines. 39% of vulnerabilities still have new affected versions released in the month of data collection (normalized *New Release Span* is 100%), which indicates that nowadays there are still a non-trivial number of downstream developers who fail to upgrade their vulnerable dependencies. Although the distribution of *New Release Span* is dissimilar to the half-life, they both have valley-like shapes, which proves the polarization in vulnerability remediation of the Maven ecosystem. We further measured **Full-life** (the number of days that P_{vul} drops to zero) instead of *Half-life*, and it turned out that only 196 (9.08%) of vulnerabilities have finite *Full-lives*, which means only 9% have all downstream libraries' latest versions fixed regardless of how long time it took.

Finding 2: 82.22% of vulnerabilities affected 50%+ downstream libraries when they were disclosed. The *vul rates* of vulnerabilities have been decaying in a decelerating manner over time, but till our data collection date, 58.73% of them still maintained 50%+ *vul rates*. There are 39% of vulnerabilities that still affect the new versions of downstream libraries that were released in the month of data collection. Only 9% of vulnerabilities terminated their persistence.

B. RQ2: Study of Underlying Causes

We aim to uncover underlying causes in this RQ. Inspired by the fact that 94% of new versions are transitively affected by other vulnerable downstream libraries, we attempted to investigate the causes based on vulnerability propagation paths.

1) Distribution of the causes

We used a general model that included roles from the source of vulnerability to end users in the vulnerability propagation path as depicted on the left in Figure 5. The roles are ***Vulnerable libraries***, ***Medium dependents***, and ***End users***. From RQ1, it is known that the misbehavior of these roles may block the patches from downstream libraries, which leads to persistent vulnerabilities. Hence, we further investigated what kind of misbehavior blocked the patches. To clearly clarify

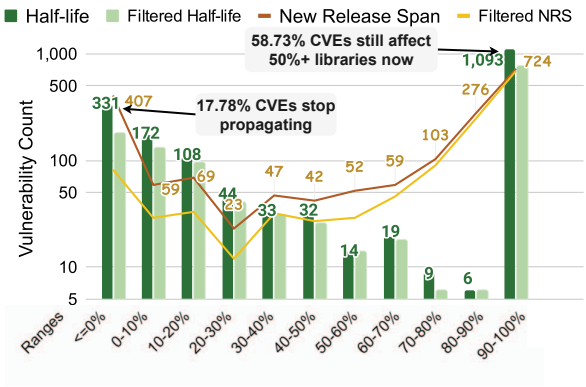

Fig. 4. Distributions of Normalized Half-lives and New Release Span

causes without overlapping, the blockage of patches ascribes to the first role that conducts misbehavior during a bottom-up investigation because downstream roles automatically inherit configurations from the upstream.

Based on the sequence of release time of two parties on each dependency relationship, we could summarize 6 causes out of three types of dependency relationships among 4 roles as illustrated in Figure 5. In the figure, the rectangular box with dashed lines refers to the absent version vertex that is supposed to be present. And the boxes filled with purple color refer to the roles that are to blame for the patch blockage. For example, in the first column, the **Cause 1** is presented: The downstream dependents are affected by the vulnerability because the vulnerable library fails to release the patched version in time. Note that the ***First Depts*** are split apart from ***Medium Depts*** as an independent role because the first-level dependents directly determine the versions of the vulnerable libraries for the other medium dependents and can explicitly select the strategy between version ranges and SoftVers.

In Figure 5, **Cause 2** refers to the *First Depts* still using the vulnerable versions even if the patched versions are available. The **Cause 3** refers to the *First Depts* failing to release new versions that depend on the patched versions so that the downstream dependents are forced to use the non-patched versions of *First Depts*. Similarly, **Cause 4** and **Cause 5** refer to the corresponding misbehavior of *Medium Depts*. The last **Cause 6** stands because the versions explicitly overridden by *End users* are still vulnerable versions.

Next, with the causes summarised, the importance of each cause is embodied by its proportion of occurrences. To avoid duplicated counts, we only counted the number of paths where blockage of patches occurred. Figure 6 illustrates the proportions of each cause over all valid paths. Firstly, the *Cause 1* is ruled out, because it is caused by the absence of patches instead of the blocked patches. Secondly, it is seen that the *Medium Depts* account for most of the paths, which is 36.72%, and *First Depts* account for a very close number of paths, which is 35.24%. Considering that the misbehavior of any role along the path could lead to the blocked patches, it

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

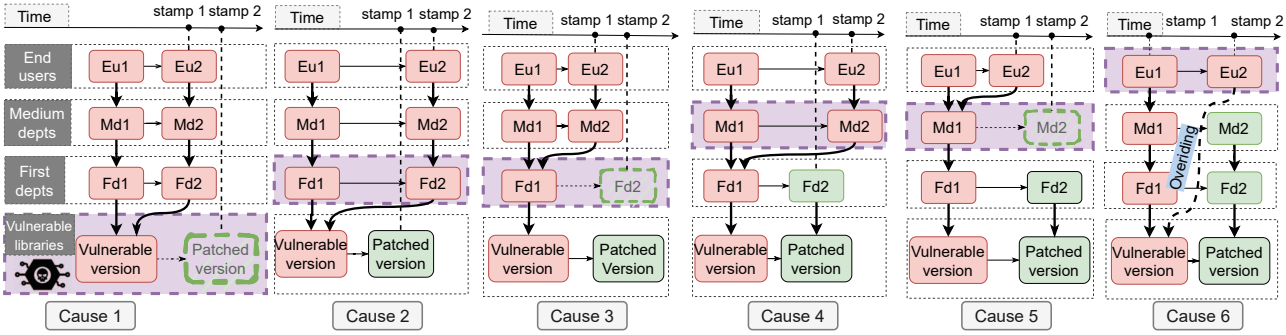

Fig. 5. Scenarios of Different Causes

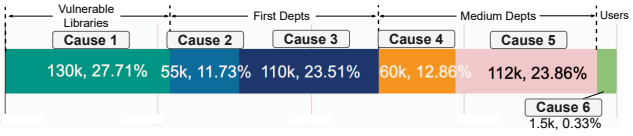

Fig. 6. Proportions of Each Cause

is remarkable that *First Depts* could affect the similar amount ($165k$ and $172k$) of paths as the rest all *Medium Depts* at 2-15 depths, which proves that *First Depts* is the most critical role regarding facilitating the patch adoption than other roles. Finally, although the proportion of *Cause 6* is small, it proves that *End users'* decisions are not always reliable. In fact, much domain knowledge and manual efforts are required for *End users* to select the best version against all vulnerabilities.

Finding 3: It is concluded that misbehavior by *First Depts* (35.24%) and *Medium Depts* (36.72%) are guilty of the majority of affected paths. The *First Depts* are the most significant role in terms of unblocking patches.

In reality, most developers are only concerned by the vulnerabilities in their direct dependencies according to a study [18]. If a *First Dept* uses the vulnerable version, the downstream libraries would automatically inherit the vulnerable version, which means that developers of *First Depts* should be aware of the vulnerability and promptly upgrade vulnerable direct dependencies to patched versions instead of relying on downstream developers. Unfortunately, due to widely used SoftVers (99%) in Maven, the versions specified for vulnerable libraries offer limited flexibility to upgrade against vulnerabilities. It would be unrealistic to force developers in Maven to swerve to version ranges abruptly because SemVer is not properly complied with in Maven and the backward compatibility has to be manually assured. Thus, to avoid reliance on developers, an automatic and scalable way to introduce flexibility to dependency versioning is required to mitigate persistent vulnerabilities.

Finding 4: The root cause of the misbehavior is the widely used SoftVers which greatly limit the flexibility of dependency version selection. Without flexible version ranges, the downstream libraries and applications are automatically prone to vulnerable dependencies even if patched versions are released.

IV. Methodology and Evaluation

Because the limited flexibility hinders the spread of patches, we aim to introduce the flexibility to unblock the patches. First, we reviewed the existing solutions to identify the pros and cons, based on which, our solution Ranger is proposed and evaluated to answer the following research questions:

- **RQ3**: ***How do existing solutions address persistent vulnerabilities?***
- **RQ4**: ***How effective is Ranger regarding mitigating the persistence of vulnerabilities?***

A. RQ3: Review of Existing Solutions

The solution recommended by Maven is the semantic version ranges [29]. Unfortunately, considering that SemVer is not properly complied with, instability could be introduced by ranges so that ranges are rarely adopted. Apart from ranges, the most used approach is the transitive version override. If the versions of transitive dependencies are vulnerable, any dependent can override the transitive vulnerable versions by *dependencyManagement*. Hence, as solutions supported by Maven, ranges and version overriding can be used to mitigate the persistence of vulnerabilities. Note that besides Maven, other popular Java Package Managers, Gradle [37] and Ivy [38] implement similar overriding mechanisms, *Dependency Constraints* and *Dependency Overriding* respectively to determine the versions of transitive dependencies if the transitive dependencies exist. Thus, we refer to this overriding mechanism as *dependency version Overriding*. There are also other workarounds, such as tampering with local libraries of vulnerable dependencies to manually backport patches for deployment environments. But temporary workarounds are too infrequently used to be discussed. Furthermore, *exclusion* supported by Maven is not discussed either, because it is used

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

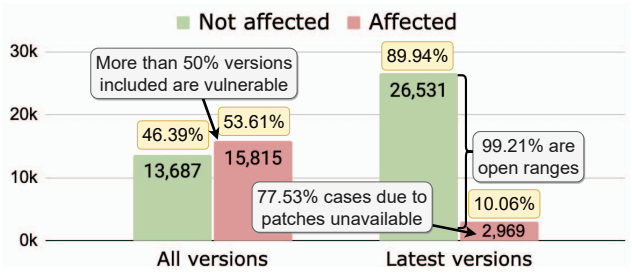

Fig. 7. Usage of Vulnerability Related Version Ranges

to exclude unused transitive dependencies which are not worth mitigating the vulnerabilities for.

1) Study of the Usage of Ranges

From the $82m$ collected dependency relationships, only $637,783$ (1.02%) of them use the semantic version ranges. Out of the range-use dependencies relationships, $29,556$ (4.63%) are specified for the vulnerable libraries by *First Depts*. We further investigated these vulnerability-related ranges to reveal how many vulnerabilities can be automatically bypassed.

As illustrated by the left two bars in Figure 7, considering all versions within the ranges, 53.61% of versions are vulnerable. However, Maven would usually select the latest (semantically highest) version in a version range as the resolved version of the dependency. Thus, if only the latest versions in these ranges are considered, the proportion of vulnerable latest versions drops to 10.06% in the right 2 bars in Figure 7, which proves that version ranges can effectively free dependents from vulnerabilities if patched versions are included. To understand how patched versions were introduced, we went through the ranges whose latest versions are not vulnerable. Out of the $26,531$ non-vulnerable versions, 99.21% of them belong to ranges that are actually right open ranges, such as *[1.1,)* without defining the upper bounds. Because the right open ranges would always be resolved to the latest version, the potential breaking changes could be introduced to the dependents whenever any incompatible new versions are released.

Finding 5: Although the version ranges allow flexible upgrades of vulnerable dependencies, they are rarely used in Maven (1.02%). The fact that the latest versions of 89.94% of version ranges of vulnerable libraries were no longer vulnerable proves the effectiveness of version ranges. However, 99.21% of ranges that successfully bypassed vulnerabilities were open ranges that are subject to unpredictable incompatibility issues. Hence, to properly use version ranges, compatibility has to be assured.

2) Study of the Version Overidding

Because only *Medium Depts* and *End users*, the indirect dependents of vulnerable libraries, would use *dependency version Overriding* to control the versions of transitive dependencies, we study the effectiveness of *dependency version Overriding* for them. In total, there are $639,710$ (6.50%) POM files for versions that use *dependency-Management*, from which we extracted the overridden versions per POM file. After matching the overridden versions with vulnerability mappings, we found $295,951$ POM files have overridden versions of vulnerable libraries. Then, these files were categorized into 2 cases in Table I regarding whether they bypassed the vulnerabilities: (1) *Affected*: Any overridden version in the POM file was still vulnerable. (2) *Bypass*: The default version was vulnerable but the overridden was not. Note that there was overlapping because a POM file may have multiple overridden versions.

TABLE I
COUNTS OF POMS WITH *dependencyManagement*

With vul libraries	Affected by CVEs	Bypass CVEs	Overlapping
295,951	254,043 (86%)	256,841 (87%)	214,933 (72%)

It turned out 86% of POMs bypassed the vulnerabilities in transitive dependencies because probably their developers were aware of the vulnerabilities and explicitly addressed them with *dependencyManagement*. However, it is surprising that 72% ($214,933$) POM files both bypassed some CVEs and introduced other CVEs at the same time. Only 14% of POMs completely bypassed all vulnerabilities. It implies that fixing vulnerabilities with version overriding is a non-trivial job, which is the first weakness of version overriding, **Knowledge and efforts**. The developers must equip with extensive domain knowledge of vulnerabilities and invest efforts to ensure their eradication. Although version overriding is able to address vulnerabilities for the current project within a POM file, it is not inheritable according to Maven Specification [39] so that it does not benefit the downstream libraries. Another weakness of version overriding is **Non-inheritability**, because of which, Since the version overriding only works for current projects instead of dependents that depend on the projects, the vulnerable versions are still being used by downstream libraries unless all developers along the propagation path conduct the same overriding. Therefore, the patch versions cannot be automatically adopted by downstream users. In conclusion, the version overriding can only serve as a temporary workaround instead of boosting the self-healing of the ecosystem.

Finding 6: The adoption rate of dependency version overriding is 6.50% and only 14% of adopters completely bypassed all vulnerabilities. Because dependency version overriding requires knowledge and manual effort and is unable to benefit downstream users due to non-inheritability, it is not effective in eliminating persistent vulnerabilities.

Another solution worth discussing is Plumber [16] which addresses the persistent vulnerabilities in the NPM ecosystem. Plumber employs a dependency graph to identify the dependents that block fixes of vulnerabilities. Subsequently, it endeavors to upgrade the blocking dependents to compatible versions. If upgrading is not possible within the bounds

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

of compatibility, Plumber generates remediation suggestions, such as backporting and migration, both of which require manual intervention. However, Plumber is not applicable to Maven, because it relies on compatible ranges that are pre-specified by developers, a feature that is prevalent in NPM [27] but not in Maven, which further necessitates the compatible version ranges for Maven.

B. RQ4: Methodology and Evaluation of Ranger

1) Requirements of the Solution

Based on the previous research question, existing solutions, such that open version ranges are subject to breaking changes and dependency version overriding is non-inheritable and requires intensive manual efforts. Despite the limited usage, version ranges were proven to be effective for unblocking the patches. However, due to legacy reasons, developers predominantly utilize SoftVers, making it impractical to mandate a shift toward version ranges, not to mention that version ranges have to be manually curated by developers. Therefore, our objective is to propose an automated solution for restoring version ranges of both vulnerable libraries and dependencies that transitively depend on vulnerable libraries from SoftVers. By restoring the version ranges, vulnerability fixes within the ranges can propagate smoothly and automatically to downstream users. Moreover, for the purpose of ecosystem-level implementation, Ranger should possess the ability to continuously monitor the Maven ecosystem for blocking dependents and promptly provide the restored version ranges along with corresponding suggestions to the developers of such blocking dependents. This approach would expedite the propagation of patches throughout the ecosystem.

2) Design of Ranger

To this end, we have proposed Ranger, which comprises a server-side edition and a client plug-in. The client plug-in for Ranger can be integrated into a developer's workflow as a Maven plug-in. For a Maven project, this plug-in can automatically replace the SoftVers in the POM file with curated compatible version ranges, and the developer can effortlessly publish the updated POM file with version ranges to benefit downstream users. On the other hand, the server-side edition of Ranger employs the ALSearch algorithm to continuously monitor an up-to-date dependency graph for instances of vulnerability fix blockage caused by SoftVers. When a blockage is detected, Ranger calculates compatible version ranges for the vulnerable constraints and reports this suggestion of version ranges to the relevant developers.

As depicted in Figure 8, we first introduce the plug-in that accepts a dependency with SoftVer and class files of the project as input. Given that version ranges specified by developers typically consider compatibility and functionality, Ranger aims to ensure them for the restored version ranges. Specifically, given a SoftVer v_s, Ranger retrieves sorted candidate versions $V_{cand} = \{v_1, v_2, ..., v_n\}$ from the MCR as a list as well as the version and vulnerability mappings from the dependency graph. Then Ranger determines which versions from V_{cand} should be included in the restored range V_r to ensure V_r is more secure, flexible, and compatible. We further formulate the problem into a Multi-Objective Optimization problem:

Algorithm 1: Algorithm of Ranger

Input: SoftVer v_s, candidate versions V_{cand}, class files f
Output: Restored version range V_r

1 $V' \leftarrow set(v_s)$
2 $dt_{v_s} \leftarrow dependencyTree(v_s)$
3 $vul_{v_s} \leftarrow queryCVE(dt_{v_s})$
4 **foreach** v_{cand} *in* V_{cand} **do**
5 $dt \leftarrow dependencyTree(v_{cand})$
6 $vul \leftarrow queryCVE(dt)$
7 **if** $\&\ \ vul \leq vul_{v_s}$ **then**
8 $V' \leftarrow v_{cand}$
9 $sort(V')$
10 $V_{upper} \leftarrow V'.truncate(v_s, v_{last})$
11 $V_{lower} \leftarrow V'.truncate(v_{first}, v_{v_s})$
12 **foreach** v *in* V_{upper} **do**
13 $api = compatibilityCheck(v_s, v)$
14 **if** $\neg reachable(f, api)$ **then**
15 $V_r \leftarrow v$
16 **foreach** v *in* $reverse(V_{lower})$ **do**
17 $api = compatibilityCheck(v_s, v)$
18 **if** $\neg reachable(f, api)$ **then**
19 $V_r \leftarrow v$
20 **foreach** v_r *in* V_r **do**
21 **if** $queryCVE(dt(v_r)) > min(Vul(V_s))$ **then**
22 V_r **remove** v_r
23 **foreach** v_r *in* V_r **do**
24 **if** $\neg unitTest(v_r)$ **then**
25 V_r **remove** v_r
26 **return** V_r

- **Objective 1** (Primary): The maximum number of vulnerabilities for all versions in V_r is minimized to guarantee any version resolved by Maven is more secure than v_s.
- **Objective 2** (Secondary): V_r should include as many candidate v as possible for better flexibility.
- **Constraint 1**: V_r must be compatible with v_s.
- **Constraint 2**: any v_r in V_r must has not greater vulnerabilities than v_s to ensure the effectiveness of restoration.

$$
\begin{aligned}
min \quad & f_1 = max(\sum_{n=1}^{dt(v_r)} count_{vul}(n)) \\
max \quad & f_2 = |V_r| \\
s.t. \quad & c_1 : compatibility(v_s, v_1) = 1 | \forall v_r \in V_r \\
& c_2 : \sum_{n=1}^{dt(v_r)} count_{vul}(v_r) \leq \sum_{n=1}^{dt(v_s)} count_{vul}(v_s) | \forall v_r \in V_r
\end{aligned}
$$

where $dt(v) = \{n_1, n_2, ..., n_t\}$ is to resolve a dependency tree from the version v. the total vulnerabilities of v are the sum of numbers of vulnerabilities associated with each node in $dt(v)$ to include transitive vulnerabilities.

We implemented Algorithm 1 in Ranger to solve the problem above. From L1-L8, Ranger first queries the number of

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

vulnerabilities for the resolved dependency tree of SoftVer v_s and each version in V_{cand}. To adhere to constraint c_2, the versions with more vulnerabilities than v_s are filtered out. Then from L9-L11, Ranger sort the filtered versions in a SemVer order and split the list of versions into the upper and lower parts by v_s to add potential versions bi-directionally for more candidates. For both the upper and lower parts, Ranger checks the compatibility between candidates and v_s. Note that the compatibility checkers employed by Ranger can handle all types of code-based compatibility as specified in the Oracle documentation [40], including Source, Binary, and Behavioral Compatibility. The Source and Binary Compatibility are ensured by two commonly used tools, revapi and jcp [41], [42] with high accuracy. For Behavioral Compatibility, we used the only static detector Sembid [43]. Specifically, Ranger calculates incompatible APIs with the checkers and compares them with the reachable APIs collected from the call graphs. The call graphs are constructed with Soot Spark [44] from the class files of the project and byte code of the dependency to determine whether any incompatible API is reachable. If a candidate version has no reachable incompatible APIs, it is included in the range. In L13, compatibility checkers serve as a pre-filter for the final validation because they are static and more efficient than testing. In L20-L22, Ranger excludes versions that have more vulnerabilities than the minimum required in order to satisfy Objective f_1. In L23-25, given the heavier resource demands of unit tests, Ranger further excludes versions failing the unit test serving as the final validation.

Regarding the server-side edition of Ranger, the initial step involves identifying the blocking dependents, denoted as *First Depts*, by means of the ALSearch algorithm. The plug-in running on the server proceeds to calculate and test the compatible version ranges using the repositories stored in our database. If a version range covering the patched version is successfully restored, Ranger generates a report that is sent to the relevant developer. In cases where range restoration fails, Ranger attempts to locate the *Second Dept* of the failed *First Dept* from the dependency graph and calculates the restorable range towards the *First Dept* instead of the vulnerable library. This is because *First Dept* is the direct dependency of *Second Dept* and only specified versions of direct dependencies are transitive for the rest of the dependents. This process is repeated 10 times until no range can be restored.

3) Evaluation of Ranger

To showcase the effectiveness of Ranger in real-world scenarios, we initially evaluated the plug-in on a dataset of 252 GitHub repositories that included vulnerable versions of *log4j-core* in their dependency trees, as of 01 Apr 2023. Subsequently, we conducted a large-scale evaluation on another dataset to demonstrate the effectiveness of Ranger for mitigating persistent vulnerabilities in the Maven ecosystem.

• **Evaluation of Plug-in: Dataset:** From the $9,220$ repositories in Section III, we retrieved the dependency by Maven command and check if any vulnerable log4j version was still in use. 374 repositories were derived. Only 252 of them could be successfully compiled and tested. **Results:** We ran Ranger to only restore the version ranges for *log4j-core* in 252 parent POM files to control the variables. 160 secure ranges of them were restored with a 63.49% restoration rate. Before running the compilation and unit tests, 179 raw ranges were statically calculated, which means unit tests could reduce 19 false positives. It should be noted that false negative cases were present in our evaluation, as false alerts produced by static compatibility checkers exclude the potential candidate versions, and unit tests only narrow the range but not widen it. Thus we manually checked the failed cases and found that 11 could have been restored. The results were summarized in Table II with the actually restorable repositories accounting for 67.85%. Although Ranger had some false negatives, it hardly introduced false positives that could break current and downstream projects. It was proven that 93.57% of restorable version ranges could be automatically restored by Ranger.

• **Evaluation of Server-side Edition: Dataset:** To simulate a scenario where developers of downstream dependents adopt the version ranges generated by Ranger upon the disclosure of a vulnerability, we compiled a list of all affected libraries

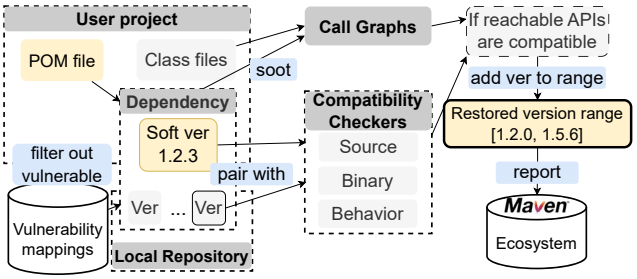

Fig. 8. Overview of Ranger

TABLE II
RESULTS OF RANGER

	Restored	Failed Unit Tests	Restore Rate	Recall/Precision
*GT	171	N.A.	67.85%	N.A.
Ranger	160	19	63.49%	93.57%/100.00%

1) GT stands for ground truth.

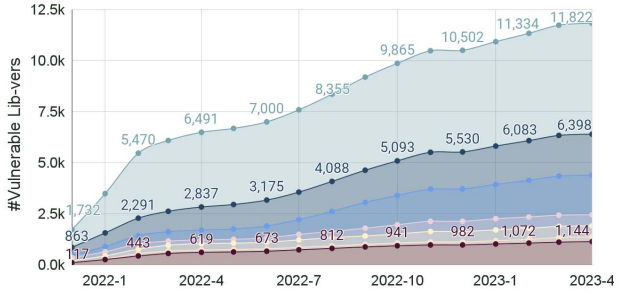

Fig. 9. Number of Vulnerable Lib-vers over Months after Applying Ranger to Dependents at 1-10 Depths

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

and versions published after the disclosure of Log4Shell. This resulted in a total of 11,822 library-version pairs, denoted as lib-vers. **Results:** Regarding the lib-vers, Ranger first restored ranges for the *First Depts* at Depth 1. This resulted in a successful restoration of 486 out of 668 dependent lib-vers. The generated ranges were then applied to the dependency graph, and we performed ALSearch again to retrieve the affected *Second Depts* at Depth 2 on an updated graph. Out of 927 *Second Depts*, 731 were successfully restored. We repeated this process for a total of 10 depths and evaluated the number of vulnerable lib-vers over time, as shown in Figure 9.

In total, it took $4,110$ iterations to successfully restore $3,109$ version ranges with a 75.64% restoration rate. As a result, 90.32% of the vulnerable lib-vers were successfully remediated from Log4Shell, leaving only $1,144$. It is clear that the number of vulnerable lib-vers increases much more slowly over time after applying Ranger to the 10th dependents than the primitive state. This suggests that the propagation of the vulnerability was effectively suppressed from the beginning upon disclosure of Log4Shell. Moreover, it is observed that the number drops 45.95% when Ranger is only applied to the *First depts* at Depth 1, which indicates that *First depts* have a significant impact on downstream libraries yet not enough to suppress the propagation. Also, the marginal effect of Ranger drops fast as depth goes deep in the Figure and there were only 8 ranges to restore at Depth 9 and 10, which means Depth 10 is effective enough against persistent Log4Shell.

However, there still remained $1,144$ unfixed lib-vers requiring manual intervention. We categorized the remaining cases into three: (1) **No compatible patched versions to upgrade** (481 cases, 42%): Ranger found there was no version satisfying the constraints. For these cases, Ranger generated a report with breaking APIs and call chains with suggestions of manual fixes for developers to resolve the incompatibility. (2) **No secure versions available** (592 cases, 52%): This mostly happens for dependents at Depth 2+ because their direct dependencies may not have published a secure version that transitively depends on a patched version of *log4j-core*. It is a common case, especially for a newly disclosed vulnerability, for which, Ranger would suggest the developers find a substitution if the vulnerable library is reachable. If not reachable, a suggestion to exclude the vulnerable library would be suggested. On the other hand, Ranger will continue to monitor the availability of patched versions. (3) **Internal error** (71 cases, 6%): These were caused by issues irrelevant to the design of Ranger, namely, failed jar downloading, failed call graph generation, and the errors of compatibility checkers.

Finding 7: Our evaluation demonstrated that Ranger, as a plug-in, was successful in restoring secure version ranges for 63.49% of the 252 real-world GitHub repositories, with a high recall of 93.57%. In a simulated experiment, the server-side edition of Ranger was able to restore version ranges for $3,109$ (75.64%) of the dependents which successfully remediated $10,678$ (90.32%) of downstream vulnerable projects.

V. Discussion

• **Compatibility check should be aligned with SemVer, especially in Maven.** Many studies [22], [23], [25], [43], [45] have revealed that SemVer has not been well adhered to by developers in the Maven ecosystem, leading to prevalent SoftVer. Although Ranger could restore version ranges to include patched versions, the patched versions must be those already published. To timely apply the patches upon releases, version ranges have to be open ranges or semi-open ranges, e.g. caret range ^1.2.3 [46], which requires strict compliance with SemVer to assure compatibility. Therefore, in the long run, Ranger could mitigate the persistent vulnerabilities, but only the widely used and strictly backward compatible open version ranges could nip them in the bud.

• **More efforts and resources should be leaned on widely used but poorly maintained libraries.** As revealed by our evaluation in Section IV-B3, there were 592 cases without patched versions to upgrade to. Following a manual investigation, it was discovered that several libraries served as dependencies in a large number of projects or libraries, but their maintenance was inadequate. To address persistent vulnerabilities, it is imperative that the maintainers explicitly release patched versions for the benefit of downstream users. Therefore, this kind of libraries should arouse the collective awareness of the community, and the resources of open-source software governance should be directed towards these widely-used but poorly-maintained libraries to promote a more secure ecosystem.

VI. Threats of Validity

The primary threat of the study is the assumption that dependents of vulnerable libraries were considered affected without fine-grained reachability or triggerability analysis. Because analyzing the reachability of all vulnerabilities in the entire Maven ecosystem at a large scale is quite expensive, we did not take it into consideration. Furthermore, vulnerable libraries are also packaged into the deployment environment, and having vulnerability is not a secure practice because they could be exploited someday given the evolving source code. Hence, to promote the best security practice in the Maven ecosystem, we made such an over-assumption.

Another threat is the assumption that successful compilation and passing unit tests after applying version ranges generated by Ranger are sufficient to confirm successful version range restoration. However, in real-world software development, unit tests have limited coverage, and passing them does not necessarily guarantee that the restored version ranges satisfy all requirements of developers. Despite this limitation, unit tests are a critical component of deployment and are currently the most convenient validation approach available.

The last threat is the accuracy of the algorithm ALSearch that is used to track the downstream libraries. The first factor affecting the accuracy is the dependency graph sourced from MCR, and a few POM files in MCR could be unavailable

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

leading to incomplete dependency edges in the graph. Another factor is that the environment requirements of dependencies were ignored, which could lead to false positives because some dependencies are only installed in certain environments, such as Windows. However, these factors were proven to be corner cases in the validation experiment in Section II-B2 so that the overall conclusions are not undermined.

VII. Related Work

A. Persistence of Vulnerabilities

Researchers [9-11], [14], [16] have evaluated the propagation of vulnerabilities within the Maven ecosystem and recognized the long-term persistence of some vulnerabilities. Developers tend to address reachable vulnerabilities more than unknown ones due to the potential for exploitation, as revealed by Wu et al. [9]. Pashchenko et al. [11] found that upgrades to vulnerable dependencies are often delayed due to potential breaking changes. Li et al. [10] conducted a similar quantitative study using a dependency graph integrated with vulnerabilities. Benelallam et al. [14] proposed the Maven dependency graph that has been widely used for ecosystem vulnerability analysis. Plumber [16] proposed by Wang et al. is a viable approach to address persistent vulnerabilities in NPM but not applicable to Maven because it relies on the predefined version ranges prevalent in NPM. Although insights have been highlighted, they have not proposed any tailored solution for persistent vulnerabilities for Maven.

B. Remediation for Maven Vulnerabilities

Regarding Maven vulnerability remediation, many solutions have been proposed [12], [15], [19-21], [47-54]. Coral [47] is a systematic approach to address the vulnerabilities in the dependency trees of user projects. Du et al. [15] constructed a patch tracing system to locate patches to remediate the vulnerabilities. Industrial organization, OpenSSF [19], has proposed the best practice guidance [20] and a tool, Scorecard [21], for developers on managing vulnerabilities in dependencies for developers. Software Composition Analysis (SCA) [48], [55-58] tools have also been widely adopted to assist in the mitigation of vulnerabilities persistent in users' projects. However, these studies focused on user-oriented remediation by developers instead of ecosystem-wide vulnerability mitigation.

C. Dependency Versioning in Modern Ecosystem

Many researchers have recognized the significance of the dependency versioning scheme for the security and stability of open-software ecosystem. [2], [22], [25], [26], [45], [59] Dietrich et al. [26] studied 17 package managers to investigate the dependency versioning recommended by them and found Maven heavily uses SoftVer leading to low flexibility of dependency versions. Google [2] released a blog about persistent vulnerabilities like Log4Shell and pinpointed that the SoftVers could be a cause of the persistent vulnerabilities. Decan et al. [25] reviewed the SemVer compliance in 4 popular package managers and summarized guidance for developers to better comply with SemVer. These works highlight the limitations of dependency versioning in modern ecosystems, including the lack of flexibility in dependency management within Maven. As a solution, we proposed Ranger to restore the flexibility of version ranges within the Maven ecosystem.

VIII. Conclusion

In order to find a solution that addresses ecosystem-wide persistent vulnerabilities, we conducted an empirical study that revealed that 58.73% of vulnerabilities still impacted more than 50% of downstream libraries in the Maven ecosystem nowadays. Through this study, we quantitatively substantiated that blocked patches caused the persistence of vulnerabilities. The existing solutions are either not scalable or subject to breaking changes. Hence, we proposed Ranger as a scalable and automatic approach with compatibility assurance to unblock the vulnerability patches. Through evaluation, Ranger achieved 93.57% recall and restored $3,109$ (75.64%) ranges, which remediated $10,678$ (90.32%) vulnerable downstream projects.

• **Data Availability.** The experiment data set and algorithm are available at our website [35].

Acknowledgment

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner(s). This research is partially supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), the NRF Investigatorship NRF-NRFI06-2020-0001, the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. This research/project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] "Log4j Remote Code Execution," https://www.netskope.com/blog/cve-2021-44832-new-vulnerability-found-in-apache-log4j, 2021.
[2] "Google Open-source Insight," https://blog.deps.dev/, 2023.
[3] "Log4j Vulnerability News," https://www.securityweek.com/one-year-later-log4shell-remediation-slow-painful-slog/, 2023.
[4] "Log4j Vulnerability News," https://thenewstack.io/one-year-of-log4j, 2022.
[5] "Log4j Vulnerability News," https://securityintelligence.com/articles/log4j-vulnerability-changed-oss-cybersecurity/, 2023.
[6] "Log4j Vulnerability News," https://asia.nikkei.com/Spotlight/Datawatch/Cyberattacks-on-Japan-soar-as-hackers-target-vulnerabilities, 2023.
[7] "Log4j Vulnerability News," https://www.cybersecuritydive.com/news/cves-rise-2023-struggle-to-patch/641955/, 2023.
[8] "National vulnerability database," https://nvd.nist.gov/, 2023.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

[9] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zhou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *45th International Conference on Software Engineering*, 2023, pp. 1–12.

[10] Q. Li, J. Song, D. Tan, H. Wang, and J. Liu, "Pdgraph: a large-scale empirical study on project dependency of security vulnerabilities," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 161–173.

[11] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1513–1531.

[12] A. M. Mir, M. Keshani, and S. Proksch, "On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem," *arXiv preprint arXiv:2301.07972*, 2023.

[13] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 333–343.

[14] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: a temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 344–348.

[15] D. Du, X. Ren, Y. Wu, J. Chen, W. Ye, J. Sun, X. Xi, Q. Gao, and S. Zhang, "Refining traceability links between vulnerability and software component in a vulnerability knowledge graph," in *International Conference on Web Engineering*. Springer, 2018, pp. 33–49.

[16] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem," *IEEE Transactions on Software Engineering*, 2023.

[17] N. Imtiaz, A. Khanom, and L. Williams, "Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages," *IEEE Transactions on Software Engineering*, 2023.

[18] 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*, 2018, pp. 1–10.

[19] "Home - open source security foundation," https://openssf.org/, 2023, (Accessed on 02/12/2023).

[20] "ossf/wg-best-practices-os-developers: The best practices for oss developers working group is dedicated to raising awareness and education of secure code best practices for open source developers." https://github.com/ossf/wg-best-practices-os-developers, 2023, (Accessed on 02/12/2023).

[21] "Openssf scorecard," https://securityscorecards.dev/\#what-is-openssf-scorecard, 2023, (Accessed on 02/14/2023).

[22] 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*. IEEE, 2014, pp. 215–224.

[23] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[24] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the semantics into semantic versioning," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2020, pp. 157–179.

[25] 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, 2019.

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

[27] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.

[28] "Semantic Versioning," https://semver.org, 2021.

[29] "Maven Version ranges," https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, 2023.

[30] "Maven Soft Version Constraint," https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, 2023.

[31] "Maven repositories," https://mvnrepository.com/, 2023.

[32] "Github Security Advisory," https://github.com/advisories, 2023.

[33] "Google Open-source Database," https://docs.deps.dev/bigquery/v1, 2023.

[34] "Snyk Vulnerability Database," https://security.snyk.io/, 2023.

[35] "Data set," https://sites.google.com/view/ase23maven, 2023.

[36] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[37] "Gradle Dependency Constraint," https://docs.gradle.org/current/userguide/dependency_constraints.html, 2023.

[38] "Ivy Dependency Override," https://ant.apache.org/ivy/history/2.3.0/ivyfile/dependencies.html, 2023.

[39] "Maven Versions," https://maven.apache.org/pom.html, 2023.

[40] "Oracle Java Compatibility Documentation," https://www.oracle.com/java/technologies/javase/8-compatibility-guide.html, 2023.

[41] "revapi," https://revapi.org/revapi-site/main/index.html, 2021.

[42] "japi-compliance-checker," https://lvc.github.io/japi-compliance-checker/, 2019.

[43] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556956

[44] "Soot spark," https://www.sable.mcgill.ca/soot/doc/soot/options/SparkOptions.html, 2023.

[45] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in Maven central," *arXiv preprint arXiv:2110.07889*, 2021.

[46] "Caret Ranges," https://docs.npmjs.com/cli/v6/using-npm/semver#caret-ranges-123-025-004, 2023.

[47] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, L. Zhao, J. Wu, and Y. Liu, "Compatible remediation on vulnerabilities from third-party libraries for java projects," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 2540–2552. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00212

[48] "Software Composition Analysis," https://snyk.io/series/open-source-security/software-composition-analysis-sca/, 2023.

[49] 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)*. IEEE, 2015, pp. 520–524.

[50] R. G. Kulaa, C. De Rooverb, D. M. Germanc, T. Ishiob, and K. Inouea, "Modeling library dependencies and updates in large super repository universes."

[51] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," in *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 127–136.

[52] F. Massacci and I. Pashchenko, "Technical leverage in a software ecosystem: Development opportunities and security risks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1386–1397.

[53] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "Dismal code: Studying the evolution of security bugs," in *LASER 2013 (LASER 2013)*, 2013, pp. 37–48.

[54] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Sv-af—a security vulnerability analysis framework," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 219–229.

[55] L. Zhao, S. Chen, Z. Xu, C. Liu, L. Zhang, J. Wu, J. Sun, and Y. Liu, "Software composition analysis for vulnerability detection: An empirical study on Java projects," in *Proceedings of the 2023 31th acm sigsoft international symposium on foundations of software engineering*, 2023.

[56] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[57] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on third-party libraries in Android apps: A taxonomy and systematic literature review," *IEEE Transactions on Software Engineering*, 2021.

[58] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "Ossfp: Precise and scalable c/c++ third-party library detection

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.

using fingerprinting functions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 270–282.

[59] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 319–330.

Authorized licensed use limited to: NANKAI UNIVERSITY. Downloaded on October 09,2025 at 06:14:15 UTC from IEEE Xplore. Restrictions apply.