Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem

***Abstract*—Modern software systems are increasingly relying on dependencies from the ecosystem. A recent estimation shows that around 35% of an open-source project's code come from its depended libraries. Unfortunately, open-source libraries are often threatened by various vulnerability issues, and the number of disclosed vulnerabilities is increasing steadily over the years. Such vulnerabilities can pose significant security threats to the whole ecosystem, not only to the vulnerable libraries themselves, but also to the corresponding downstream projects. Many Software Composition Analysis (SCA) tools have been proposed, aiming to detect vulnerable libraries or components referring to existing vulnerability databases. However, recent studies report that such tools often generate a large number of false alerts. Particularly, up to 73.3% of the projects depending on vulnerable libraries are actually safe. Aiming to devise more precise tools, understanding the threats of vulnerabilities holistically in the ecosystem is significant, as already performed by a number of existing studies. However, previous researches either analyze at a very coarse granularity (e.g., without analyzing the source code and constraints) or are limited by the study scales. This study aims to bridge such gaps. In particular, we collect 44,450 instances of ⟨CVE, upstream, downstream⟩ relations and analyze around 50 million invocations made from downstream to upstream projects to understand the potential threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. Our investigation makes interesting yet significant findings with respect to multiple aspects, including the reachability of vulnerabilities, the complexities of the reachable paths as well as how downstream projects and developers perceive upstream vulnerabilities. We believe such findings can not only provide a holistic understanding towards the threats of upstream vulnerabilities in the Maven ecosystem, but also can guide future researches in this field.**

I. Introduction

The past two decades have witnessed a surge of *software reuse*. An increasing number of *third-party libraries* (TPL) have been released under open-source licenses, and the development of such libraries heavily depend on the infrastructures or functional components of each other. A recent estimation reveals that around 35% of an open-source project's code come from its depended libraries on average [1]. The numerous diverse libraries and their complex dependency relations naturally form large-scale social-technical ecosystems, the famous of which are the *Maven* for Java, *npm* for JavaScript, and *et al.* For instance, the Java ecosystem, managed mainly by *Maven*, has indexed over 9.51 million third-party libraries [2], which have facilitated the development of Java projects significantly for a long period.

Unfortunately, TPLs also suffer from various *vulnerability* issues, and the number of disclosed vulnerabilities in open-source libraries has been increasing steadily since 2009 [3]. Such vulnerabilities can pose significant security threats to the whole ecosystem, not only to the vulnerable libraries themselves, but also to the corresponding ***downstream*** projects (i.e., if we regard a library as an ***upstream***, those libraries depend on this upstream are denoted as the corresponding downstream). It has been estimated that around 74.95% of the TPLs that contain vulnerabilities are widely utilized by other libraries [4]. For instance, the recently spotted vulnerabilities in *Apache Log4j2* [5], have affected over 35,000 Java packages, amounting to over 8% of the Maven ecosystem [6]. The growing trend of the vulnerabilities discovered in open-source libraries results in the inclusion of "*vulnerable and outdated components*" in the OWASP Top 10 Web Application Security Risks [7], which is still ranked sixth currently.

The increasing number of exposed vulnerabilities in open-source libraries and the high-risk threats they pose to the ecosystem have attracted increasing attention from both the industry and academia. Automated tools aiming for detecting and assessing the vulnerabilities in open-source projects are booming over recent years, such as Dependency-Check [8] (*a.k.a. OWASP DC*) and Snyk [9]. The basic intuition behind most of these tools is to perform Software Composition Analysis (SCA) by analyzing the dependency configuration file or the information gathered during compilation, and then search for vulnerable components referring to existing vulnerability databases (e.g., *National Vulnerability Database* (NVD)). Warnings will be reported to developers once known vulnerable dependencies are detected, and developers are suggested to take mitigation actions such as *upgrading the dependency to non-vulnerable versions*.

However, recent studies spot that such tools will generate a high number of false alerts since they mainly work at a *coarse granularity* without analyzing whether the upstream vulnerabilities will actually *affect* the corresponding downstream projects [10]. In particular, Zapata *et al.* reveals that in the *node.js* ecosystem, up to 73.3% of the projects depending on vulnerable TPLs are actually *safe* [10]. Ponta *et al.* also reveals that 88.8% of the warnings generated by OWASP DC are false positives. Such a high rate of false positives will annoy developers, and our user study also confirms that some developers are bothered by such imprecise warnings and take unnecessary actions simply to get rid of such warnings (see Section VI-B). Even worse, such warnings will lead to unwanted upgrade of dependencies, which might further introduce other dependency conflicts or incompatibility issues [11], [12], [13]. Therefore, tools that are able to assess the security threats of upstream vulnerabilities to downstream projects more precisely are much desired.

To serve for such a practical need, we are motivated to perform a *holistic* and *quantitative* study to understand the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. We choose Java since it remains to be one of the most popular languages over the decades [14], and the ecosystem is also managed by matured tools (e.g., Maven). In particular, we perform analysis at a *finer* granularity at the source code level to understand to what extent are downstream projects threatened by upstream vulnerabilities in the ecosystem. Our main goal is to understand the likelihood the upstream vulnerabilities can be exploited by downstream projects and how to assess such likelihood. In particular, we investigate from multiple aspects, including whether the downstream projects are *reachable* to upstream vulnerabilities as well as the complexity of those *constraints* along the reachable paths. Besides, we also perform *quantitative* analysis and *a user study* to understand how downstream projects perceive upstream vulnerabilities, including the responses they make, the behind reasons as well as the developers' concerns. To enable the above analysis, we collect a large-scale dataset from the Maven ecosystem, including 837 CVEs with 1,141 patches corresponding to 615 different libraries in the ecosystem. We also collect another 29,960 unique downstream projects that depend on the above vulnerable libraries as upstreams. Such instances form 44,450 triplet relations of ⟨*CVE, upstream, downstream*⟩ (see Section III-A for more details). To our best knowledge, this is the largest dataset concerning the number of CVE with patches in the Java ecosystem together with the plenty of upstream/downstream relations. Our study mainly makes the following interesting yet significant findings.

1) Only around 25.7% of the libraries, the vulnerable functions can be reached by at least one of its downstream projects. On average, an vulnerable library affects 10.4% of its downstream projects. Among all the vulnerable functions, most of them actually cannot be reached by any of the downstream projects (i.e., 86.1%), and thus the security threats to other projects are limited.
2) Over half of the reachable instances, the length of the invocation path is more than five. For 27.3% of the reachable invocation paths, there are more than 10 associated constraints that are required to be satisfied in order to reach the vulnerable functions. Among such constraints, 39.5% of the variables are reference types, indicating that such constraints cannot be easily solved.
3) Via investigating among 49,766,554 invocations from downstream to upstream in the ecosystem, we find that around 4.4% are risky that are reachable to upstream vulnerabilities. For 27.1% of such risky invocations, the return value will further propagate in downstream (e.g., passed as parameters to other functions).
4) Before the discovery of a CVE in the ecosystem, the downstream projects have utilized the vulnerable upstream for 986 days on average. Upon the release of the CVE, 86.0% of the downstream have taken action such as *upgrading the vulnerable versions*. The response speed varies with an average time of 270 days.
5) Most of the downstream developers (67.0%) are aware of upstream vulnerabilities thanks to the wide adoption of SCA tools. However, existing SCA tools often achieve high positive rates. To assess the security threats of upstream vulnerabilities, most of the developers perform manual analysis based on their domain knowledge. Therefore, it still calls for precise automated tools to help assess the security threats in the ecosystem.

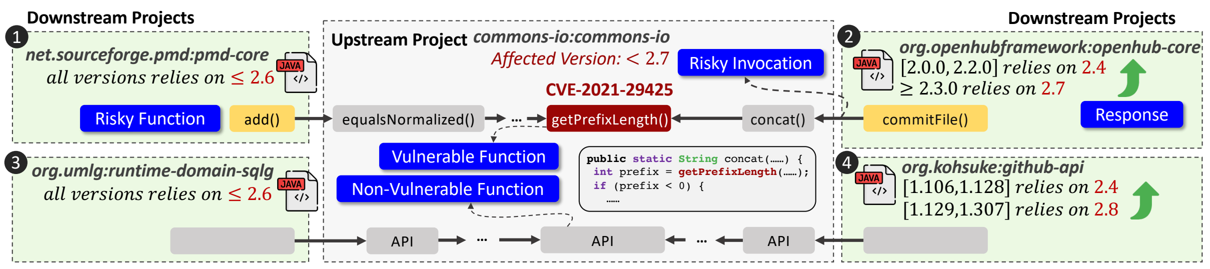
In summary, we make the following contributions:

- **Large-scale dataset.** Aiming to understand the threats of upstream vulnerabilities to downstream projects, we collect a large-scale dataset from multiple sources, forming 44,450 relations of ⟨CVE, upstream, downstream⟩. We open-source this dataset to facilitate future researches at: **https://github.com/MavenEcoSys/MavenEcoSysResearch**.
- **Comprehensive study.** We perform a comprehensive study to understand the reachability of upstream vulnerabilities from the perspective of downstream. Our study is performed at a fine granularity, which investigates the characteristics of the associated constraints and contexts. Besides, this study also investigates how downstream projects respond to upstream vulnerabilities.
- **User study.** This study performs a user study to understand how downstream developers perceive upstream vulnerabilities. We observe that despite most of them are aware of the upstream vulnerabilities, a large proportion do not take any action due to various reasons.
- **Significant findings.** We distill a series of original yet interesting findings based on our empirical analysis. Such findings cannot only provide a holistic view towards the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem, but also can guide future researches in this field.

II. Background & Motivation

A. The Maven Ecosystem

Java remains to be one of the most popular languages over the decades [14]. One major reason that contributes to the massive success of Java is its open-source ecosystem, which has attracted active contributions from many developers, thus forming a huge and complex ecosystem. Therefore, numerous Java third-party libraries (TPL) have been developed. Maven is now the most widely adopted management tool for the Java ecosystem [2]. Currently, Maven has indexed over 9.51 million Java libraries [2], and each of them can be differentiated by a unique identifier, which is denoted as GAV (i.e., *GroupId:ArtifactId:Version*). Developers can conveniently utilize a package by specifying the corresponding GAV of the package in the configuration files (e.g., `pom.xml`).



```
public static String concat(……) {
 int prefix = getPrefixLength(……);
 if (prefix < 0) {
   ……
```

Fig. 1: Vulnerable upstream project `commons-io` and four of its downstream projects

B. Library Vulnerabilities and Their Threats

The recently spotted vulnerabilities in *Apache Log4j2* [5], have affected over 35,000 Java packages, amounting to over 8% of the Maven ecosystem. Due to the pervasiveness of such critical vulnerabilities in TPLs and their extensive usage in the ecosystem, it is crucial to understand the impact of exposed vulnerabilities on other projects in the ecosystem. To help mitigate the threats of known vulnerabilities to other projects in the ecosystem, many automated tools have been proposed. For instance, OWASP DC is a widely-used SCA (Software Composition Analysis) tool in Maven [8], which attempts to detect vulnerable dependencies containing exposed vulnerabilities. Other SCA tools such as Snyk [9] and Github Dependabot [15] perform similar analyses but mainly differ in the vulnerability database they provide. Such tools will report for a project once its dependencies contain disclosed vulnerabilities and warn the corresponding developers to take actions, such as upgrading the version of the concerned dependencies. However, even if a project relies on vulnerable dependencies, it does not mean that the project will be *actually* affected by the vulnerabilities [10]. For such cases, upgrading/downgrading dependencies might be unnecessary, which might even introduce dependency conflicts [16], [17] or compatibility issues [18].

C. A Motivating Example

We use vulnerability CVE-2021-29425 exposed in project `commons-io` as an example to motivate this work. The vulnerability affects the versions below 2.7 of `commons-io`. It is introduced since developers forget to check the validity of the host name in function `getPrefixLength()`, which exhibits a high chance of introducing a path traversal vulnerability. Project `commons-io`, as an ***upstream*** library, has been utilized by 23,974 other projects (denoted as the ***downstream*** projects) in the ecosystem as indexed by Maven, and among which over 14,000 of them utilize those versions that are affected by CVE-2021-29425. For the affected version 2.4 only, it has been utilized by 4,386 downstream projects. Fig. 1 shows four examples of those downstream projects. Via manually analyzing this vulnerability and the corresponding affected downstream projects, we can make the following observations.

First, *certain downstream projects that rely on a vulnerable upstream are actually not threatened by the vulnerability.* As shown in Fig. 1, although all the four downstream projects depend on the vulnerable version of `commons-io` (i.e., version 2.4), not all of them can actually *reach* the vulnerable code. For `pmd-core` and `openhub-core`, they can reach the vulnerable function `getPrefixLength()` transitively. On the contrary, the other two projects (i.e., `github-api` and `runtime-domain-sqlg`) cannot access the vulnerable function by all means. Existing tools as aforementioned [8], [9], [15] will report for all the downstream projects to be vulnerable since they ignore the reachability to the vulnerable code, thus resulting in false positives.

Second, *even if the vulnerable function is reachable from downstream projects, the chance of exploitability is different.* Fig.1 shows the corresponding call graphs for the two projects that are reachable to the vulnerable function. As can be seen, the downstream transitively invoke the vulnerable function through other APIs such as `equalsNormalized()` and `concat()`. In particular, for `pmd-core`, it passes through four other functions to reach the vulnerable function while `openhub-core` only passes through one. Along the reachable call path, various constraints are required to be satisfied to reach the vulnerable code. Particularly, three constraints should be satisfied for `pmd-core`, which are `filename1!=null`, `filename2!=null` and `size!=0`. On the contrary, for project `openhub-core`, it can reach the vulnerable function directly without constraints once the invocation to `concat()` can be triggered. Such constraints along the path restrain the vulnerability from being exploited in downstream projects, and thus the complexity of them can reflect the extent to which the downstream is threatened by the vulnerability.

Third, *downstream developers take various action in response to upstream vulnerabilities.* As aforementioned, project `openhub-core` is reachable to the vulnerable function of CVE-2021-29425 without other constraints along the path. We observe that the project has upgraded the depended version of `commons-io` from 2.4 directly to 2.7, in which the vulnerability has been patched, since version 2.3.0 of `openhub-core`. However, for project `pmd-core` which is also reachable to the vulnerable function, we do not observe any action taken by developers in response to the vulnerability. On the contrary, project `github-api`, which is not reachable to the vulnerable function, has upgraded the depended version from 2.4 to 2.8. Therefore, it triggers our great interest to investigate the distributions of such downstream responses to upstream vulnerabilities and the behind reasons of such responses.

Based on the above analysis, we are motivated to investigate the pervasiveness of such observations among the whole Maven ecosystem. In particular, we are interested to answer the following research questions.

- **RQ1.** How is the reachability of downstream projects to the upstream vulnerabilities in the Maven ecosystem?
- **RQ2.** How hard it is for downstream projects to reach upstream vulnerabilities?
- **RQ3.** How do downstream developers respond to exposed vulnerabilities in upstream projects?

Answering such questions can bring the following benefits. First, it can provide all users of Maven a holistic view towards the extent to which current projects are threatened by upstream vulnerabilities in terms of reachability. Second, it can reveal the characteristics concerning the exploitability of upstream vulnerabilities, and further facilitate the construction of models to assess the threats of upstream vulnerabilities. Such assessment models can help reduce the false positives of existing automated tools (e.g., SCA tools). Third, it can help understand how downstream projects and developers perceive upstream vulnerabilities. Eventually, it can facilitate the security assurance of vulnerabilities across various projects and ensure supply chain security in the Maven ecosystem.

III. Study Methodology

In this Section, we first present the dataset construction to facilitate such a large-scale empirical study. We then present the employed methodologies to answer the designed questions.

A. Data Collection

To perform a large-scale study to understand to what extent are downstream projects affected by upstream vulnerabilities, we need to construct a dataset of known vulnerabilities of open-source projects, which, particularly, should satisfy the following criteria. First, the corresponding patches should be available since patches are required to identify the vulnerable components and functions, which are used to investigate the reachability of those vulnerabilities from the perspective of downstream projects. Second, the vulnerable project should be indexed by Maven, which can be used as upstream projects by other downstream ones. Third, the vulnerable upstream should be used by a sufficient number of downstream projects to facilitate our analysis. Specifically, we construct the benchmark dataset as follows.

1) Vulnerability Collection: We first collect a set of exposed vulnerabilities with patches. We observe that there are many existing studies that have created different datasets of vulnerabilities with patches of Java open-source projects [19], [20], [21], [22], and each of them contains hundreds of CVEs with patches. After deduplication, we extract 993 unique CVEs with 2,084 different patches (i.e., a patch refers to the fixing *commit* of the CVE) from the existing four studies. Aiming to create a larger dataset, we find that the Veracode vulnerability database [23] also provides the information of CVEs with the corresponding patches as well as the vulnerable versions of the upstream project. Therefore, we implement a crawler to automatically extract CVEs with patches from this dataset and enhance the number of CVEs from originally 993 to 1,647 with 2,813 different patches.

To further expand the dataset, we implement a crawler based on the following two heuristics. 1) Searching the commit log message of Java open-source projects to see if it contains the information for fixing specific CVEs. 2) Searching the official webpage of open-source projects (if have) to look for CVE information. Using the crawler, we further enhance the number of CVE entries from 1,647 to 1,834 after deduplication.

We further manually check all the collected instances and remove certain invalid data. For instance, the links to certain commits might be invalid. Besides, some fixing patches with different commit IDs might be the same since they refer to the patches at different branches. Eventually, we select 1,712 unique CVEs with 3,448 different patches after filtering.

2) Vulnerable Function Localization: We then obtain the affected vulnerable functions based on the code modifications performed by the CVE patches. Identifying the vulnerable functions is necessary since we need to analyze the reachability of vulnerabilities as well as their potential threats. In particular, we deem a function vulnerable if it exists in the version before the CVE patch is employed and has been modified by the patch. For those newly introduced functions by the fixing patch, we do not regard them as vulnerable. Unfortunately, we observe that there are many patches that perform function-irrelevant modifications. For instance, some patches do not modify any Java source file while editing the configuration files instead. After removing such cases, 1,421 CVEs with 2,217 patches have been left.

3) Affected GAV Identification: Many Java projects consist of multiple components, and each of them will be released as an individual `jar` file and identified by a unique GAV (i.e., *GroupId:ArtifactId:Version*) in Maven, a different naming system from CPE (i.e., Common Platform Enumeration) provided by NVD. The GAV is required to search for downstream projects in Maven. However, except for the Veracode vulnerability which provides GAV for each upstream of CVE, it is non-trivial to obtain the GAV identifier directly based on the corresponding patch of other sources. In this study, we choose to identify the affected component via manual analysis, and eventually we kept 1,085 CVEs corresponding to 839 unique artifacts in the Maven ecosystem after this step.

4) Downstream Software Collection: We construct the dependency graph of the entire Maven ecosystem using Eclipse Aether [24] to obtain the dependency relationship. Notably, we randomly select one version of the same downstream software that depends on certain upstream software to avoid data redundancy, and we further filter out the downstream that share the same GroupID with the upstream [25], for they most likely come from one project, which might introduce bias for our analysis. This step filters those upstream software that has no downstream software.

Finally, we keep 837 CVEs with 1,141 patches corresponding to 615 different upstream software in the ecosystem. On average, for each upstream, we collect 72 different downstream projects, thus forming 44,450 triplet ***instance*** of $\langle$CVE, upstream, downstream$\rangle$. Those relations concern 29,960 unique downstream projects since a downstream might rely on mul-

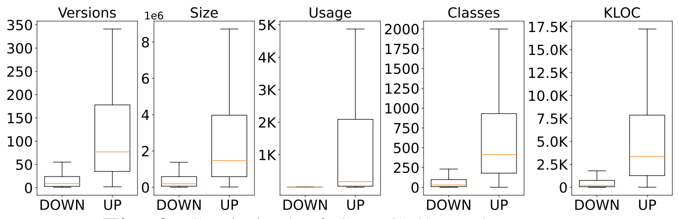


Fig. 2: Statistical of Our Collected Dataset

tiple vulnerable upstreams. Fig. 2 shows the statistics of our collected data. It shows that the included upstream and downstream projects are (1) large-scale (upstream contains 75.6 KLOC on average with 1,082 classes); (2) well-maintained (an upstream release 117 versions on average while a downstream release 25); (3) impactful (over 37.4% of the upstream have over 500 direct usages).

B. Methodology

To answer the research questions as listed in Section II, we employ the following methodologies.

1) RQ1:Reachability: To understand the reachability of downstream projects to upstream vulnerabilities for each instance, we first analyze the CVE patches to extract the **vulnerable function**. A function that contains a vulnerability (i.e., modified by the corresponding vulnerability fixing patch) is denoted as a *vulnerable function*. We then construct the *call graph* (CG) for the upstream and downstream, and then investigate whether there exists paths from the downstream to the upstream vulnerable functions. In particular, we define the following concepts and extract the corresponding information.

Reachability. A function p is reachable to another function q if there exists a path from p to q in the corresponding CG, and we denote such a path as **Reachable CG Path**.

Risky invocation. An invocation to an upstream API in the downstream is ***risky*** if the invoked API is vulnerable or is reachable to any vulnerable function in the upstream. As shown in Fig. 1, invocations to `concat()` made in function `commitFile()` and `equalsNormalized()` in `add()` are risky since they can all reach the vulnerable function.

Risky function. We denote those downstream functions that make risky invocations as risky functions. As explained above, `commitFile()` and `add()` in Fig. 1 are risky functions since they make risky invocations.

Downstream projects become vulnerable through risky invocations. Such risky invocations and functions can reflect whether the downstream is reachable to the upstream vulnerabilities. Therefore, we extract the above information from the dataset and analyze their distributions to answer RQ1.

2) RQ2:Reachable Constraint and Contexts: In RQ1, we merely investigate the reachability in terms of CG path. However, it is not necessary that the vulnerability can be *exploited* even if it is reachable since those constraints along the path as well as the context of the risky invocations can restrain the vulnerability from being exploited in downstream projects. Therefore, in this RQ, we are motivated to investigate at a finer granularity by inspecting the code at each function along the CG path. Specifically, we construct the *inter-procedure control flow graph* (ICFG) for each reachable CG path, extract the associated constraints and then characterize their features. In particular, we perform our analysis from three perspectives.

First, we investigate the **Reachable ICFG Path Ratio**, which is measured by the number of paths in the ICFG that are reachable to the vulnerable function over the total number of paths for each reachable CG path. A higher ratio indicates higher threats since the chance from the risky invocation to the vulnerable functions is higher. To tackle the path explosion problem [26], we unroll loops only once and eliminate paths with contradicted constraints. However, the paths can still be explosive since our analyzed subjects are often complex and in large-scales. To avoid this problem, we regard the ratio as 0 if the total number of different paths exceeds 10^5.

Second, we characterize the constraints associated with each reachable ICFG path, including the number of constraints, the type of *operators* and *variables* involved. Finally, we investigate the contextual information surrounding the risky invocations in the downstream. Specifically, we investigate whether the risky invocation is guarded by any *if checkers* or *try-catch* statements (guarded/not guarded). If so, the security threats induced by the risky invocation might be mitigated. We also investigate if the return value of the risky invocation or its tainted values propagates (e.g., through value return or passing to other functions as parameters) in the downstream. If such cases are observed, the security threats induced by the risky invocation might be worsened.

3) RQ3:Downstream Response: In this RQ, we aim to understand whether the downstream projects have taken action in response to those vulnerable upstreams. For each CVE, we know the *affected versions* of the corresponding vulnerable upstream (see Section III-A). Therefore, we denote those modifications in the downstream that change the dependency of the vulnerable upstream from *affected versions* to *unaffected versions* as ***responses***. Such modifications usually include *upgrading/downgrading* the versions of the dependency or *removing* the vulnerable dependency [27], [28]. As shown in Fig.1, downstream projects `openhub-core` and `github-api` have made responses to the vulnerable upstream. Specifically, they have upgraded the dependency from affected versions to unaffected ones (i.e., the affected versions are `(, 2.7)`).

To answer this RQ, we first identify the corresponding *GitHub repository* from libraries.io [29] for each GAV in our dataset. Be noted that we exclude those instances whose repository information cannot be identified, where we keep only 24,000 instances. Moreover, one repository sometimes corresponds to several GAVs, and we guarantee that one repository occurs only once, because there is only one group of contributors in one repository and including it several times willl cause bias to the statistics. We keep 13,000 instances here. We also remove those projects that are not actively maintained since such projects are less likely to contain useful information towards security maintenance. In particular, we only keep those repositories that contain more than 100 commits and also contain at least one commit within one year after the corresponding CVE is published. Finally, we keep

4,073 instances ⟨CVE, upstream, downstream⟩ for our analysis in this RQ. Specifically, we mine from the commits of the repository to see if they contain targeted responses as defined above, and then investigate the distributions of the response ratio and speed of downstream projects.

However, we find that the behind intuitions for such responses (i.e., modifying the dependency of the vulnerable upstream) are complex (e.g., resolving dependency conflicts issues [30], [31]) and developers usually do not specify specific intentions when modifying such dependencies. Despite the fact that such responses can relieve the downstream project from being affected by the vulnerable upstream, whether the intention of such commits are actually related to the vulnerability issue remains unknown. Therefore, it further motivates us to perform a user study to further understand the real intention of such extracted responses.

In particular, we randomly send the questionnaire to 500 developers who make the response to the CVE (e.g., committing the changes in response to the vulnerable dependency) from 714 developers responding to CVEs in recent three years. For those downstream projects which have not taken any responses, we send the email to the developer with the most contributions to the configuration file, and finally select 500 random developers from all developers without a response (725). We want to make a trade-off between guaranteeing a comprehensive survey and avoiding spamming the open-source community out of ethical considerations, and thus we choose not to send it to each developer. We expect to receive 100-150 replies, and we think this is a sufficient number (as other studies [27], [28] usually perform user studies at similar or smaller scales). As Ma et al. [28] indicates, the reply rate is between 14%-20%, so we send 1,000 emails in total, with response/no response to the upstream vulnerability half to half.

The questionnaire is sent to the selected developers by email, specifying the downstream software, the upstream software that contains the vulnerability, and the corresponding CVE. Eventually, we receive 64 emails from those developers with responses to the CVE and 45 emails without, leading to a response ratio of 10.9%. Particularly, we are interested to know *whether downstream developers are aware of the upstream CVE*, *the real intention behind the responses*, and *how they usually assess the security threats*.

IV. RQ1: Vulnerability Reachability

In this Section, we analyze the reachability of downstream projects to the upstream vulnerabilities. In particular, we investigate from the perspective of upstream and downstream respectively to understand the threats of exposed vulnerabilities in the ecosystem.

A. Upstream Vulnerability

To understand the extent to which the upstream vulnerabilities are accessed by the downstream projects, we first analyze them from the perspective of upstream vulnerabilities. In particular, we analyze the distributions from two granularities, *vulnerable function* and *vulnerable upstream*.

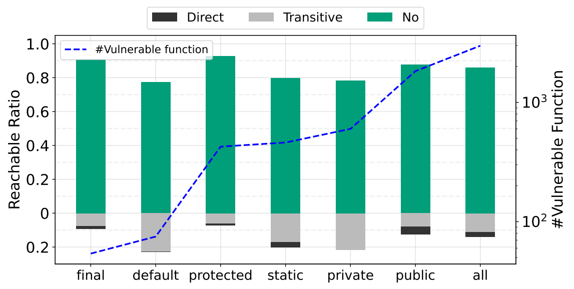


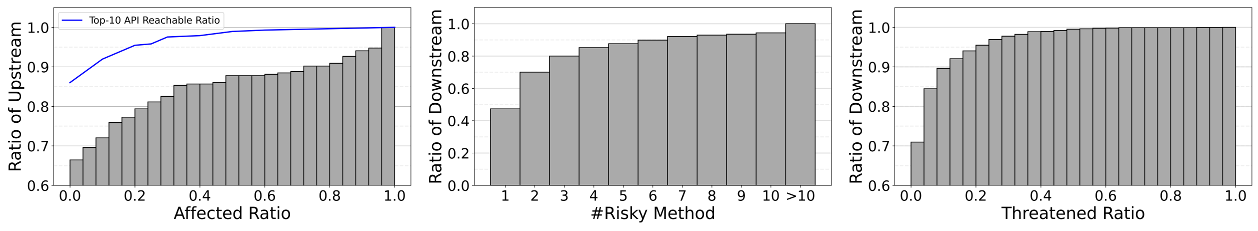
Fig. 3: The reachable ratio for vulnerable functions with different modifiers, the line shows the amount of functions.

Vulnerable Function. As aforementioned in Section III-A, we in total collect 2,990 vulnerable functions in this study, and Fig. 3 shows the distributions of them that can be accessed by the corresponding downstream projects separated by different modifiers. On average, 2.9% of the vulnerable functions can be directly accessed while 11.0% can be transitively accessed. Besides, vulnerable functions with all types of modifiers, including those *private* and *default* functions that are not visible to downstream projects, can also be accessed by downstream projects either directly or transitively. The direct access ratio is the highest (i.e., 4.8%) for *public* vulnerable functions. It falls into our intuition that such cases can pose great threats to the downstream projects since the vulnerable functions can be directly accessed by the corresponding downstream.

Finding 1. *Most of the vulnerable functions (i.e., 86.1%) actually cannot be accessed by the corresponding downstream projects, thus posing no security threats to the downstreams. Besides, those private and default vulnerable functions are also likely to be accessed by downstream projects.*

Vulnerable Upstream. As for the vulnerable upstream project, we find that 9.9%, and 22.9% of them can be directly or transitively accessed by downstreams respectively. Moreover, three libraries that can be accessed are found to be the top 10 popular project [32] in the Maven ecosystem, which are *junit*, *commons-io* and *jackson-databind*.

As aforementioned, we collect on average 75 downstream projects for each vulnerable upstream project. We then investigate the percentage of the downstream projects that are reachable to the vulnerable methods, and Fig. 4a shows the results. For those vulnerable upstream projects that are *actually used* by its downstreams (i.e., the downstream invokes at least one API in the upstream), we find that on average, a vulnerable upstream affects 10.4% of its downstream projects. For 28 different upstreams (e.g., *httpclient*:4.3), over 80% of the downstream projects can reach the vulnerable functions, thus posing significant security threats to the ecosystem. We also make an interesting observation that there are certain *commonly used APIs* in the upstream that are reachable to the vulnerable functions. Such APIs are usually the entry points to the upstream (i.e., a constructor), once such APIs are reachable to the vulnerable functions, downstream software will be affected for a large proportion. Identifying such APIs is significant since if we can take precautions to them



(a) Affected Ratio v.s. Upstream Ratio **(b)** The Number of Risky Function **(c)** Threatened Ratio/Downstream Ratio

Fig. 4: The Distribution of Reachable Ratio and the Number of Risky Function

(e.g., adding sanitizers), the threats of upstream vulnerabilities to downstream projects can be significantly mitigated.

Finding 2. *Around 25.7% of the vulnerable upstream projects can be reachable by the corresponding downstream projects. A vulnerable upstream threats 10.4% of its downstream projects, on average, in terms of reachability. There are popular upstreams and commonly used APIs that lead substantial downstream projects to become vulnerable, which deserves more attention to ensure ecosystem security.*

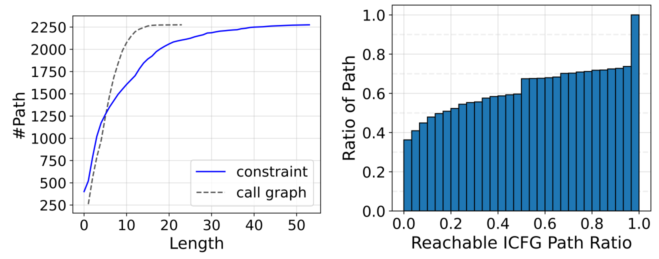
B. Downstream Projects

In this subsection, we aim to understand the extent to which downstream projects are threatened by the upstream vulnerabilities. In particular, we analyze from two perspective: *risky invocation/ function* and *downstream threatened ratio*.

Risky Invocation and Function. Downstream projects become vulnerable through risky invocations (as defined in Section III-B). We identify 49,766,554 unique invocations made by the downstream projects to upstream libraries in our dataset in total. It turns out that 4.7% of such invocations are risky that expose the downstream software to danger. In particular, a downstream might contain multiple risky functions, and Fig. 4b shows the distribution. In particular, 1,265 different downstream projects contain at least two risky functions, and 136 contain over 10 risky functions.

Threatened Ratio. To further understand the potential threats of upstream vulnerabilities to downstream projects, we analyze the *threatened ratio* for each downstream. The threatened ratio measures the proportion of the functions in the downstream that are reachable to any risky functions (including risky functions themselves) over the total number of functions in the downstream. A higher threatened ratio indicates a larger proportion of the downstream functions will be potentially threatened by vulnerabilities. Fig. 4c shows the distributions, and we can see that for most of the downstream projects (i.e., 88.1%), the threatened ratio is less than 10.0%. For a small proportion of downstream (i.e., 0.5%), the threatened ratio is over 50%, indicating that over half of the functions in the downstream are reachable to upstream vulnerabilities.

Finding 3. *Around 4.7% of the invocations from the downstream to upstream are risky. As for the threatened ratio of downstream, 88.1% is less than 10.0%, which means for most of the downstream, only less than 10.0% of its functions are reachable to the upstream vulnerabilities.*



(a) Constrain/Path Length **(b)** Cumulative Reachable Ratio

Fig. 5: Path Reachable Ratio and Path Length

V. RQ2: Reachable Constraints and Contexts

As illustrated in Section III-B2, in this RQ, we explore from following three aspects: *reachable path ratio*, *the characteristics of path constraints* and *risky invocations' contexts*.

A. Reachable ICFG Path Ratio

In Fig. 5a, we show the length distribution of all the reachable CG paths, and we can see that the maximum length is 23, indicating the downstream needs to pass through 23 different functions to reach the vulnerable functions. The length of 42.7% of such reachable CG paths is less than 5, and that ratio is 91.2% for the length of 10. However, there might exist many constraints along a reachable CG path if we consider the *control-flow* in each function. Therefore, we construct the *inter-procedure control flow graph* (ICFG) between the downstream and upstream, and then measure the *reachable ICFG path ratio* as introduced in Section III-B2.

Fig. 5b shows the results. We can find that for 25.3% of the reachable CG paths, the reachable ICFG path ratio is 1, which means all the paths in the call path can reach the vulnerable function. Besides, for 32.7% of the reachable CG paths, the ratio is over 50%, indicating that among all the ICFG paths along the CG path, half of them can eventually reach the vulnerable functions. In addition, for around 29.7%, the reachable ICFG path ratio is extremely low (i.e., <0.01), indicating a limited degree of security threats.

B. Path Constraints

We first investigate the distribution towards the number of constraints along each reachable CG path, and Fig. 5a shows the results. We can see that for 17.6% of the reachable CG path, the number of constraints is 0, indicating that the vulnerable functions can be directly accessed. For 29.5% of the CG paths, the number of constraints is more than 10, which indicates that for such cases, at least 10 different constraints

TABLE I: The distribution of constraint characteristics

Operator		Variable Source		Variable Type	
Type	Ratio	Type	Ratio	Type	Ratio
$\neq$	46.70%	Invocation	55.10%	Primitive type	60.50%
$==$	46.00%	Filed	16.40%	Other Reference Type	36.20%
$\geq$	2.30%	Parameter	15.40%	String	2.20%
$>$	2.10%	Constant	13.10%	Array	1.10%
$<$	1.90%				
$\leq$	1.00%				

are required to be resolved in order to reach the vulnerable functions. Such a ratio is 8.0% for the length of 20.

Finding 4. *The length of 42.7% of the reachable CG paths is less than 5. For 29.5% of such paths, at least 10 constraints are required to be resolved to reach vulnerable functions.*

We then investigate the characteristics of such constraints, in particular, from the perspectives of the *variable* and *operator* involved. Table. I shows the statistical results. With respect to the operators, since we analyze at the Soot IR code [33], which is a typical form of *three-address code*, all the operators of condition expressions can be classified into the six categories. In particular, the unequal (46.7%) and the equal mark (46.0%) take the majority proportion. The equal mark is the hardest to be satisfied in practice, which will often lead the path to be infeasible [34]. With respect to the variable type, we can observe that variables of the primitive types (e.g., int, float, double) take the largest proportion (60.5%). Reference type of String and Array collectively take the proportion of 3.3% while the rest of the reference types (e.g., self-defined classes) take the proportion of 36.2%. We are also interested in where the values of those variables in the constraints originate. Therefore, we trace the *def-use* chains of those variables and summarize four common sources: *return value of invocation*, *parameters of the functions*, *filed reference* and *constant defined in the function*. We find that the return value of invocation takes the largest proportion (55.1%) while the other three take similar proportions. Particularly, a non-negligible proportion (13.1%) of the constraints involve constant values.

Finding 5. *Variables of reference types and those originated from invocations are often involved in path constraints. Therefore, inter-procedure analysis is necessary when assessing the security threats among upstream and downstream.*

C. The Context of Ricky Invocations

As discussed in Section III-B2, we mainly investigate two types of contextual information: (1) whether the risky invocation is guarded; (2) whether the return value of the risky invocation propagates (i.e., used as return or parameters for other functions). Fig. 6 shows the distribution results. The category exhibits the highest threat is $\langle NG, R, P\rangle$ (i.e., not guarded but used as both return value and parameters), which accounts for 0.04% of the cases. Besides, 27.1% of the risky invocations can propagate return value $\langle G/NG, R, P\rangle$, which may induce further threats to the downstream software. Even worse, among these risk invocations, 12.5% can access the vulnerable function without any constraints.

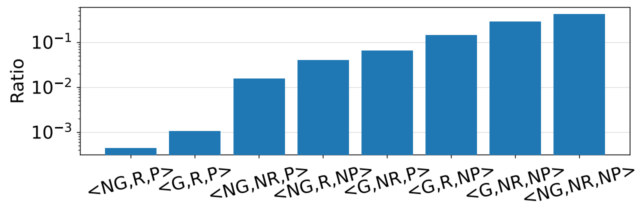


Fig. 6: Context distribution of risk invocations. G/NG denotes guarded/not guarded; R/NR denotes returned/not returned; P/NP denotes used as parameters/not used as parameters.

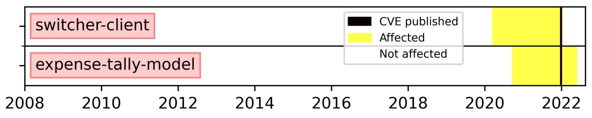


Fig. 7: Two affected downstream of CVE-2021-44832

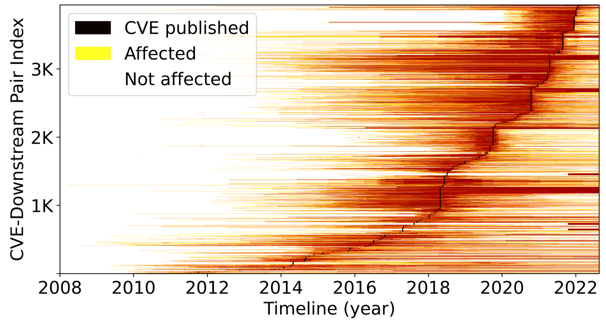


Fig. 8: Upstream vulnerability threats to downstreams, sorted by the CVE release time as indexed by the **black line**.

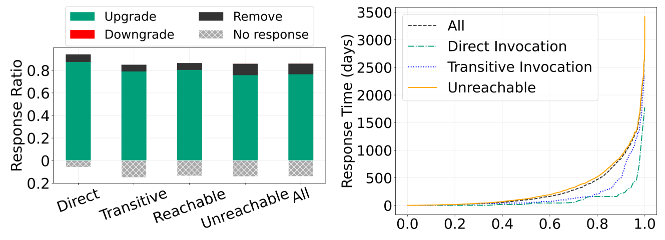
Finding 6. *Around half (48.9%) of the risky invocations are not guarded by any checkers, and the return value of 27.1% of the risky invocations will propagate.*

VI. RQ3:Downstream Response

In this Section, we aim to understand how downstream projects and developers respond to upstream vulnerabilities.

A. Downstream Response Ratio and Speed

Overview Distribution. We first aim to obtain the overview picture of how downstream projects are affected by upstream vulnerabilities and how they respond during software evolution. For each triplet, we extract the upstream's version specified in the downstream's configuration to see if it is affected by the specific CVE. We then track the specified version in the downstream to see if they change during software evolution. Fig. 7 shows two affected downstream, `switcher-client` and `expense-tally-model`, of an upstream `log4j-core`, which is affected by CVE-2021-44832. These two projects depend on the vulnerable version of `log4j-core` and became affected starting from 2020.03.09 and 2020.09.13 respectively. The vulnerability was discovered and a CVE was indexed on 2021.12.28. Until the discovery of the vulnerability, the two downstream have utilized this vulnerable upstream for 659 and 471 days. Upon the discovery of the CVE, `switcher-client` has upgraded the version of `log4j-core` to an unaffected version as shown in Fig. 7 in 12 days, thus getting rid of



(a) Response distribution of different reachable condition **(b)** Interval between CVE published time and response time.

Fig. 9: Response Ratio and Response Speed

the vulnerability issue. Similar action has also been taken by downstream `expense-tally-model` while it takes 153 days.

We make the above analysis for all collected instances to obtain an overview picture towards how the downstream projects are affected by existing vulnerabilities currently in the ecosystem, and Fig. 8 shows the results. In particular, we choose the time from 2008.01.01 to 2022.08.17 since the earliest CVE in our dataset was published in 2008. Each horizontal line presents a CVE-Downstream pair. Fig. 8 shows that downstream projects are often affected by upstream vulnerabilities for quite a long time before the corresponding CVE was exposed (on average 986 days). Upon the release of the CVE, a large proportion (i.e., 86.0%) of the downstream projects have taken action, and the average time taken is 270 days. Unfortunately, we also observe no responses to the upstream vulnerabilities for the rest of the downstream projects until 2022.08.17. Be noted that several thick red bars appear in Fig. 8. This is because, for each CVE, we have clustered its downstream together, while each horizontal line is originally yellow and the cluster becomes a thicker red bar when the affected yellow lines are stacked together.

Finding 7. *Before the discovery of the CVE, the downstream projects utilized the vulnerable upstream for 986 days on average. Upon the release of the CVE, 86.0% of the downstream projects have taken actions such as upgrading. The response speed varies with an average time of 270 days.*

We then investigate the distributions of the response ratio and speed in detail in terms of their reachability as follows.

Response Ratio. We show the response ratio together with various reachable conditions in Fig. 9a. The results also show that for those downstream that can directly reach the vulnerable functions, the response ratio is the highest (i.e., 94.2%). In particular, 87.4% of the downstream projects have upgraded the version of the vulnerable upstream, and 6.8% of the downstream projects have removed the vulnerable upstream. It falls into our intuition since such cases exhibit the highest security threats. However, we do not observe an obvious difference between reachable cases in total and unreachable ones in terms of response ratio.

Response Speed. We further investigate the speed of the response taken by developers. In particular, we measure the interval (i.e., in days) between the CVE published time and response commit submitted time. Fig. 9b shows the statistical results that the response speed for those reachable cases is much faster than that of unreachable ones. In particular, for 80% of the cases, downstream with direct invocations take action within 161 days while such a number is 524 for unreachable cases. For 90% of the cases, downstream with direct invocations take action within 217 days while the number is 506 for transitive invocations and 890 for unreachable cases.

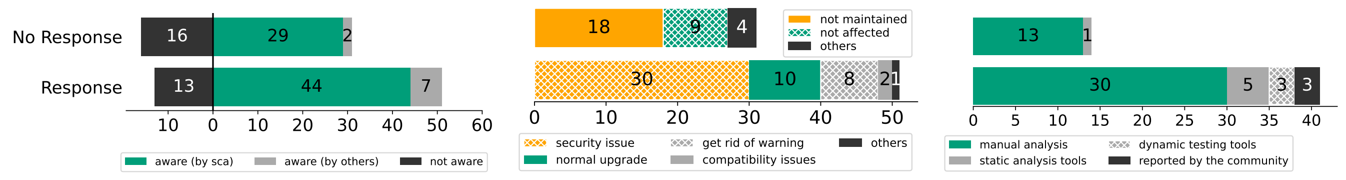
Finding 8. *Despite the fact that the intuition behind the action taken by developers is complex, downstream projects with direct or transitive access to upstream vulnerable functions are more likely to take action in a quicker way.*

B. Developer Survey

Among all the survey responses, we observe that 81.7% of them have over ten years of programming experience. As shown in Fig. 10a, we observe that most (72.2%) developers are aware of the target CVE. Specifically, such a ratio is higher for those cases with responses (79.7%) than those without (68.9%). At the same time, we find that 73 (89.0%) of all aware developers notice the CVE due to SCA tools. The high ratio indicates that SCA tools are widely used in the ecosystem and are also playing a significant role in prompting developers to be aware of vulnerable dependencies.

For developers who make no responses but are aware of the CVE, Fig. 10b shows the behind reasons. The results reveal that for over half the cases (18/31), the software is no longer maintained. However, such abandoned libraries widely exist in the ecosystem and can be further used by other libraries, thus posing significant security threats to the ecosystem. Another main proportion (9/31) of the developers find that the project is actually *not affected* by the upstream vulnerabilities, thus taking no action. For those projects where we observe responses to the CVE and the developers are also aware of the security issue, the behind reason also varies as shown in Fig. 10b. Only around 58.8% (30/51) of the responses are made to mitigate the threats of upstream vulnerabilities, which indicates for certain cases, the upstream vulnerability may not affect the project or the developers think the vulnerabilities are insignificant to be repaired. In particular, 10 (19.6%) responses are made due to simple regular upgrades (e.g., upgrading the dependency for new features but accidentally eliminate the security threats). Eight (15.7%) responses are made to simply get rid of the warnings from automated tools such as OWASP DC [8] and Snyk [9], among which only one developer verifies that his software is indeed affected by the upstream vulnerability, indicating the high false positives of SCA tools. Other two (3.9%) responses are made for compatibility issues.

For those developers who claim that they are aware towards the threats of upstream vulnerabilities, we further investigate how they identify and assess such threats. As Fig. 10c shows, most of the developers (51.8%) manually inspect whether the vulnerability would threaten their software via manual analysis based on domain knowledge. A small percentage (17.9%) of them analyze through dynamic or static analysis tools. Such results indicate the lack of precise automated tools to help assess the threats of security issues among the ecosystem.



(a) Awareness of CVE **(b)** Reasons for Responses **(c)** How the Threats were Assessed

Fig. 10: The Major Results of Our User Study

Finding 9. *Most of the downstream developers (67.0%) are aware of upstream vulnerabilities thanks to the wide adoption of SCA tools. Downstream projects do not take responses to upstream vulnerabilities since the projects are not affected or not maintained. Upstream vulnerability threats could be mitigated due to other non-security reasons, such as normal upgrade and addressing compatibility issues.*

In addition, we attach a comment section at the end of the survey to allow developers to raise any opinions about the upstream CVE and ecosystem security in general. Several developers give positive comments towards SCA tools and think they are critical to resolving security issues. However, another two developers think the compatibility issues cost much more than the security issue. If developers rely on SCA tools to determine if an upgrade is necessary, they may need to invest a lot of effort to resolve the collateral compatibility issues. In particular, they comment "*this particular upgrade breaks backward compatibility in bad ways, and thus the cost is too high.*", which reveals that more accurate metrics for assessing potential security threats are much needed. One developer also shows interest in our research by commenting "*I'd love to know the results of your research*".

VII. Related Work

Security Issues of Third-party Libraries. Many works have performed comprehensive studies aiming to understand the security issues in various ecosystems (e.g. JavaScript, Python, and Ruby) [35], [36], [37], [38], [39], [40]. They target the whole ecosystem or the supply chain and investigate several kinds of security issues. Several studies [41], [42], [43] perform studies on the usage and effectiveness of SCA tools in actual production to resolve vulnerable dependencies. Derr et al. [44] perform the first study concerning outdated dependencies and the security issues in Android. Cox et al. [45] propose several metrics to measure the outdatedness of dependencies. Ma et al. [28] collect cross-project correlated bugs and design a survey to study the response of developers for Scientific libraries in the Python ecosystem. Kula et al. [27] investigate to what extent do developers update their dependencies and how they respond to vulnerability advisories in the Java ecosystem. They find that 81.5% of the selected systems still keep outdated dependencies and do not respond to the vulnerability. Wang et al. [46] conduct an analysis on 806 open-source projects and 544 security bugs to study the usage and the update of dependencies and analyze security issues in the third-party library. Different to the above studies, we perform cross-project analysis to understand the upstream vulnerabilities with a specific focus on the Maven ecosystem in this study. Besides, our analysis performs at a finer granularity not only including the analysis towards CG but also including the ICFG with the associated path constraints.

Reachability of Cross-project Vulnerabilities. Recent studies propose to consider the reachability of the corresponding vulnerable function to advance existing tools. In particular, Ma et al. [47] leverage symbolic execution on the downstream to verify the reachability of the functions with general bugs in the upstream. However, they target general bugs instead of vulnerabilities, and focus on the scientific libraries in the Python ecosystem. They claim that they have extracted the conditions from the bug reports, which is probably infeasible for vulnerabilities to get from CVE description. Serena et al. conduct several studies [48], [25], [49] on the reachability of upstream vulnerabilities. In addition, they combine static analysis and dynamic execution of test suites to estimate the reachability of CVEs in the Java ecosystem [50]. However, the tool it provides (Steady) only considers the reachability from call graph using static analysis. They also try to improve Evosuite to dynamically estimate the reachability of upstream vulnerable functions [51]. Unfortunately, the tool is only experimented on crafted examples, and the practical usefulness is compromised. Besides reachability, our study also considers the callsites' context and the constraints along the reachable ICFG path to understand the threats.

VIII. Threats To validity

Data collection. Although we have collected a large number of CVEs with patches as well as upstream and downstream projects, we are unable to cover all the CVEs in Maven, mainly because the patches corresponding to the CVEs are often inaccessible. However, we have collected as much data as possible from different sources. We have also manually refined the collected data (e.g., vulnerable version ranges and the patch commits) to ensure the quality of our analyzed dataset.

Ethical Considerations. To avoid spamming the open-source community, we only select active developers and recent year CVEs, and all emails are sent by the accredited organization email address with clear information about the vulnerability and the developer's response to it. The developers have responded positively to our user surveys and also showed interests to our researches.

IX. Conclusion

In this study, we conduct a large-scale empirical investigation to comprehensively understand the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. Specifically, we analyzed 837 CVEs with 1,141 patches corresponding to 615 different upstream projects in the Maven ecosystem, which covers around 50 million of invocations from downstream projects to upstream libraries, what's more, we further conduct a user study with over 100 downstream developers. Our analysis is performed from multiple aspects and eventually, we distilled multiple interesting yet significant findings characterizing the potential security threats of upstream vulnerabilities. Such findings can not only benefit developers to understand the security threats in the Maven ecosystem, but also can guide future research in the field.

References

[1] M. Pittenger, "Open source security analysis: The state of open source security in commercial applications," *Black Duck Software, Tech. Rep*, 2016.

[2] Apache, *The Maven Repository*, Aug. 2022, https://mvnrepository.com/repos/central.

[3] Snyk, "The state of open source security," https://snyk.io/stateofossecurity/pdf/The%20State%20of%20Open%20Source.pdf, 2022, accessed: 2022-7.

[4] X. Zhan, L. Fan, S. Chen, F. Wu, 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.

[5] Apache, "Apache log4j2," https://github.com/apache/logging-log4j2, 2022, accessed: 2022-7.

[6] Google, "Understanding the impact of apache log4j vulnerability," https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html, 2022, accessed: 2022-7.

[7] OWASP, "Top 10 web application security risks," https://owasp.org/www-project-top-ten/, 2022, accessed: 2022-7.

[8] D. Check, https://owasp.org/www-project-dependency-check/, 2022, accessed: 2022-7.

[9] Snyk, https://snyk.io, 2022, accessed: 2022-7.

[10] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 559–563.

[11] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 138–147.

[12] K. Huang, B. Chen, L. Pan, S. Wu, and X. Peng, "Repfinder: finding replacements for missing apis in library update," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 266–278.

[13] W. Liu, B. Chen, X. Peng, Q. Sun, and W. Zhao, "Identifying change patterns of api misuses from code changes," *Science China Information Sciences*, vol. 64, no. 3, pp. 1–19, 2021.

[14] GitHub, *Top Programming Language*, Aug. 2022, https://githut.info.

[15] Dependabot, https://github.com/dependabot, 2022, accessed: 2022-7.

[16] 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.

[17] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[18] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 64–73.

[19] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracer: Finding patches for open source software vulnerabilities," *arXiv preprint arXiv:2112.02240*, 2021.

[20] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "Crossvul: a cross-language vulnerability dataset with commit data," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1565–1569.

[21] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[22] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022, pp. 621–633.

[23] T. V. V. Database, https://www.sourceclear.com/vulnerability-database, 2022, accessed: 2022-7.

[24] Aether, https://wiki.eclipse.org/Aether/What_Is_Aether, 2022, accessed: 2022-7.

[25] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, 2020.

[26] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[27] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[28] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 381–392.

[29] libraries.io, https://libraries.io/, 2022, accessed: 2022-7.

[30] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 518–529.

[31] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 507–511.

[32] Apache, *Top Projects of Maven Repository*, Aug. 2022, https://mvnrepository.com/popular.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[34] Y. Song, X. Zhang, and Y.-Z. Gong, "Infeasible path detection based on code pattern and backward symbolic execution," *Mathematical Problems in Engineering*, vol. 2020, 2020.

[35] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," *arXiv preprint arXiv:2002.01139*, 2020.

[36] 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, pp. 1–41, 2022.

[37] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[38] 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," *arXiv preprint arXiv:2201.03981*, 2022.

[39] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[40] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 446–457.

[41] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, 2017, pp. 84–94.

[42] 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.

[43] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, “On the use of dependabot security pull requests,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 254–265.

[44] 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*, 2017, pp. 2187–2200.

[45] 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. IEEE, 2015, pp. 109–118.

[46] 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)*. IEEE, 2020, pp. 35–45.

[47] W. Ma, L. Chen, X. Zhang, Y. Feng, Z. Xu, Z. Chen, Y. Zhou, and B. Xu, “Impact analysis of cross-project bugs on software ecosystems,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 100–111.

[48] H. Plate, S. E. Ponta, and A. Sabetta, “Impact assessment for vulnerabilities in open-source software libraries,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 411–420.

[49] 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.

[50] S. E. Ponta, H. Plate, and A. Sabetta, “Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[51] E. Iannone, D. Di Nucci, A. Sabetta, and A. De Lucia, “Toward automated exploit generation for known vulnerabilities in open-source libraries,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 396–400.