Empirical Software Engineering (2025) 30:20
https://doi.org/10.1007/s10664-024-10581-2

Understanding vulnerabilities in software supply chains

Yijun Shen[1] · Xiang Gao[1,2] · Hailong Sun[1,2] · Yu Guo[1]

Accepted: 21 October 2024

Abstract

Context Due to the dependency relations among software, vulnerabilities in *software supply chains* (SSC) may cause more serious security threats than independent software systems. This poses new challenges for ensuring software security including the spread of risks and the increase in maintenance costs.

Objective To address the challenges, there needs a deep understanding of how a vulnerability is in SSC in terms of vulnerability source, propagation, localization, and repair. However, no studies have been conducted specifically for this purpose.

Method To fill this gap, we provide an experience study of real-world vulnerability characteristics in the context of SSCs. Specifically, we examine the vulnerability source first and further study the fine-grained vulnerability propagation, localization, and repair of libraries and their corresponding client programs.

Results The key findings are summarized as follows: a) 99% of vulnerabilities in client programs are caused by their dependencies, and 81.26% of SSC vulnerabilities detected by package-level analysis are false positives; b) for vulnerability localization, the vulnerability database does not have enough information to help direct localization, but the vulnerability descriptions in the open-source vulnerability database provide much important information for indirect localization. c) client developers deal with vulnerable dependencies in many ways including upgrading dependencies, modifying client code, and deleting relevant code or vulnerable dependencies.

Conclusions Based on these observations, we make suggestions for future research in this direction: a) when testing important client programs, vulnerability detection tools should pay attention to both client code **and** the dependent libraries; b) localizing vulnerability based on vulnerability descriptions is not straightforward, hence a proper combination of program analysis and description analysis is expected to improve localization accuracy; c) there can be various strategies for dealing with vulnerable libraries, and automating the enforcement of those strategies will be expected.

Keywords Vulnerability · Software supply chain · Vulnerability source · Vulnerability propagation · Vulnerability localization · Vulnerability repair

Communicated by: Slinger Jansen

Extended author information available on the last page of the article

Published online: 06 November 2024

1 Introduction

Nowadays, more than 95% of enterprises actively or passively use open-source software (OSS) assets when developing key IT systems (Gartner 2021). Depending on third-party libraries, developers can save a lot of development costs and implement certain functionalities quickly. The relationship between upstream (library) and downstream (clients) components is analogous to the supply-and-demand relationship among products in business supply networks. Such dependencies among software components are also called *software supply chain*. SSCs provide great convenience for software development, but they also amplify the weaknesses of SSCs.

Because the influence of any vulnerable software component may propagate along the chains, the existence of SSCs exacerbates the challenge of maintaining software security, thus threatening the whole software system. In other words, if a widely-used OSS library has vulnerabilities, it may affect the security of the entire software ecosystem in the corresponding SSC. For instance, a *Remote Code Execution* vulnerability (CVE-2021-44228) was detected on Log4J, an Apache Java-based logging utility, in December 2021, which allows unauthorized attackers to execute any code remotely. According to the report from the Google Security Team, Log4J is directly or indirectly used by 170,000 projects, meaning that over 8% of all software packages are pedantically affected by this vulnerability. Therefore, understanding vulnerabilities in SSCs is of vital importance.

Unfortunately, addressing security issues in an SSC is challenging. First, in a complex SSC ecosystem, a vulnerability could occur in any component, which presents many challenges for vulnerability detection, localization, fixing, etc. Second, existing techniques are mainly designed for dealing with vulnerabilities in an independent software system without knowing the dependency relations between software packages. Complex software dependencies make the ecosystem difficult to analyze and maintain. To understand the vulnerability in SSC, in this paper, we aim to study their characteristics from four perspectives: vulnerability source, vulnerability propagation, vulnerability localization, and vulnerability fixing. The overview of our study approach is shown in Fig. 1. Our goal is to understand the properties of SSC vulnerabilities and provide insights on how to deal with them.

Vulnerability Source and Propagation Tracking the source and propagation of vulnerabilities is a crucial step in understanding vulnerabilities in SSC. Previous research has extensively explored vulnerability sources (Cox et al. 2015; Gkortzis et al. 2021; Wang et al.

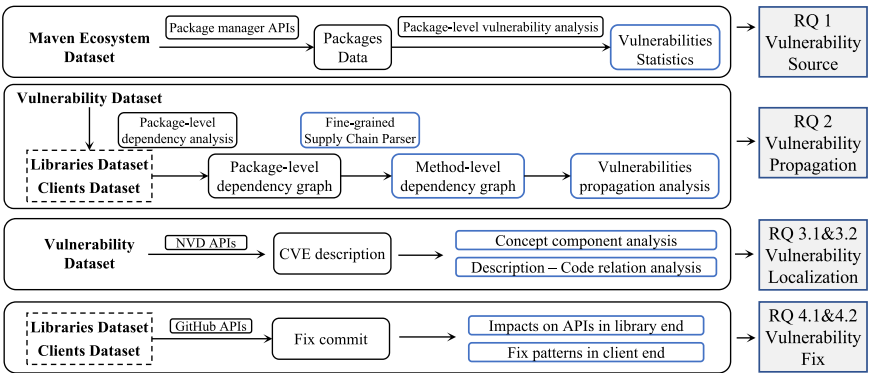

Fig. 1 The overview of our study approach

2020; Xia et al. 2013), vulnerability propagation (Liu et al. 2022; Wu et al. 2023), and the scope of vulnerability influence (Decan et al. 2018; Imtiaz et al. 2021; Soto-Valero et al. 2021; Zapata et al. 2018). However, existing works either perform coarse-grained package-level analysis, which can produce many false positives, or requires manual effort, which is not scalable.

To address these issues, we first discover the source of vulnerabilities in SSC by constructing a package-level dependency network and primarily investigate the following questions: What are the sources of vulnerabilities in SSC, and how many of them come from direct or indirect dependencies, as opposed to the client program itself?

Moreover, we implemented a fine-grained method-level dependency analysis to perform a more precise analysis. Based on our implemented tool, we also investigate the following questions: What are the sources of vulnerabilities in SSC, and how many of them come from direct or indirect dependencies, as opposed to the client program itself? Do the vulnerabilities in libraries truly propagate into client programs? How do client programs inherit vulnerabilities from third-party libraries? By exploring these questions, we will gain a better understanding of the challenges and opportunities associated with vulnerability propagation in complex SSC.

Vulnerability Localization Localizing the faulty code responsible for program vulnerabilities is a key step for assessing the scope of vulnerability influence, analyzing the root cause of vulnerabilities, and fixing vulnerabilities. Although fault localization (Wong et al. 2016) has been extensively studied, existing techniques typically depend on high-quality tests (Abreu et al. 2007) or bug reports (Youm et al. 2015) with complete bug information such as stack trace and exception type. However, for zero-day vulnerabilities, well-written bug reports and high-quality tests may not always be available. Within the SSC context, knowing the vulnerability location becomes more important. Specifically, knowing the vulnerable class or function that causes a security vulnerability can help determine the clients affected by the vulnerabilities, enabling affected client developers to take early actions to handle the security problem. Additionally, localizing the vulnerable code can help fix the bug quickly, minimizing security risks, particularly for widely-used libraries.

The Common Vulnerabilities and Exposures (CVE) description is the vulnerability information that is widely accessible by both library and client developers. However, compared to bug reports, CVE descriptions typically contain less information and just present high-level vulnerable behaviors. This makes it challenging to localize the exact location of vulnerable code based on CVE descriptions alone. Therefore, we studied several questions, including whether CVE descriptions can help localize vulnerability code, what kind of information in the description provides relevant concepts for localization, and whether the buggy code can be localized at the file level, method level, or statement level. Is there any other information that can be utilized to localize vulnerable code? Investigating these questions will enable us to understand the challenges and opportunities involved in localizing vulnerabilities in a complex SSC ecosystem.

Vulnerability Repair Previous research has thoroughly examined the vulnerability repair process in SSC and demonstrated that fixing these vulnerabilities can be a time-consuming task (Alfadel et al. 2021; Decan et al. 2018; Prana et al. 2021; Yasumatsu et al. 2019). However, no existing work has investigated the fixed patterns of vulnerabilities in SSC. Although fixing the vulnerabilities in an independent program has been studied extensively (Gao et al. 2020; Huang et al. 2019; Li and Paxson 2017; Perl et al. 2015; Xu et al. 2017), the fix patterns of vulnerabilities in SSC can differ significantly because they can be resolved at both the library and client ends. When developers fix a vulnerability in a library, they create a patch and release a new version of the library. Furthermore, client developers may also propose

solutions to mitigate the impact of the vulnerable library. In this paper, we investigate the patch patterns from both the library and client ends. Specifically, we explore the following questions: How do developers fix vulnerabilities residing in libraries? How do client maintainers fix vulnerabilities propagated from dependencies? When a library vulnerability is reported, how do client developers respond to address the vulnerability? Examining these questions will provide us with a better understanding of vulnerability fixes and inspire the design of automated vulnerability repair tools.

To answer these questions, we first collect the top 10000 Java artifacts sorted by GitHub Stars and all the vulnerabilities that exist in their historical versions to analyze the source of the vulnerabilities in SSC. Then we construct a dataset that includes 50 real-world libraries and their 61 corresponding vulnerability (CVE). For each library, we mine its dependents to construct a client dataset. Based on these datasets, we perform fine-grained analysis on the vulnerability propagation, the relations between CVE descriptions and bug locations, and the patch patterns.

In summary, we investigate the following research questions in this paper:

RQ1: What is the origin of vulnerabilities in complex SSC? The first RQ aims to explore the uniqueness of vulnerabilities in SSC. So in Section 4, we select the top 10000 Java artifacts sorted by GitHub Stars and collect all the vulnerabilities that exist in their historical versions. Based on the existing package-level vulnerability analysis tool, we divide these vulnerabilities into from-self and from-dependency categories and analyze their proportions and recent trends in changes respectively. The result shows vulnerabilities in SSC introduced by dependencies persist in open source projects and are mostly introduced by deeper indirect dependencies.

RQ2: Are these original vulnerabilities truly propagated to client programs? RQ2 aims to analyze the insufficiency of current vulnerability propagation analysis and propose a tool to solve the problem. Many vulnerabilities do not propagate down the supply chain to downstream projects; it's just that the existing analysis tools are not precise enough, leading to some false positive results. Therefore, in Section 5, we conducted an analysis of the 61 vulnerabilities associated with 50 distinct libraries, as well as the 1,280 clients that directly utilize these libraries, constructing their package-level propagation chains. Then proposed and implemented a method-level supply chain construction tool. Based on this tool, we conducted a method-level propagation chain analysis. The results showed that the package-level analysis indeed had a significant number of false positives, meaning that the corresponding vulnerabilities did not actually propagate to downstream projects.

RQ3.1: Do CVE descriptions include enough information for localizing vulnerabilities? RQ3.1 aims to analyze whether existing vulnerability databases can provide us with method-level vulnerability information. Since we have implemented the method-level vulnerability propagation analysis, as long as we can obtain the specific methods affected by the original vulnerabilities, we can determine all downstream vulnerabilities. In recent years, vulnerability databases have been collecting a large amount of related information and structuring it, so we attempt to observe whether existing vulnerability databases can meet the needs of our tool. We analyzed all 61 vulnerabilities and manually extracted the concepts of vulnerability type, component, vector, root cause, and impact in Section 6.1. The results show that most of the vulnerability database data does not directly provide the location of the relevant vulnerabilities.

RQ3.2: How can public vulnerability descriptions help vulnerability localization? RQ3.2 seeks to explore whether specific data about vulnerable methods can be indirectly obtained based on existing vulnerability data. Therefore, we analyzed the connection between the vulnerability descriptions and the vulnerable code in the 61 vulnerabilities in Section 6.2. The results indicate that most of the vulnerability locations can be identified manually, and further, automatic identification can also be achieved using Natural Language Processing methods.

RQ4.1: How are vulnerabilities fixed by a library? RQ4.1 aims to analyze how libraries modify the API to fix vulnerabilities. After identifying the source location and the specific functions, we attempt to explore whether any patterns can help with automatic vulnerability repair in the future. So we analyze the fix commits of 61 vulnerabilities in the vulnerability dataset to observe the changes on API respectively in Section 7.1. The results show a few fixes of libraries are incompatible, which may lead to incompatible problems when dependents apply these upgrades of libraries.

RQ4.2: How are vulnerabilities fixed by the affected client programs? RQ4.2 aims to analyze how vulnerabilities propagate along the supply chain are repaired. We conducted a manual analysis on identified 92 actual clients affected by these vulnerabilities through propagation analysis and observed the distinct strategies to address the vulnerabilities in Section 7.2. The experimental results show that there are indeed some commonalities in clients that can be applied to future automatic vulnerability repair techniques.

Overall, the contributions of this paper are summarized as follows:

❶ This work provides a study on real-world SSC vulnerability characteristics. Specifically, we examined the vulnerability source and propagation, vulnerability localization, and vulnerability repair.
❷ We propose an efficient finer-grained vulnerability analysis method to conduct method-level vulnerability analysis. The proposed tool could achieve more precise vulnerability propagation analysis.
❸ Based on our study, we have several findings on vulnerability propagation, localization, and fixes. Example findings include that 81.26% of vulnerability propagation detected by package-level analysis is false positive, 64% of CVEs exhibit connections between description text with vulnerable code directly or indirectly, client developers may solve vulnerable libraries by upgrading or directly deleting them, etc.
❹ Based on the key findings, we have several insights and make corresponding suggestions on how to handle the observed problems and present directions for future research.
❺ We have released our dataset and fine-grained vulnerability analysis tool [1], specifically for analyzing vulnerabilities in the context of SSCs.

2 Related Works

Security risks in the SSC have gradually increased and attracted increasing attention from researchers (Foundation 2020; Sonatype 2021; Synopsys 2022). Research on vulnerabilities in SSC has gradually become popular in the last few years (Enck and Williams 2022; Vu

[1] https://github.com/YijunShen/supply_chain_vul

et al. 2020). It helps the maintainers to understand the security of their software systems, especially the OSS that are developed in a collaborative way (Tan et al. 2022).

2.1 Tracking Vulnerabilities in SSC

To track the vulnerabilities in independent projects and complex SSC, existing works have investigated the vulnerabilities from the following aspects.

First, to understand vulnerabilities sources in SSC, existing works have proved that vulnerabilities could come from the latest but vulnerable dependencies (Gkortzis et al. 2021), outdated dependencies (Cox et al. 2015; Wang et al. 2020; Xia et al. 2013), nonstandard or illegal code reuse (Reid et al. 2022; Wyss et al. 2022). More specifically, Gkortzis et al. (2021) found there is a strong correlation between the number of dependencies and the number of vulnerabilities. They showed the usage of dependencies by invoking APIs between packages brought vulnerability into the SSC. Moreover, several studies analyzed library vulnerabilities in the NPM (Decan et al. 2018; Pfretzschner and ben Othmane 2017; Staicu et al. 2016; Zapata et al. 2018; Zimmermann et al. 2019), JavaScript (Lauinger et al. 2017), Maven (Prana et al. 2021), and PyPI (Alfadel et al. 2021) ecosystem. Those studies prove that security risks exist in all dependency networks. Nevertheless, those studies showed the vulnerabilities can be inherited from upstream dependencies, but lack of deep and qualitative analysis of different vulnerability sources (direct dependency, indirect dependency, or client program itself).

To understand the vulnerability propagation in SSC, Liu et al. (2022) conducted a large-scale research on how vulnerabilities caused security risks along the NPM dependency tree, and found vulnerabilities propagated from libraries still widely exist in clients, even for the latest versions. Zapata et al. (2018) manually checked how high-severity vulnerabilities are propagated to clients at the function level, and found out the defects were not actually propagated to most of the clients. However, those works either only focus on the NPM ecosystem (Liu et al. 2022) or require manual analysis (Zapata et al. 2018). Different from those works, our study is mainly focused on the Maven ecosystem (can also be generalized to other ecosystems) and does not require manual analysis, hence can be scaled to a large dataset. Additionally, although Wu et al. (2023) analyzed vulnerability propagation on Java programs via fine-grained analysis, they did not investigate vulnerability sources, localization, etc. Moreover, the fine-grained analysis conducted by Wu et al. (2023) is not precise enough because they ignored the complex object-oriented programming features when building call graphs.

With the development of software component analysis (SCA), many software use the automated package manager to manage third-party dependencies, which provides convenience to analyze the vulnerability scope based on package-level dependency relations. Such techniques have become mature (Decan et al. 2018), and some products like Dependabot [2] and dependency-check [3] have been developed. However, it has been proved that not all libraries are actually used by the clients, especially for indirect dependencies (Soto-Valero et al. 2021). Most of the security warnings produced by those tools are false-positive because the vulnerable function in the warned NPM packages is not used by the client (Zapata et al. 2018). Some studies (Dann et al. 2022; Imtiaz et al. 2021) also investigated current industry-leading software component analysis (SCA) tools, and showed that the package-level SCA is hard to detect all kinds of vulnerabilities and their accuracy is limited. In contrast, our study focuses

[2] https://github.com/dependabot

[3] https://owasp.org/www-project-dependency-check/

on the finer-grained function-level analysis to determine whether a vulnerability is actually propagated into client programs.

2.2 Vulnerability Localization

Vulnerability localization determines a set of suspicious buggy statements that cause the vulnerability. In the context of the software supply chain, finding the root node of a long vulnerability propagation chain is necessary, which can also benefit carrying out subsequent risk mitigation and elimination. Traditional bug localization to some extent can be used to do vulnerability localization, including Spectrum-based and Report-based localization. Besides, Pattern-based and Code Similarity-based methods are also being conducted extensively.

Spectrum-based localization (Reps et al. 1997; Xie et al. 2013) is widely used which find candidate buggy statements by analyzing the execution trace of passing and failing tests. However, according to a recent study (Liu et al. 2019), the conclusions drawn from this method are based on several strong assumptions, only a subset of bugs/vulnerabilities can be correctly localized by Spectrum-based techniques. Researchers also found that there is no spectrum-based vulnerability method that achieves optimal localization results on all evaluation programs Debroy and Wong (2013), Santelices et al. (2009). To address this issue, Xuan and Monperrus (2014) utilized machine learning to integrate multiple spectra methods to improve localization effects. However, the trained model is only limited to the same type of defect and it is challenging to establish a cross-project learning model. Moreover, spectrum-based localization techniques require high-quality test cases, which are not always available.

Report-based localization (Wijayasekara et al. 2014) is an automatic defect identification method by matching vulnerable code to the vulnerability report. Open-source software often uses defect tracking systems (such as Bugzilla, JIRA, or GitHub's Issue feature) to record defects. Previous researches focus on using information retrieval models to improve the accuracy of defect localization, including BugLocator (Zhou et al. 2012) which relies on an improved Vector Space Model, BRTracer (Wong et al. 2014) and Lobster (Moreno et al. 2014) which utilize the stack trace information, BLUiR (Saha et al. 2013) which uses the code structure information, and genetic algorithms (Almhana et al. 2016; Wang et al. 2014). These methods can also be applied to vulnerabilities as well. Unfortunately, the reports are often only sent to the security organization, remove the public record of the vulnerability, and solve the problem in the original libraries. As for those maintainers of client projects, they always objectively wait for the launch of a patch to upgrade the dependencies.

Pattern-based methods primarily involve the induction and summarization of patterns for a certain type of vulnerability, thereby facilitating the identification and localization of the code. Viega et al. (2000) manually summarize a series of vulnerability patterns in C/C++ languages to locate the vulnerability automatically. Yamaguchi et al. (2013) extract "missing checks" type defect patterns for vulnerabilities such as user permission check omission, buffer overflow, and integer overflow. Grieco et al. (2016) add features defined by human experts to generate vulnerability patterns automatically. This kind of method is only used for one or several types of vulnerabilities, which can hardly apply to wide datasets. Code similarity-based methods divide a program into a set of code fragments and represent each code fragment as a vector, then calculate the similarity between the vulnerability code and the program source code to find the most likely vulnerability location. Jang et al. (2012) proposed a matching method based on hash values, while Pham et al. (2010) suggested similarity matching based on abstract syntax trees. Additionally, Sejfia and Schäfer (2022), a machine-learning-based approach, is proposed to learn relevant features of vulnerability

from a defective code database, to identify vulnerable code in the source code. However, these techniques can only be used for vulnerabilities with known types, and the accuracy is limited.

Existing techniques mainly focus on localizing buggy statements in independent projects. In contrast, our study discusses the possibility of localizing the vulnerability code in the complex SSC. We study whether the vulnerable code can be identified using the description in the vulnerability database, which can be easily accessed by ordinary developers of a client.

2.3 Vulnerability Repair

Regarding vulnerability repair in SSC, existing research (Decan et al. 2018; Prana et al. 2021; Yasumatsu et al. 2019) has mainly focused on investigating the time and process of releasing patches. These studies found that many vulnerabilities in libraries remain unpatched in client programs for an extended period, often taking at least three to fifteen months to be fixed. Furthermore, automated program repair, a technique designed to generate patches automatically (Goues et al. 2019), has gained increasing attention in recent years. The potential for automatically fixing vulnerabilities has also been widely explored (Gao et al. 2020; Huang et al. 2019). However, existing vulnerability repair methods primarily focus on fixing independent programs. Our study aims to examine the pattern of patches in SSC and provide inspiration for automatic vulnerability repair methods.

3 Dataset

In this section, we present the process of constructing datasets for our study, as shown in Fig. 2. To understand the vulnerabilities in SSC, our research utilizes data from the Maven ecosystem. On one hand, Maven is among the pioneering ecosystems that employ a package manager for dependency management in software programs, thereby possessing a more comprehensive Software Bill of Materials (SBOM). On the other hand, in comparison to other programming languages, Java exhibits more intricate rules of code invocation. This implies that our research on the Java ecosystem could be more readily applicable to other languages.

Maven Ecosystem Dataset To reveal the severity of vulnerabilities in SSC in the Maven ecosystem, we collect the top 10,000 Java packages sorted by GitHub Stars from Libraries.io

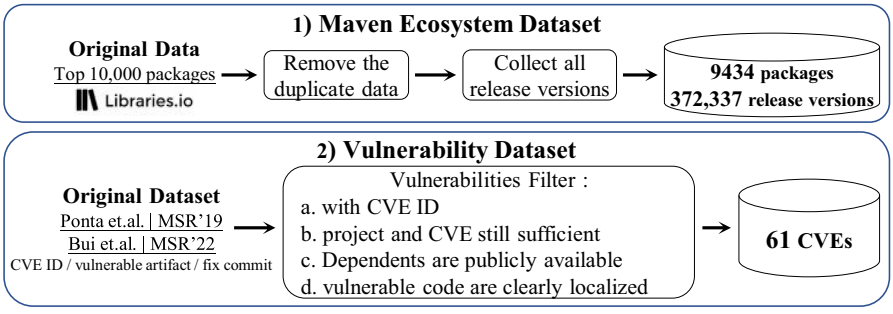

Fig. 2 Dataset construction process

[4]. We have employed the platform's provided Application Programming Interface (API) to procure information regarding the names of open-source projects, their GitHub star counts, their release versions, and the details of all their dependencies. The quantity of data we are able to procure is constrained by the API limitations of the platform, which stipulate a maximum of 10,000 packages can be accessed. The range of GitHub stars for these packages spans from 9,240 to 73,297, which is indicative of the majority of mainstream Java packages. After removing the duplicate data, we obtain 9434 packages with all their history versions (372,337 release versions in total).

Vulnerability Dataset To understand how the vulnerability is propagated, localized, and fixed, we further deeply analyze a set of vulnerabilities. Analyzing all the vulnerabilities in Maven Ecosystem is impossible, hence we select the latest vulnerabilities from two existing vulnerability datasets: (1) the vulnerability dataset collected by Ponta et al. (2019) and (2) the Vul4j dataset collected by Bui et al. (2022). Both datasets provide the vulnerability IDs, URLs of the repositories of vulnerable projects, the IDs of fix commits, and the class information. We use those two dataset because (1) they contain a large number of representative vulnerabilities obtained from the National Vulnerability Database (NVD), where the first dataset includes 624 publicly disclosed Java vulnerabilities across 205 distinct open-source Java projects, and the second dataset includes 79 vulnerabilities from 51 open-source Java projects; (2) they also provides fixing commits for each vulnerability, which is useful for localization and patch analysis; and (3) the first dataset was constructed in 2019, and we do not use the most recently disclosed vulnerabilities because those vulnerabilities and the influenced clients may have not been patched yet.

Over this dataset, we construct the method-level vulnerability dataset according to the following criteria:

- **C1. The vulnerability has been assigned with a CVE ID:** The original datasets collect data from different vulnerability databases. The vulnerabilities from the different databases have different IDs and descriptions, which may mislead our analysis. Hence, we just retain the vulnerabilities with CVE IDs.
- **C2. The vulnerability has been detected in the past 6 years, and the CVE ID and the libraries are still active:** Some CVE IDs have been labeled as "rejected", while some libraries have been abandoned in the past few years. For instance, the project 'tomcat85' is no longer a repository name, and it has been merged into 'tomcat' as a branch.W To ensure that information is still valid, we choose vulnerabilities detected from 2017 to 2023, and remove the invalid CVE and libraries.
- **C3. Dependents of projects are publicly available:** To analyze the problem along the supply chain, we need to find which clients use these vulnerable libraries as dependencies. We use 'Project Dependents' [5], an API provided by Libraries.io to collect dependent information.
- **C4. The vulnerable method can be clearly localized:** To analyze the vulnerability localization, we identify the vulnerable code according to the changed code in a fix commit. To keep the localization task simple, we only keep the vulnerabilities whose fix commits only change one *.java* file.

Eventually, we obtaine 61 CVEs from 50 projects with their CVE ID and fix commit url, and used them as our vulnerability dataset. Furthermore, to conduct a fine-grained vulnerability analysis, we performed a manual examination of each vulnerability, thereby obtaining

[4] https://libraries.io is one of the largest open-source data platform which monitors open source packages across 32 different package managers.

[5] https://libraries.io/api#project-dependents

corresponding information such as CWE IDs, projects, packages, files, and methods. The construct process is as follows:

- *CWE ID:* Using the API provided by NVD to search each CVE ID and parse the information of CWE ID in the return value in JSON format. If there is more than one CWE source from different organizations(such as NIST or Red Hat), we choose the information from NIST.
- *project:* Parsing the fix commit url and directly using the owner and artifact information in it.
- *file and method:* We identify the location of the vulnerable file and method by manually analyzing the different information in the fix commit.
- *package:* To confirm the affected package, we initially proceed to the parent directory of the vulnerability file to locate the pom.xml file and utilize the artifact and group information contained therein. The pom.xml file is the SBOM for the Maven ecosystem, documenting not only the metadata of the package itself but also the information of all its direct dependencies.
  If the project does not contain a pom.xml file, it necessitates the acquisition of three additional pieces of information: the vulnerable java file, the Common Platform Enumeration(CPE) description, and the package set: At the top of each vulnerable Java file, we are able to parse information pertaining to the package in which it resides; Concurrently, the Known Affected Software Configurations information provided by the NVD includes the affected CPEs and also furnishes details regarding the vulnerable packages; Finally, the complete set of packages included in the current project, as parsed by GitHub, can be obtained from the Insight-dependency graph-dependents feature provided by the GitHub platform. By integrating these three sets of information, we can ascertain the package in which the vulnerable code resides.

These details provide an accurate description of the specific methods where the vulnerabilities reside, facilitating our experiments in vulnerability propagation, localization, and repair

Dataset Statistics To ensure the representativeness of this vulnerability dataset, we conducted an analysis of the packages to which the vulnerabilities belong, the types of vulnerabilities, and the coverage rate of the OWASP Top 25 CWE (2021).

Examination of Table 1 reveals that vulnerabilities are relatively evenly distributed across each package. Furthermore, The highest-ranked package is com.google.guava:guava, which

Table 1 Package statistics in the vulnerability dataset

Package name	Vuls	Rank in MvnRepo
org.jenkins-ci.main:jenkins-core	3	618
org.apache.tomcat:tomcat-catalina	3	821
org.apache.commons:commons-compress	3	146
org.springframework.security:spring-security-core	2	209
org.apache.tomcat:tomcat-coyote	2	2042
com.inversoft:prime-jwt	2	71848
org.apache.tomcat.embed:tomcat-embed-core	2	430
org.apache.xmlgraphics:batik-dom	2	2195
42 other packages	1(mean)	4666(median)

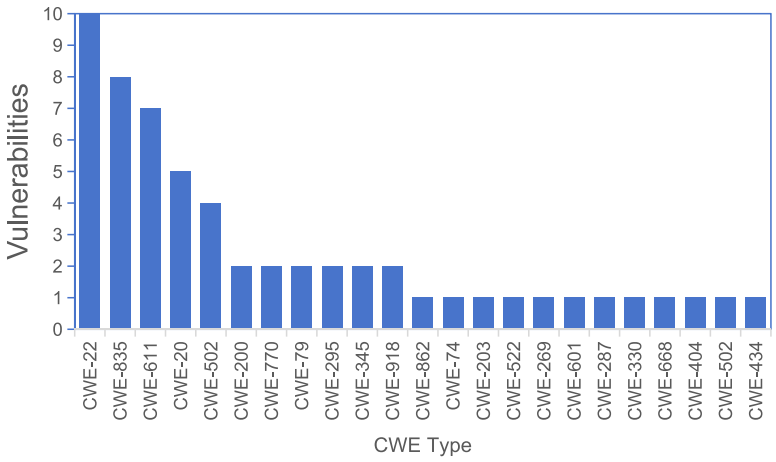

Fig. 3 Distribution of vulnerabilities by CWE categories

is positioned at 5th place, while the lowest-ranked package is ro.pippo:pippo-jaxb, ranked at 292,770th place. Packages within the top 10,000 constitute 66% of the total number, and those within the top 100,000 account for 90% of the entire collection. Considering that the Maven Central Repository currently encompasses 36.5 million packages, the packages with vulnerabilities in the dataset are all relatively popular software packages at present.

As depicted in Fig. 3, the 57 vulnerabilities within our dataset are categorized under xxx types of Common Weakness Enumeration (CWE), with an additional 4 vulnerabilities that have yet to be classified. Among these, the vulnerability type designated as CWE-22 is the most prevalent, accounting for 10 instances, which constitutes 36% of the total count. According to *2023 CWE Top 25 Most Dangerous Software Weaknesses*(CWE Top 25) [6], 27 of the 61 vulnerabilities identified are classified within the scope of the CWE Top 25, which encompasses 9 distinct Top 25 CWEs.

The dataset and statistics are publicly available in a GitHub repository. [7]

4 Vulnerability Source

Vulnerabilities seriously affect the security of software in the SSC, hence, for a given program, we first study the source of vulnerabilities. Given a vulnerability in the library program, previous studies have evaluated whether it can be inherited by a client program with the goal of understanding the scope of vulnerability influence and propagation path of security risks (Decan et al. 2018; Prana et al. 2021). In the reverse direction, given a set of vulnerabilities that affect the safety of a client, the source of vulnerabilities has not been well analyzed. More specifically, given a vulnerable program P, how many of the vulnerabilities are caused by the code written by P's developers (i.e., artifact itself), how many of them are inherited from P's direct dependencies or indirect dependencies?

Understanding the vulnerability source has two main usage scenarios. First, for client program developers, understanding vulnerability sources could help take quick actions to

[6] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[7] https://github.com/YijunShen/supply_chain_vul

Table 2 Source of vulnerability

	pkgs	#vul	from slf.	from dep.	direct dep.	indirect dep.
All versions	9434	382,962	299	382,663	–	–
Latest version	7666	166,577	91	166,486	29,927	136,559

eliminate the effects. If vulnerabilities are from the artifact itself, developers can fix them by themselves. If they are from direct dependencies, developers can delete/replace the direct dependencies or bypass the vulnerabilities by changing the way of using direct dependencies. However, if a vulnerability is from the indirect dependency, it may not only propagate to more than one direct dependency, which makes it hard to fix all of them. Therefore, developers can only wait for the fix update of relative direct dependencies. Second, for security researchers, understanding the source of vulnerability could provide guidance on vulnerability scope analysis and repair strategies. Moreover, understanding vulnerability sources could help security researchers design testing strategies for detecting vulnerability in complex SSC.

Study Setup To quantitatively analyze the vulnerability sources, we examined the data of the *Maven Ecosystem* dataset shown Section 3. This dataset contains 9434 widely-used artifacts and 372,337 released versions. The dependency and vulnerability information are obtained from Open Source Insight [8], an industrial OSS statistical analysis platform developed by Google. Open Source Insight maintains a dependency network of the whole Maven Central Repository. Besides, it also establishes a connection to GitHub Advisory Database, which is an open-source vulnerability database hosted by GitHub. The analysis result of Open Source Insight provides the dependency and the vulnerability information of the artifact.

To find how many vulnerabilities are caused by the artifact itself or the dependency, we first query Open Source Insight to get all vulnerabilities of each version, and remove duplicated data in all historical versions of each artifact. This step collects all the vulnerabilities induced by code changes in the life cycle to date. Second, we query Open Source Insight to get the dependency information of each version and the corresponding vulnerability information. By matching the dependency name and the product name that is affected by a vulnerability, we identify the vulnerability source of each artifact.

Results and Findings Table 2 shows the results of vulnerability sources.

In 372,337 historical versions of these 9434 packages, 299 vulnerabilities exist in the artifact, while 382,663 vulnerabilities exist in the dependency. Vulnerabilities from the dependency account for 99.92%, which means most of the risks come from third-party libraries. Package 'org.springframework.boot: spring-boot-autoconfigure' has the largest number of vulnerabilities (380), and all of them are inherited from its libraries. Apparently, most of the vulnerabilities occur in the libraries of the client. This result proved that when importing a third-party library to reduce development costs, people should pay more attention to the dependencies that may bring a large number of vulnerabilities. Moreover, the more versions a package releases, the better this package is maintained. Among the 9434 packages, 1448 packages have more than 100 versions. In those well-maintained packages (¿100 versions), 1195 of them have vulnerabilities in the history, with 0.02% of vulnerabilities being from the client code itself, while 99.98% are from the libraries. As expected, well-maintained artifacts have much fewer vulnerabilities from itself than unwell-maintained artifacts (0.02% versus 0.08%).

[8] https://deps.dev

To discover whether the vulnerabilities have been fixed now, we further analyze the vulnerability information in the latest version of each artifact. Among all the 9434 packages, we observe that 7666(81.25%) artifacts still have unfixed vulnerabilities. Specifically, there are a total of 166,577 vulnerabilities that have not been fixed, and 91(0.05%) of them are from the artifact, while the rest 99.95% are from the dependency. For instance, the latest version `1.5.3.1.RELEASE` of package 'io.kwy.boot:spring-boot-autoconfigure' has the largest number of unfixed vulnerabilities (224), and all of them are from libraries. This is typically 'assembled software' by leveraging the open-source mode. It has 166 dependencies, while two-thirds of them are indirect dependencies. Using a large number of third-party libraries and lack of maintenance lead to a large number of vulnerabilities. Fortunately, no client depends on this outdated artifact at present. Among 1448 well-maintained packages, 849 (58.63%) of them have unfixed vulnerabilities. Although maintaining well helps reduce the vulnerabilities, more than half of artifacts are still at risk.

To learn which level of the dependency vulnerabilities come from, we further analyze the dependency information in the latest version of each artifact. Evaluation results show that 29,927 vulnerabilities occurred in 130,718 direct dependencies, while 136,559 vulnerabilities occurred in 463,871 indirect dependencies. Much more vulnerabilities are from indirect dependencies than direct dependencies. Vulnerabilities in both direct and indirect dependencies could significantly affect the security of the project.

To further understand the vulnerabilities source of huge, well-maintained artifacts like 'org.neo4j:neo4j', we selected the software with the highest number of minor versions in the most recent major release, and manually analyzed the trends in the quantities of direct and transitive dependencies, as well as the number of associated vulnerabilities, within each software artifact. Table 3 shows selected artifacts and the number of versions in the most recent major release.

Subsequently, we employed line charts to illustrate the corresponding trends. The x-axis presents the versions of each artifact, while the y-axis presents the quantity of dependencies and corresponding vulnerabilities. Meanwhile, the dashed blue line represents the trend in the number of direct dependencies, while the dashed red line indicates the trend in the number of transitive dependencies. Concurrently, the solid blue line delineates the trend in the number of vulnerabilities introduced by direct dependencies, and the solid red line corresponds to the trend in the number of vulnerabilities introduced by transitive dependencies.

Table 3 Number of versions of popular artifacts

No.	Package name	Versions
1	io.dropwizard:dropwizard-core	2.0.0 ~ 2.1.12 (58)
2	org.apereo.cas:cas-server-core	6.0.0 ~ 6.6.15.2 (85)
3	io.quarkus:quarkus-core	2.0.0.Final ~ 2.16.12.Final (85)
4	org.redisson:redisson	3.0.0 ~ 3.31.0 (131)
5	org.hibernate:hibernate-core	5.0.0.Final ~ 5.6.15.Final (138)
6	org.neo4j:neo4j	4.0.0 ~ 4.4.34 (105)
7	org.elasticsearch:elasticsearch	7.0.0 ~ 7.17.22(71)
8	org.springframework.boot:spring-boot	2.0.0.RELEASE ~ 2.7.18(119)
9	org.springframework:spring-core	5.0.0.RELEASE ~ 5.3.37(103)
10	cn.hutool:hutool-core	5.0.0 ~ 5.8.28 (109)

Table 4 Trend of dependencies and vulnerabilities in popular artifacts

No.	[Pearson,pvalue](d)	Relationship(d)	[Pearson,pvalue](ind)	Relationship(ind))
1	[0.20, 0.14]	Non-Positive	[0.84, 2.2e-16]	Positive
2	[-0.79, 2.7e-19]	Non-Positive	[0.63, 1.4e-10]	Positive
3	nan	Non-Positive	[0.97, 1.7e-55]	Positive
4	[-0.65, 5.6e-16]	Non-Positive	[0.50, 8.3e-10]	Positive
5	[-0.77, 1.9e-28]	Non-Positive	nan	Non-Positive
6	[-0.19, 0.05]	Non-Positive	[0.03, 0.74]	Non-Positive
7	nan	Non-Positive	[-0.68, 3.9e-11]	Non-Positive
8	nan	Non-Positive	nan	Non-Positive
9	nan	–	nan	–
10	nan	–	nan	–

In these 10 artifacts, we observed 3 distinct types of relationships: positive correlation, non-positive correlation, and well-maintained("-" is used to represent in Table 4). Meanwhile, we used the Pearson correlation coefficient to verify the connection between the number of dependencies and vulnerabilities. The results are listed in Table 4. The second and third columns represent the Pearson correlation result and the p-value between the number of direct dependencies and corresponding vulnerabilities, while the fourth and fifth columns are the results between the number of indirect dependencies and corresponding vulnerabilities.

Positive correlation implies that the general trend of the number of direct dependencies and the number of vulnerabilities they introduce, or the number of transitive dependencies and the number of vulnerabilities they introduce, increases and decreases in tandem. As shown in Fig. 4, although the blue dashed and solid lines do not have a clear correlation, the red dashed lines, corresponding to the solid lines of the same color, generally ascend or descend concurrently. As shown in Table 4, the average correlation coefficient of the first

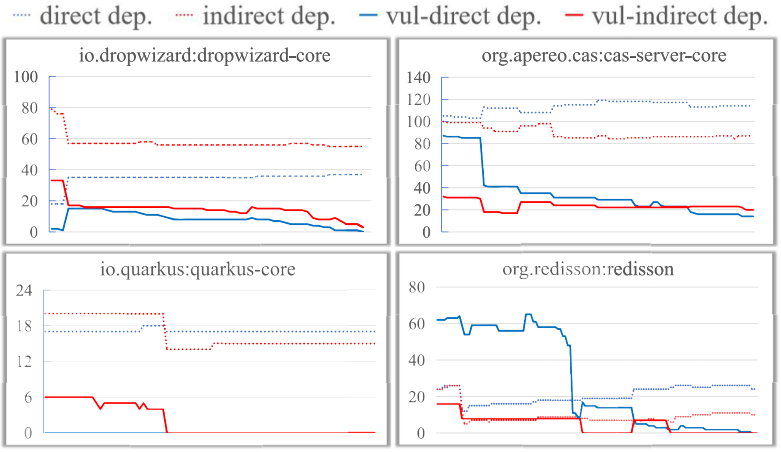

Fig. 4 The quantity of dependencies exhibits a positive correlation with the corresponding number of vulnerabilities

four artifacts is 0.735, which also proves the synchronized growth relationship between the number of indirect dependencies and corresponding vulnerabilities.

Non-positive correlation implies that there is no inherent association between the number of direct dependencies and the number of vulnerabilities they introduce, or between the number of transitive dependencies and the number of vulnerabilities they introduce, as shown in Fig. 5.

Well-maintained("-") indicates that the maintainers of the artifact are exceptionally cautious, ensuring that the dependencies they incorporate are either free from vulnerabilities or that any existing vulnerabilities are swiftly rectified. At the same time, they often exist as the most upstream part of the supply chain, with the number of third-party libraries used being in the single digits, which also prevents the introduction of most vulnerabilities.

According to those figures, we have the following 3 main observations: 1) In these artifacts that belong to the same major version, the overall trend in the number of vulnerabilities is downward (i.e., the solid line portions in each chart). This implies that as time progresses, developers are increasingly inclined to address and remediate vulnerabilities. 2) The positively correlated line charts demonstrate that as the software incorporates an increasing number of dependencies, a corresponding rise in the introduction of vulnerabilities occurs. 3) Even after vulnerabilities within direct dependencies have been fully remediated, those within transitive dependencies tend to persist for a more extended period (i.e., the solid red lines in each chart).

To verify the hypothesis based on observation 2), we conduct a Spearman's rank correlation test (Dodge 2008) to find the relationship between the number of direct or indirect dependencies and the number of vulnerabilities that are directly or indirectly induced. The results indicate there is a positive correlation between the number of indirect dependencies and the number of vulnerabilities that are indirectly induced(with average value: $\rho = 0.433$, p-value = 0.0017). On the contrary, the result show there is no correlation between the number of direct dependencies and the number of vulnerabilities that are directly induced(with average value: $\rho = -0.462$, p-value = 0.0098). A plausible explanation may be attributed to the well-maintained packages, which, despite an increase in dependencies over time, promptly address and rectify vulnerabilities within their controllable direct dependencies.

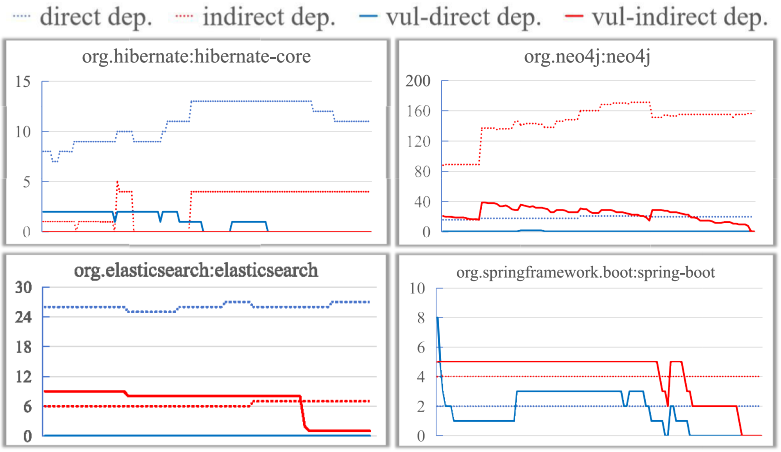

Fig. 5 The quantity of dependencies exhibits no correlation with the corresponding number of vulnerabilities

The result proves that paying attention on maintenance can help reduce the risks propagated from the dependencies, but it is hard to fix all vulnerabilities entangled with the complex dependency network.

> **Key Finding for RQ1:** Most vulnerabilities in the historical versions of an artifact are caused by its dependencies, especially indirect dependencies. Even the latest versions still have a large number of vulnerabilities, with nearly all of them from the third-party library.

5 Vulnerability Propagation

In this section, we track the vulnerability in SSC in a finer-grained manner. Although Library.io, Google Insight, or other package management systems have presented the scope of influence of vulnerabilities, they are based on package-level dependency analysis. It has been proved that plenty of bloated dependencies, i.e., the dependencies that are included in dependent list but are not actually used, exist in the SSC. Hence, the package-level techniques cannot precisely track the propagation of the vulnerabilities. To solve this problem, we propose an efficient method-level dependency analysis technique and then perform fine-grained vulnerability propagation analysis. Understanding the precise vulnerability propagation can help us evaluate the risk of client programs.

Study setup. The overview of our methods that conduct fine-grained propagation analysis is shown in Fig. 6

To do so, we first collect a set of client programs that depend on the vulnerable libraries in the vulnerability dataset from Section 3. The client programs are mined by following steps:

– **S1. Find out which versions of which packages of a library project are vulnerable.** A project may have more than one package, while a package often has more than one release version. Apparently, not all of them are affected by the vulnerable code. We first figure out the vulnerable package and its corresponding version.

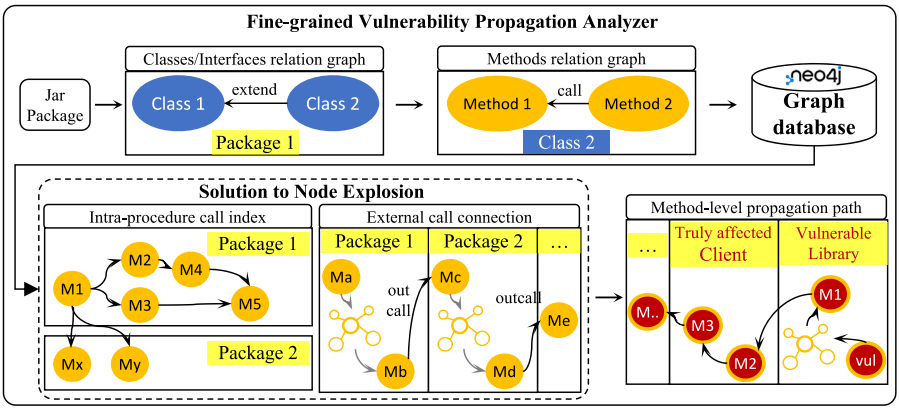

Fig. 6 The overview of fine-grained propagation analysis

- **S2. Collect clients that depend on the obtained vulnerable package with a certain version.** Based on the *Dependents* function of Open Source Insight, we collect the dependents of the latest vulnerable version of each library. (In case the dataset is too large, we only consider the top 300 items in the dependent list.)
- **S3. Filter the client programs without jar packages.** As shown below, our fine-grained analysis tool takes jar packages as inputs, hence, we filter the clients which cannot get the jar package.

Eventually, we mine 1280 clients satisfying the above criteria. All the packages and their relevant dependency information are stored in graph database Neo4j [9]. By querying the nodes and edges propagated from the vulnerable package node, we can construct the propagation path at the package level.

Our fine-grained dependency analysis tool is implemented based on java-call-graph [10], a call-graph generator applied to a jar file. The method-level dependency graph is constructed as below:

- *Analyze the relationships of class inheritance and interface implementation.* Parsing the parent class inherited by the subclass and the interface implemented by the class during declaration, we construct the class dependency graph in the package.
- *Exam the relationships of method calls, overrides, and overloads* By utilizing the tool java-call-graph, we can obtain the augmented method call graph. For cases of multiple inheritance, we use the class dependency graph mentioned above to determine which method it actually inherits. Besides, the method calls are categorized into internal and external, which represent the calling and being called methods within the same package or between two packages.
- *Store nodes and edges in the graph database, and query the propagation path* After storing all the method nodes and augmented calling edges into Neo4J, we query the graph database to construct the method-level propagation paths.

However, due to the fact that the number of method nodes often grows by more than two orders of magnitude compared to the number of package nodes, coupled with the existence of cyclic calls between methods. Even the method propagation paths between just two packages can lead to memory overflow and program crashes. To address this problem, we design an optimized strategy for constructing fine-grained propagation paths. Specifically, the steps for constructing the vulnerability propagation path using this optimized method are as follows:

- *Establish an intra-procedure call index for each package.* For a given method node m, we first determine the methods that are reachable from m along a call path within the same package, and store the list of reachable methods as an attribute of node m.
- *Identify all nodes involved in external calls between the vulnerable package and downstream client packages.* Querying the nodes involved in the vulnerability package for external calls from all method nodes in the client package.
- *Construct the propagation path of the vulnerable method.* Querying the path from the vulnerable method node to the external call nodes. If a path exists, further concatenate the client package to its downstream dependencies (i.e., from the first-level dependency to the second-level dependency), and so on until the most downstream package is reached.

Then, we translate the vulnerability reachability analysis into a database query problem, i.e., query the paths from the vulnerable method to any of the client methods. Subsequently,

[9] https://neo4j.com

[10] https://github.com/gousiosg/java-callgraph

we conducted package level and method level vulnerability propagation analysis on all vulnerabilities and their direct dependencies based on the above methods. Based on the above fine-grained analysis, we answer the following question: *how many client programs are really affected by the library vulnerabilities*.

Results and Findings For the 50 libraries and 1280 clients, our tool constructs a vulnerability propagation graph for 37 vulnerable libraries and the related 492 clients. This graph consists of 4,905 vulnerable method nodes and 5,408 vulnerable function calls. Meanwhile, 75,105 vulnerable function invocation paths are detected according to the graph. We found that only 40.54% of vulnerable methods are really used by one or more client programs, while the others are never used by any client program, meaning that those vulnerabilities have not been propagated to downstream programs. For instance, CVE-2018-1324 is a vulnerability caused by an incorrect field parser in the function *parseCentralDirectory Format*, which could cause an infinite loop. This vulnerability affects package *org.apache.commons:commons-compress:1.15*, and malicious attackers could use this vulnerability to conduct denial of service attacks. By investigating the 94 client programs of this package, we did not find any client program actually invokes (both directly and indirectly) this vulnerable function.

Moreover, we further analyze the number of client programs that are actually affected by the vulnerabilities. Among 491 client programs, 92 (18.74%) of them really invoke the vulnerable methods, while the remaining 399 (81.26%) clients are not affected by the vulnerabilities. For instance, package *commons-io:commons-io:2.6* is affected by CVE-2021-29425, which is caused by the lack of necessary checks in the function *getPrefixLength*. This vulnerability can allow attackers to access the directories without permission. This vulnerable package is used by *org.openksavi.sponge:sponge-core:1.14.0*, which invokes function *concat* from package *commons-io* and then *concat* invokes vulnerable function *getPrefixLength*. Therefore, *sponge-core* is really affected by this vulnerability. Detecting the indirect vulnerable function invocations requires precise call graph construction.

Additionally, to demonstrate the effectiveness of our method, we choose the vulnerable library package, *org.codehaus.plexus: plexus-archiver: 3.5*, with the most available direct dependents to construct a complete vulnerability propagation graph. The package-level propagation graph indicates that vulnerability CVE-2018-1002200 in this library propagates to 41 direct dependents and 141 related indirect dependents up to 9 layers deep. However, according to the method-level propagation graph generated by our tool, only 16 direct dependents and 2 indirect dependents are actually affected by the vulnerable method *org.codehaus.plexus.archiver.AbstractUnArchiver:extractFile*. The propagation path is shown as Fig. 7.

Based on the above results, most of the vulnerabilities are actually not propagated out of the library programs, and most client programs are not affected by the library vulnerabilities. In other words, the package-level analysis tools, such as Google Insight, DependaBot (Depend-

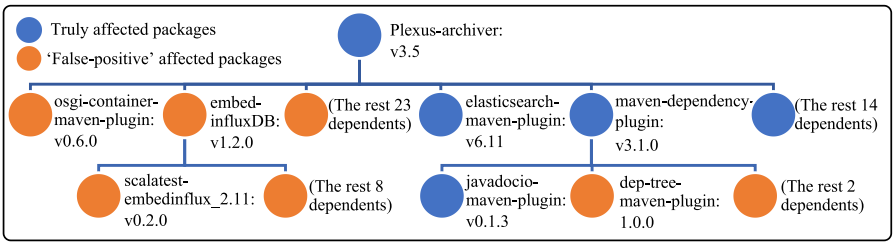

Fig. 7 The propagation path of vulnerability CVE-2018-1002200

abot 2023), and Dependency-Check (Dependency-Check 2023), produce a lot of false positives. False positives will result in two main problems: (1) increase the burden of developers in maintaining client programs, and (2) cause developers to lose trust in the package-level analysis tools. Hence, we argue that fine-grained analysis is necessary to generate precise security warnings (report the vulnerabilities that actually affect the client programs) and help developers increase the security of their programs.

As we mentioned above, fine-grained analysis is hard to scale to large programs. Moreover, our tool also has some shortcomings, which do not consider calls through reflection and dynamic proxies, which may result in incomplete call graphs. Even if a vulnerable function is eventually invoked along the call graph, it does not mean the vulnerability affects the client program. This is mainly because triggering a vulnerability requires certain conditions, e.g., the divisor must be set as zero to trigger a divide-by-zero bug, which may not be satisfiable along the path from the client program to the vulnerable function. Checking whether the vulnerable condition can be satisfied requires dynamic information, which is hard to obtain and not scalable. In general, fine-grained analysis based on static analysis could filter out a large portion of false positives, but still need to be further investigated to increase the precision.

This approach is similarly extensible to other programming languages. For instance, in the case of Python, we can initially leverage package-level analysis tools to establish a dependency network at the package level. Following this, instruments such as pycg can be employed to construct a function call graph. Given that Python lacks an inheritance mechanism akin to Java, the construction of additional transitive relationships is rendered unnecessary. Subsequently, the methodology for constructing fine-grained dependency networks discussed in this paper can be engaged for both dependency analysis and the analysis of vulnerability propagation.

Key finding for RQ2: Most (81.26%) of vulnerability propagation detected by package-level analysis is false positive, which may increase the burden of developers in maintaining client programs. Fine-grained vulnerability propagation analysis needs to be further investigated.

6 Vulnerability Localization

Localizing the defective code causing vulnerability is one of the key steps to analyze the scope of vulnerability influence and fix the vulnerability. In the scenario of SSC, once a vulnerability is reported, it is important for client developers to figure out the root cause of the vulnerability and its location. Thus, client developers can determine whether the client program is affected by the vulnerability and take action quickly. More specifically, to understand whether a vulnerability affects a client program, client developers need to know the vulnerable functions in the library and then perform the fine-grained vulnerability propagation analysis explained in Section 5. In this section, we study the present situation of vulnerability localization in SSC, as well as the challenges and opportunities of locating vulnerability through the information extracted from the public vulnerability description.

6.1 Vulnerability Description Study

Localizing vulnerable code may either rely on high-quality test cases (Abreu et al. 2007) or well-written reports (Youm et al. 2015). Unfortunately, high-quality test cases may not be always available. Well-written reports may only be accessible for authorized security organizations to keep secrets of specific attacking ways and prevent the defect from being exploited by attackers. Compared to well-written bug reports, general vulnerability descriptions are widely accessible. For instance, when a CVE is reported, a general description is often released. However, general vulnerability descriptions are usually shorter and include less information. Therefore, vulnerability localization based on CVE descriptions is more difficult and challenging. To localize bugs based on CVE descriptions, we first need to figure out how much information can be obtained from a CVE description and whether it is sufficient to localize the vulnerable class or functions.

The CVE maintenance team has established guidelines for composing CVE descriptions (CVEProgram 2024). To produce a good CVE description, it is recommended that nine fields should be included: vulnerability type, affected products, vendor, affected and fixed versions, vulnerable components, vulnerable vectors, root cause, attackers, and impact. Example 1 in Fig. 8 is a complete illustration of a CVE description that contains all nine fields. A well-written CVE description can facilitate the identification of the vulnerable function. Specifically, the vendor, product, and version fields are useful for locating vulnerable libraries, while the component and vector fields typically specify vulnerable files/classes and methods, which can be employed to pinpoint vulnerable classes/functions. Additionally, the root cause and impact fields can help identify vulnerability locations. For instance, example 2 in Fig. 8 indicates that an overflow bug was caused by the UTF-8 decoder, allowing us to identify the code that decodes the UTF-8 text as the location of the vulnerability. Similarly, example 3 in Fig. 8 demonstrates that the vulnerability was caused by redirection functions, enabling experienced developers to identify the faulty code. It is important to note that not all CVE descriptions contain every field. We measure the percentage of each field in general CVE descriptions.

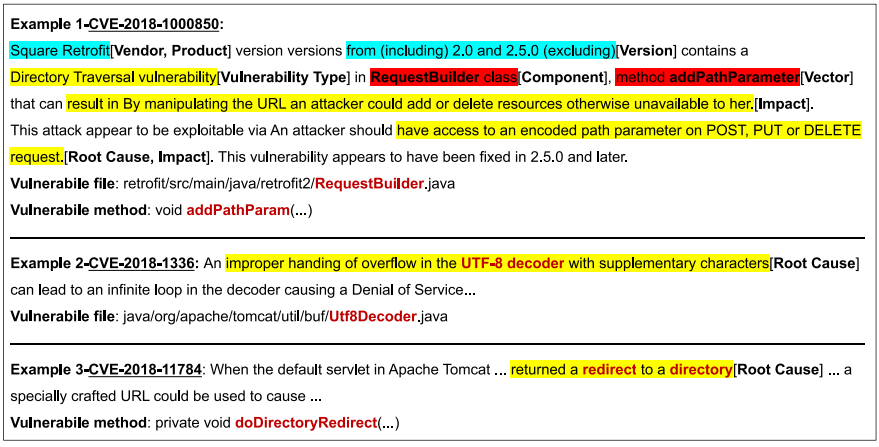

Fig. 8 The connection between the description and the vulnerable part

Study setup We conducted a manual analysis of 61 CVEs in the *vulnerability dataset* to determine the amount of information that can be obtained from CVE descriptions.

Initially, we query the NVD for textual descriptions of vulnerabilities corresponding to each CVE ID present in our dataset. Subsequently, through observation, we have identified that five of the aforementioned nine fields have a strong correlation with the vulnerability code, which aids in pinpointing the segments of code that contain the vulnerabilities. They are components, vectors, vulnerability type, root cause, and impact. The 'component' field assists in pinpointing the file or class where the vulnerability resides; 'vectors' often directly indicate the specific methods or variable names affected by the vulnerability; the 'vulnerability type' provides pattern information of the vulnerability, which can rapidly confirm the vulnerability segment for some common vulnerabilities; and 'root cause' and 'impact' may directly offer textual descriptions of the same fields or syntax as the vulnerability code, or indirectly provide textual descriptions of the functionality of the vulnerable code.

Therefore, we categorize these five concepts into four groups and proceed to mark them accordingly: Components, Vectors, Vulnerability Type, and Impact/Root Cause. We invited three postgraduates in the field of software engineering to annotate the same dataset. All of their research focuses are related to vulnerability detection, and they are capable of accurately identifying the relevant concepts of vulnerabilities. The rest annotation tasks of the study in this paper are also annotated by these three participants. We ask them to annotate the vulnerability-related concepts in accordance with the following rules: Components and vectors are typically programming words or recognizable paths (e.g., *RequestBuilder*, *addPathParameter* in Example-1 of Fig. 8). Vulnerability types are usually nouns or noun phrases preceding the word "vulnerability" (e.g., `Directory Traversal` Vulnerability) or followed by the word "cause". Impact and root cause refers to the remaining text, excluding nouns like products, vendors, versions, and attackers. Each annotator works independently to annotate the same data. If the annotation result is different, we employ the method of majority voting (Sheng et al. 2008) to ascertain the definitive annotation result. Using these criteria, we asked the annotators to extract relevant information and counted the number of concepts that included these elements in CVE descriptions.

Results and Findings In Fig. 9, the distribution of fields in vulnerability descriptions is depicted.

It is evident that 34 (56%) CVE descriptions specify the vulnerability type, whereas 44% of CVE descriptions do not identify the type of vulnerability, despite it being strongly recommended. This may be because identifying the vulnerability type is not always straightforward. The writer of the description may not be familiar with all types, and a single defect may belong to multiple types. However, we note that almost all CVEs are assigned at least one CWE [11] ID, which is a vulnerability category established by the MITRE Corporation. Therefore, if a vulnerability type is difficult to summarize, a CWE type can be added to the description. On the other hand, almost all (97%) of the CVE descriptions provide general information about the impact and root cause of the vulnerability. However, the details regarding the impact and root cause can vary significantly across different CVEs. At times, there may be multiple keywords that developers can use to trace the vulnerable code, while at other times the description may be too vague to comprehend the vulnerability. As there is rich textual information available regarding the impact or root cause, there is a strong possibility of extracting useful information from it.

[11] https://cwe.mitre.org

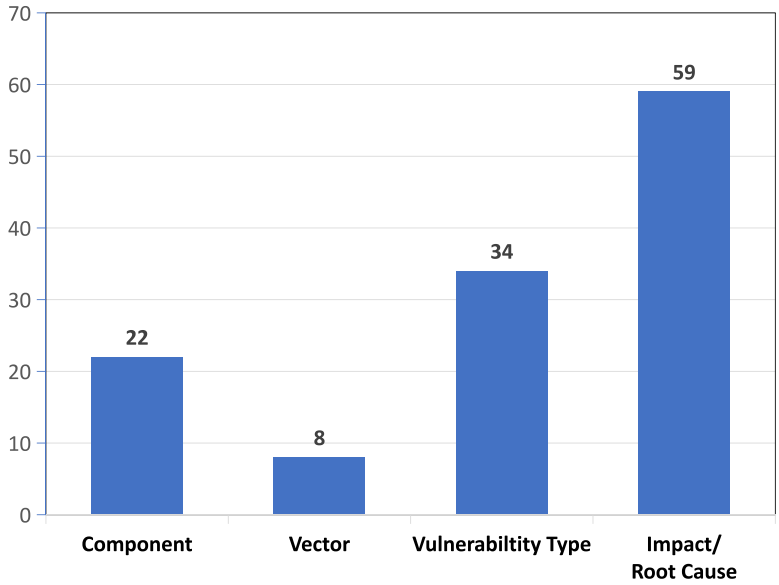

Fig. 9 Statistics of information in the description

Furthermore, the number of CVEs that provide information about the vulnerable file and method has decreased to 22 (36%) and 8 (13%), respectively. We believe that there are two reasons for this loss of critical information. Firstly, the CVE list is maintained by the MITRE Corporation, which authorizes various organizations worldwide to review vulnerability reports and assign CVE IDs with security information, including descriptions. This type of "crowd sourcing" organizational system makes it difficult to maintain the quality of CVE descriptions. Secondly, the security warning tends to conceal weak points to prevent other attackers from exploiting vulnerabilities. As a result, some reviewers may deliberately omit components or vectors. The lack of information about vulnerable files and methods makes it extremely challenging to localize vulnerabilities based on general CVE descriptions.

Key finding for RQ3.1: CVE descriptions usually summarize the type and root cause of vulnerabilities, but only a small part of them present the specific file (36%) and method (13%) causing the vulnerability. This brings challenges to localizing vulnerable code according to CVE descriptions.

6.2 Localization Based on Vulnerability Description

Furthermore, we investigate the possibility of localizing specific vulnerable code by utilizing the information in the description of vulnerability.

Study Setup We conduct the study by analyzing the connection between the 4 kinds of elements extracted from the description and the source code changed in the fix commits. Through our observation, the connection can be divided into three categories:

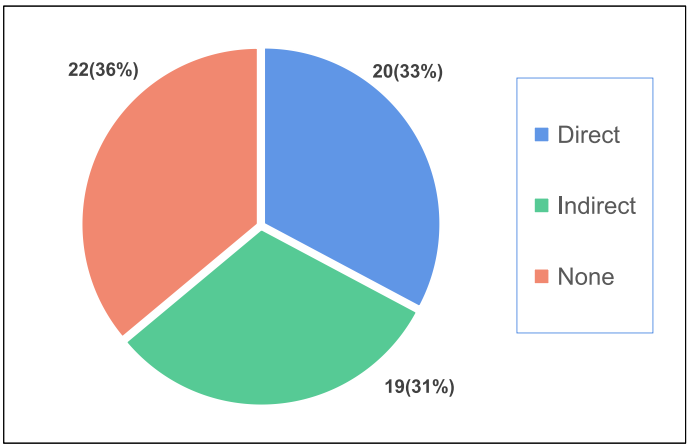

Fig. 10 Proportion of three categories of connection

– *Direct connection*: The information in the description directly points out the vulnerable files or methods (Example 1 in Fig. 8). Obviously, only the description containing information on components or vectors can have a direct connection with vulnerability.
– *Indirect connection*: The keywords in the description can be found in the name (or the word by decomposing name) of vulnerable variables, methods, classes, or files (e.g., Example 2&3 in Fig. 8).
– *No connection*: There is no similarity between the words in the description and the vulnerable code.

If the direct connection exists, we can directly localize the risk point from the description. Besides, if there is an indirect connection between the vulnerability description and source code, it also has a great possibility to localize the vulnerability by extracting keywords in the description. If there is no connection, localizing the vulnerable code from the information in the description is more challenging.

Based on the above criteria, we conduct the study on the *vulnerability dataset* by comparing the vulnerable file, method, and code in the dataset with the vulnerability description concepts extracted in Section 6.1. Three annotators are assigned the same data and work independently. Majority voting is also used to ascertain the definitive annotation result.

Results and Findings The result is shown in Fig. 10.

We observe that 33% of CVEs can directly locate the vulnerable part by utilizing the information in their description. All of these descriptions contain at least one of the components or vectors' names. Surprisingly, not all the CVEs whose descriptions contain components or vectors can lead to the location of vulnerability.

Example 4-CVE-2018-1000615:
ONOS ONOS Controller version 1.13.1 and earlier contains a Denial of Service (Service crash) vulnerability in **OVSDB component**(**Component**] in ONOS that can result in An adversary can remotely crash OVSDB service ONOS controller...
Vulnerabile file: protocols/ovsdb/rfc/src/main/java/org/onosproject/**ovsdb**/rfc/utils/VersionUtil.java

Fig. 11 The connection of CVE-2018-1000615 in ONOS

Figure 11 presents such an example, where the described component is the parent directory of the vulnerable file.

This indicates that we may not be able to localize the vulnerable code even if the components and vectors are provided. The described component and vector can even mislead the vulnerability localization. Beyond component and vector fields, we could also utilize different types of information to improve the localization precision.

Our observations reveal that 31% of CVEs exhibit an indirect connection between the description text and the vulnerable files or methods. Even if a description does not contain information about components or vectors, there may still be an indirect connection with the vulnerable source code. The natural language used in the description may contain valuable information (such as variable name, and functionality description) regarding the vulnerability location. For example, the root cause in Example-2 of Fig. 8 includes "decoder" and "redirect", which can be helpful for localizing the vulnerability. However, indirect information cannot be directly mapped to specific code, making it necessary to devise techniques to extract critical information from programming-related nouns or concepts. Existing techniques, such as information retrievers (Saha et al. 2013) and deep learning (Lam et al. 2017), attempt to extract key information from bug reports. In the future, these techniques can be repurposed to localize vulnerability locations based on the vulnerability description. Furthermore, beyond localizing vulnerabilities based on keywords, techniques can be designed to retrieve semantic information and build matches between natural language descriptions and code.

Overall, over 64% of CVEs exhibit a connection between the description and the vulnerable code. Although CVE descriptions contain less information than bug reports, obtaining CVE descriptions is much easier. Once the original location of the vulnerability is confirmed, automatic analysis of the vulnerability propagation along the software supply chain can help client developers evaluate the risk of their programs. There is significant potential to localize the risk point by resolving the description of vulnerabilities.

Key finding for RQ3.2: 64% of CVEs have a connection between the description text with the vulnerable code directly or indirectly. Localizing the risk point by resolving the CVE description has significant potential to help developers evaluate the risk of their programs.

7 Vulnerability Fix

Existing works have exhaustively studied the process of vulnerability repair and demonstrated that fixing vulnerabilities in SSC takes a long time (Alfadel et al. 2021; Decan et al. 2018; Prana et al. 2021; Yasumatsu et al. 2019). However, no existing work has studied how the vulnerabilities are fixed in both the library and client end. To fix a vulnerability in libraries, library developers could generate a patch and release a new version of the library. On the other hand, the affected clients may also propose some solutions to deal with the influence of the vulnerable library. In this section, we study the vulnerability fix patterns both in the library and client end.

7.1 Vulnerability Fix in Library End

To fix a vulnerability in SSC, the most straightforward way is to (1) fix the vulnerability in the library end, (2) release a new version, and (3) ask the client to use the updated version. We first study how library developers fix vulnerabilities in the library end.

Study setup. The process of fixing vulnerabilities in the library end is the same as generating patches for a single program, which has been widely studied (Gao et al. 2020; Huang et al. 2019; Li and Paxson 2017; Perl et al. 2015; Xu et al. 2017). Differently, since the third-party library is usually used in the form of the API invocation, the patches in the library end usually involve the modification of API name, parameters, default value, and return value. This modification can bring breaking changes which will affect API invocations in the client code when the client applies the patches. Hence, we analyze how libraries modify the API in the fix of vulnerabilities and its effects on client programs.

Therefore, vulnerability fix commits can be classified into three categories based on changes to the API: inner fixes, breaking changes, and others. Inner fixes modify the body of certain methods by adding or deleting code blocks, changing statements or expressions, etc., without changing the API signatures. For example, as shown in Listing 1, the patch for the vulnerability CVE-2018-8013 that occurs in *org.apache.xmlgraphics:batik-dom:1.9*, which has the vulnerabile method *readObject*.

Listing 1 Inner fix for CVE-2018-8013.

```
batik-dom/src/main/java/org/apache/batik/dom/AbstractDocument.java:
private void readObject(ObjectInputStream s):
...
- try {
- quad implementation =
     (DOMImplementation)c.getDeclaredConstructor().newInstance();
- } catch (Exception ex) {
+ if (DOMImplementation.class.isAssignableFrom(c)) {
+ quad try {
+ qquad implementation =
     (DOMImplementation)c.getDeclaredConstructor().newInstance();
+ quad } catch (Exception ex) {
+ quad }
+ } else {
+ quad throw new SecurityException("Trying to create object that is not
     a DOMImplementation.");
...
```

A verificaton is added before implementing the following code, which does not affect the name or the parameters in the declaration of the method. Applying this kind of fix hardly influences the function of the client.

Breaking changes introduce incompatible API changes, such as changes to API names, parameters, type of return values, default values, etc. For example, as shown in Listing 1, the patch for the vulnerability CVE-2017-1000498, which occurs in *com.caverock:androidsvg:1.2.1*, tries to add a boolean parameters to control the use of internal entities. If the method were called by other packages, some errors could be introduced during the compilation process. Applying this kind of fix not only induces compatibility problems possibly when calling the API, but also hides the vulnerability which makes it hard to be detected.

Others only change test and textual files, add new dependencies, etc. To some extent, this kind of fix does not harm the client.

Based on the above criteria, we conduct the study on the *vulnerability dataset* by manually analyzing the vulnerable code in the fix commits and classifying them into these categories. Three annotators are assigned the same data and work independently. In instances where their annotation results are inconsistent, the method of majority voting is still utilized to determine the ultimate annotation outcome.

Listing 2 Breaking change fix for CVE-2017-1000498.

```
androidsvg/src/main/java/com/caverock/androidsvg/SVGParser.java:
...
- private void parseUsingXmlPullParser(InputStream is) throws
    XmlPullParserException, IOException, SVGParseException{
+ private void parseUsingXmlPullParser(InputStream is, boolean
    enableInternalEntities) throws SVGParseException{
...
```

Results and Findings Our study found that breaking changes, which modify the name or parameters of methods, account for 20% of all vulnerability patches, while the remaining patches are mainly inner fixes. Among inner fixes, 44% add extra checks to the vulnerable parts, 28% add more functional code, and 5% directly delete the vulnerable code. When a library introduces inner fixes, client developers can directly upgrade the library version to apply the vulnerability patch. However, when a library introduces breaking changes, client developers need to upgrade the library and also fix the incompatible API usages, which is a complex task. As a result, most (82%) client developers are not willing to upgrade their dependencies (Kula et al. 2018). Additionally, inner fixes change the method body, which may alter the original function behaviors, posing a risk of incompatible functionality that further hinders client developers from upgrading their dependencies.

Key finding: Around 20% of vulnerability patches introduce breaking changes, while the remaining patches are inner fixes. Incompatible APIs and functionalities prevent client developers from upgrading their dependencies.

7.2 Vulnerability Fix Pattern in Client End

To deal with the security vulnerability propagated along the software supply chain, client programs may propose solutions to handle the influence of vulnerable libraries. Understanding how client programs deal with vulnerable libraries gives us implications on how to automatically solve this problem.

Study Setup To understand clients' reactions to vulnerabilities, we analyze all the fix commits that handle the vulnerabilities caused by their dependencies. We utilized the 92 clients that actually invoke the vulnerable methods in Section 5 as our initial data. It should be noted that these clients are all direct dependents of the respective libraries. This is because our reliance on manual analysis to identify the fix commits for each vulnerability necessitates a significant expenditure of time. Furthermore, changes in direct dependencies can be directly reflected in the pom.xml files of these clients, which also facilitates easier identification. Subsequently, we need to determine whether these clients have addressed the vulnerabilities and identify the specific commit content of the fixes. The steps for the search process are as follows:

- **S1. Confirm the vulnerability is repaired** We initially conduct a manual analysis of the pom.xml files associated with each software package, ascertaining whether the version of the third-party library implicated in the vulnerability within the latest codebase of the client has been updated to the patched version. Should the version of the library in question still be utilizing the vulnerable version, it is provisionally noted as unfixed; otherwise, it is marked as fixed.
- **S2. Define the search boundaries** We will address the clients marked as "fixed" and "unfixed" separately. *For clients that have been fixed*, we manually review all historical commits of the pom.xml file to identify the final "fix commit" that transitions from the vulnerable version to the patched version of the library. Here, we consider that clients which have upgraded the library version in the SBOM may initially employ some form of temporary fix, but the ultimate resolution remains the upgrade of the dependency. However, it is still necessary for us to understand their initial response to the vulnerability and the corresponding commit. Therefore, we define the search scope for this commit to be between the time of the release of the fixed version of the vulnerable library and the time when the client upgraded the vulnerable library. *For clients that are marked as "unfixed,"*, there remains the possibility that they have addressed the vulnerabilities through alternative means. Consequently, we will set the search scope for the "fix commit" within a period extending from the release date of the patched version of the vulnerable library to the subsequent six months. We chose 6 months as the counting period because the average time for vulnerability repair in dependencies is within 5 months following the release of the vulnerability fix (Prana et al. 2021).
- **S3. Filter potential fix-commit candidates.** we retrieved all commits within the search boundaries using the GitHub API. If the commit message contained the text "vulnerab", "fix" or "CVE", CVE ID, or the name of the library, we recorded it as a candidate for the client fix. The first four string suggest that the commit is related to vulnerability repair, while "the name of the library" indicates that the commit is associated with the third-party library in use.
- **S4. Ascertain the specific location of the definitive fix commit.** Finally, we manually analyzed each candidate to confirm the specific commit where the client implemented the fix.

Ultimately, we discovered that 27 clients were still utilizing the vulnerable version of the library in their most recent updates. The remaining 65 clients had addressed the vulnerability. Excluding one package that could not access its hosted repository, we identified the "fix commit" for 64 clients.

According to our observation, there are 3 main kinds of fix strategies: Directly upgrading the library version, deleting the vulnerable dependency, and manually patching. Directly upgrading the library version or removing the vulnerable dependency frequently occurs within the *pom.xml* files. As we mentioned in Section 3, in the Maven ecosystem, projects utilize the pom.xml file to document dependency information, and dependencies can be added, removed, or updated by modifying this file. Just like the example shown in Listing 3, the client *org.apache.accumulo:accumulo-test* upgrade the library *libthrift* from the vulnerable version *0.9.3* to the fixed version *0.0.3-1* in the commit {87cf69}.

Listing 3 Directly applying update in the pom.xml

```
pom.xml:
...
- <thrift.version>0.9.3</thrift.version>
+ <thrift.version>0.9.3-1</thrift.version>
```

```
...
```

As for manually patching, some library fixes are categorized as "break changes," as described in Section 7.1, like the example in Listing 4. Implementing these fixes necessitates that developers of clients manually alter the usage of certain APIs. Furthermore, some projects are equipped with development teams possessing strong technical capabilities. To expedite the process of patching vulnerabilities, these teams may choose to manually modify or disable certain APIs of the libraries.

Based on the aforementioned observations, we conducted the study by manually analyzing fixed commits of client programs and classified them into these categories.

Listing 4 Manully patching the vulnerability.

```
modules/plugin/svg/src/main/java/org/geotools/renderer/style/
    RenderableSVGCache.java:
...
- import org.apache.batik.dom.svg.SAXSVGDocumentFactory;
+ import org.apache.batik.anim.dom.SAXSVGDocumentFactory;
...

pom.xml:
...
+ <batik.version>1.10</batik.version>
...
  <artifactId>batik-dom</artifactId>
- <version>1.7</version>
+ <version>${batik.version}</version>
...
```

Three annotators are assigned the same data and work independently. If there are discrepancies in the aggregated results, we still employ the method of majority voting to determine the final annotation outcome.

Result and Statistics Table 5 shows the result of analysis.

It is evident that the majority of maintainers of client systems select the approach of directly updating the libraries to address any potential security vulnerabilities. The proportion of individuals in this segment exceeds 95%. Among them, seven clients and libraries originate from the same project. The remediation time for these clients tends to be more rapid, as the resolution of vulnerabilities within the library may be accomplished by the same team or developers who trust each other, thereby ensuring that the code changes are fully trusted. The remaining 55 clients are from external projects, with their vulnerability remediation times ranging from one to 57 months, with an average remediation time of 5 months. This may be attributed to the fact that they require additional time to receive notifications of the library vulnerabilities, as well as to deliberate on whether to adopt the provided fixes.

Table 5 Statistics on clients that have fixed vulnerabilities

Fix strategy	Clients
Directly upgrade dependency	(61)
- Clients from the same project	7
- Clients from the external project	54
Upgrade dependency with manual modifications	2
Remove dependency	1

As mentioned earlier, modifying dependency versions can be challenging when breaking changes are introduced in library updates, as it requires fixing incompatible API invocations or module imports. This is the primary obstacle that prevents client developers from upgrading or downgrading their dependencies. This type of fix, which necessitates addressing breaking changes, accounts for approximately 10% of all fixes.

We also observed that some clients directly delete the vulnerable dependency, deactivate it, or remove the invocation of the vulnerable API. While deleting or deactivating vulnerable dependencies can be a strategy to eliminate the effects of library vulnerabilities, it may significantly affect the program's behavior. To address the problem caused by dependency deletion or deactivation, client developers may choose to (1) implement the functionality themselves, (2) find a secure alternative library that provides the same functionality, or (3) temporarily disable the dependency and add it back when a patch is released. However, implementing functionality provided by libraries can be complex, and alternative libraries may not always be available. Temporarily deleting or deactivating vulnerable component may be an acceptable workaround for some clients when a library vulnerability has not been fixed. For instance, we observed that the client *com.github.bordertech.wcomponents:wcomponents-core* deactivated the vulnerable dependency immediately after the vulnerability was reported to avoid its impact. The client excluded the vulnerable component *batik-css* and *xercesImpl* in the library *antisamy* in commit d59904, as shown in Listing 5. Later, when the vulnerability was fixed, the client developer added the dependency back and upgraded it to the new library version.

However, temporarily deleting or deactivating vulnerable dependencies is not always feasible, and this strategy can only be applied if disabling the vulnerable dependency does not affect the core functionality of the client program.

Listing 5 Temporally disable an component when vulnerability was reported (commit id **d59904**).

```
wcomponents-core/pom.xml:
...
+ <groupId>org.owasp.antisamy</groupId>
+ <artifactId>antisamy</artifactId>
+ <version>1.5.7</version>
+ <exclusions>
+     <exclusion>
+         <groupId>org.apache.xmlgraphics</groupId>
+         <artifactId>batik-css</artifactId>
+     </exclusion>
+     <exclusion>
+         <groupId>xerces</groupId>
+         <artifactId>xercesImpl</artifactId>
+     </exclusion>
+ </exclusions>
...
```

Key finding: To deal with vulnerable dependencies, 95% of the affected clients upgrade the library when the new library version is released. Moreover, 3% of upgrades require solving breaking changes. Moreover, some of the affected clients choose to temporally delete the vulnerable dependency as a workaround.

8 Implication and Lesson Learned

The key findings is listed in Table 6. Based on the these findings, we have several implications for vulnerability detection, security warning generation, vulnerability localization and repair, software security Ops.

Vulnerability detection Existing vulnerability detection tools assume the third-party library is secure and just focus on the program under detection (PUD). For instance, static analyzers usually just analyze the PUD while leaving third-party libraries alone. Dynamic testing techniques that rely on instrumentation (e.g., Grey-box fuzzing AFL 2019, Sanitizer Serebryany et al. 2012), usually just instrument the PUD while not systematically testing third-party libraries. Relying on developers of third-party libraries to exhaustively test their products is

Table 6 The key findings of vulnerability characteristic on software supply chain and their corresponding implications

Finding on Vulnerability Source (Section 4)	Implications (Section 8)
(1) 99% of vulnerabilities in client programs are inherited from its dependencies, especially indirect dependencies.	When testing or analysing programs, besides the code of program under test, its dependencies should also be tested and analyzed. Helping developers at the development stage (e.g., recommending secure libraries) instead of the post-development testing stage can help improve the security of software systems.
Finding on Vulnerability Propagation (Section 5)	**Implications (Section 8)**
(2) Most (81.26%) of vulnerability propagation detected by package-level analysis is false positive	Future research should pay more attention to fine-grained vulnerability propagation analysis. To warn developers about security risks, vulnerability analysis techniques should be both scalable and precise.
Finding on Vulnerability Location (Section 6)	**Implications (Section 8)**
(3) CVE descriptions usually summarize vulnerability type and root cause, but only a small part of them present the vulnerable file (36%) and method (13%).	CVE descriptions are good sources to analyze the vulnerabilities, but relying solely on explicit information is insufficient to precisely localize the cause of vulnerabilities.
(4) 64% of CVEs exhibit connections between description text with vulnerable code directly or indirectly.	Mining deep key information embedded in CVE descriptions can potentially increase localization accuracy. Additionally, combining code analysis with description analysis could be a strategy for vulnerability localization.
Finding on Fix Pattern (Section 7)	**Implications (Section 8)**
(5) Roughly 20% of vulnerability patches introduce breaking changes, while the rest are internal fixes.	Incompatible APIs and functionalities prevent client developers from upgrading their dependencies. It is worthwhile to investigate techniques to automatically handle breaking changes.
(6) When faced with vulnerable dependencies, 95% of affected clients choose to upgrade their library as soon as a new version is released.	Upgrading is one of the most widely-used strategies for dealing with vulnerable dependencies, and automated dependency upgrades should receive more attention from both academia and industry.
(7) In response to vulnerable dependencies, affected clients may choose to delete the dependency or remove relevant code.	When new version library has not been released, deleting or deactivating the vulnerable dependency can be an acceptable workaround.

dangerous, especially for open-source software. Open-source software has the characteristics of openness, common participation, and free dissemination. Software vulnerabilities are common in open-source software, one possible reason could be the weak security awareness or insufficient technical skills of developers. Moreover, some open-source software cannot prevent attackers from injecting malicious code into software supply chain attacks (Synopsys 2024). Considering that around 89% of vulnerabilities in client programs are caused by their dependencies, it is necessary to also analyze the code of third-party libraries. Third-party libraries are usually used by invoking public APIs. To test software dependencies, it is required to invoke the vulnerability **and** generate certain arguments satisfying the precondition. In other words, when testing APIs in libraries (e.g., fuzzing APIs using LibFuzzer), besides testing a single API, we should also test different combinations of API execution, different execution order, and arguments. Vulnerability detection techniques for efficiently testing dependencies in the SSC are desired.

Security Warning Generation To analyze the security risks associated with the use of third-party libraries, existing tools, such as Google Insight, and BlackDuck (2023), (1) perform software dependency analysis or clone detection, and (2) analyze vulnerability propagation and (3) report the potential security risks. However, existing tools often produce a lot of false positives. Our study has shown that the majority of vulnerable functions in libraries are actually not reachable by the corresponding client projects. Therefore, there is a need to develop fine-grained analysis techniques that can assess the threats posed by third-party libraries and generate security warnings. Fine-grained analysis based on static analysis can help improve the precision of vulnerability detection to some extent. However, such analysis techniques are still not precise enough, and they also face scalability issues. Therefore, future research should focus on developing analysis techniques that can achieve both scalability and precision in generating warnings caused by vulnerabilities in third-party libraries.

Secure Software Development When developing software systems, developers usually prioritize functionality over software security. Although security analysis as a post-development process may eliminate many security risks, it is hard to eliminate the vulnerabilities introduced by improper design choices. For instance, a developer may choose to rely on a third-party library l_1 to implement a certain functionality, even though the library has many unpatched vulnerabilities. When the risks are detected in the post-development process, however, replacing library l_1 with an alternative library is very challenging. This is mainly because the developed software may be highly integrated with l_1 and replacing it may require redesigning or even re-implementing many parts of the system.

Considering around 89% of vulnerabilities in client programs are caused by their dependencies, we believe it is crucial to make developers aware of the risks associated with third-party libraries at the development stage. More specifically, when developers import a third-party library or invoke a certain API, security analyzers (such as our fine-grained vulnerability propagation analyzer) could provide real-time warnings or suggestions. For example, they could warn developers of the risks of using a library, suggest the correct way to use an API, or even recommend a similar but more secure library. By helping developers at the development stage instead of the testing stage, it can potentially further improve the security of software systems.

Vulnerability Localization Current techniques for localizing vulnerable code typically rely on well-written bug reports or high-quality test cases. However, these resources are not always accessible, particularly for zero-day vulnerabilities. While CVE reports can be a good source for inferring vulnerable code, most CVE descriptions are poorly written, with only 30% and

13% of them providing specific file and method information that causes the vulnerability, respectively. Fortunately, 86% of CVE descriptions analyze the root cause of vulnerabilities, and 57% of them present the vulnerability type, which can aid in localizing the vulnerable code. It is therefore worthwhile to investigate techniques that can automatically analyze the information embedded in natural language descriptions and localize the vulnerable code. With this limited information, client developers can then assess the risk of their programs through combined vulnerability localization and propagation analysis.

Another potential idea may be combining report analysis and code analysis. In recent years, deep-learning-based vulnerability detection has gained increasing attention (Chakraborty et al. 2020; Li et al. 2018). These techniques aim to train a model to determine whether a given piece of code (e.g., a function) has vulnerabilities. Learning-based vulnerability detection has achieved promising results, with around 90% accuracy on real-world programs. Therefore, we envision an idea of combining report analysis with learning-based vulnerability detection. This approach would involve using report analysis to identify the candidate vulnerable code and then using vulnerability detection to validate whether the candidate code contains vulnerabilities. By combining these two methods, we hope to increase the accuracy of vulnerability localization.

Moreover, the project-level dependency graph maintained by the package manager is too coarse-grained to model method-level dependency relationships. As a result, alarming all dependencies when a vulnerability in a library is reported can lead to many false positives, since many dependencies may not use the vulnerable function. However, once the vulnerable methods in the library are localized, a fine-grained method-level dependency graph can help precisely analyze the scope of the vulnerability's influence by determining which dependents are affected by the vulnerable code.

Vulnerability Repair Fixing vulnerabilities can be done either at the library or client end. Library developers typically fix bugs by introducing patches, while client developers usually handle the impact of library vulnerabilities by upgrading the library version. Existing techniques have explored automatic repairing processes at both ends. Specifically, automated program repair techniques rely on pre-defined transformation operators that transform the buggy program into a correct one. Many of the fix patterns, such as adding checks, changing code sequences, and deleting code, are supported by modern automatic program repair tools. However, the pre-defined transformation operators are still quite limited. Defining more transformation operators will make the patch space very large, making it difficult to search for the correct patch among them. Reducing the transformation operators will significantly affect their repairability, making it impossible to generate the correct patch. Fortunately, existing researches have observed that different types of vulnerabilities have different fix patterns (Islam et al. 2020; Li and Paxson 2017). For a particular type of vulnerability, designing specific transformation operators to fix it accurately is worthwhile.

Automatically upgrading dependency versions has been studied (Mirhosseini and Parnin 2017), but these techniques have not been widely applied in practice. Considering that a large portion of clients choose to upgrade library versions when dealing with vulnerable libraries, it is still worthwhile to investigate ways to automate the library upgrade process, especially in handling the effects of breaking changes. Automatically generating proxy programs to fill the gap between old API invocations and new libraries can be an interesting direction to explore. Additionally, disabling the vulnerable dependency is an acceptable temporary workaround when the updated library version has not been released. However, disabling a vulnerable dependency may affect the core functionality of client programs. Moreover, finding an alternative secure library after disabling the vulnerable dependency has not been

explored by the community. Building a database to map the libraries/functions that implement the same functionality would provide client developers with more options for handling vulnerable libraries.

9 Threats to Validity

As with previous real-world characteristic studies, several external and internal threats may affect the validity of our studies.

A potential external threat is the representativeness of our datasets. We selected a set of Java libraries and their corresponding vulnerabilities as our study dataset, and the analyzed results may not be generalizable to other programs or programming languages. Additionally, we only examined vulnerabilities that have been fixed, while non-fixed vulnerabilities may have different characteristics from the dataset used in this paper. Another concern is that the scope of the dataset is limited to direct dependents, ignoring indirect dependents. However, conceptually, a client program can also be the library that is depended on by other programs, so our analysis should be generalizable to indirect dependents. Furthermore, our dataset does not cover all types of vulnerabilities, but fortunately, the examined vulnerabilities in this study are the most common and significant ones. The distribution of vulnerabilities in different CWE categories is very similar to CWE Top 25 Most Dangerous Software Weaknesses, indicating that our dataset can represent general trends.

The internal threat mainly comes from the methodology of manually analyzing vulnerabilities, which can be error-prone. However, the authors of this paper are familiar with the examined language, application, and vulnerabilities, which mitigates this potential threat. To further mitigate the effect of potential threats, all authors double-checked all the results.

10 Conclusion

In this paper, we provide real-world software supply chain vulnerability characteristics, including the vulnerability source, propagation, location characteristics, and patch strategies. We propose a more efficient finer-grained vulnerability analysis method and optimizing strategy to conduct method-level vulnerability analysis. The proposed tool could achieve more efficient and precise vulnerability propagation analysis. The results of the analysis revealed several key findings. For instance, 81.26% of vulnerability propagation cases detected by package-level analysis were false positives. Additionally, 64% of CVEs exhibited a connection between the description text and the vulnerable code either directly or indirectly. When faced with vulnerable libraries, client developers often choose to upgrade them or delete them directly. Based on those findings, we suggest that future work should explore more fine-grained and precise vulnerability propagation analysis, investigate how to automatically localize the vulnerable code from the description of vulnerability in a public database, and design new automated vulnerability repairing tools by learning from the patterns with more detailed characteristics accessed from the software supply chain.

Acknowledgements This work was supported partly by National Natural Science Foundation of China under Grant No. 62141209, 62202026, and partly by Guangxi Collaborative Innovation Center of Multi-source Information Integration and Intelligent.

Data Availability The source and generating process of datasets are presented in Section 3. Data supporting the findings of this study are publicly available in the GitHub repository, https://github.com/YijunShen/supply_chain_vul

Declarations

Conflicts of Interest The authors declared that they have no conflict of interest.

References

Abreu R, Zoeteweij P, Van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: Testing: Academic and industrial conference practice and research techniques-mutation (TAICPART-MUTATION 2007), pp 89–98. https://doi.org/10.1007/10.1109/TAIC.PART.2007.13

AFL (2019) American fuzzy lop (afl). http://lcamtuf.coredump.cx/afl Accessed 08 Apr 2023

Alfadel M, Costa DE, Shihab E (2021) Empirical analysis of security vulnerabilities in python packages. In: 2021 IEEE International conference on software analysis, evolution and reengineering (SANER), pp 446–457. https://doi.org/10.1109/SANER50967.2021.00048

Almhana R, Mkaouer MW, Kessentini M, Ouni A (2016) Recommending relevant classes for bug reports using multi-objective search. 2016 31st IEEE/ACM International conference on automated software engineering (ASE) pp 286–295. https://doi.org/10.1145/2970276.2970344

BlackDuck (2023) Black duck. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html Accessed 31 May 2023

Bui QC, Scandariato R, Ferreyra NED (2022) Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In: 2022 IEEE/ACM 19th International conference on mining software repositories (MSR), pp 464–468. https://doi.org/10.1145/3524842.3528482

Chakraborty S, Krishna R, Ding Y, Ray B (2020) Deep learning based vulnerability detection: are we there yet? IEEE Trans Softw Eng 48:3280–3296. https://doi.org/10.1109/TSE.2021.3087402

Cox J, Bouwers E, Van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International conference on software engineering (ICSE), vol 2, pp 109–118. https://doi.org/10.1109/ICSE.2015.140

CVEProgram (2024) Key details phrasing. https://www.cve.org/Resources/General/Key-Details-Phrasing.pdf Accessed 12 June 2024

CWE (2021) 2021 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html Accessed 25 April 2024

Dann A, Plate H, Hermann B, Ponta SE, Bodden E (2022) Identifying challenges for oss vulnerability scanners - a study & test suite. IEEE Trans Softw Eng 48(9):3613–3625. https://doi.org/10.1109/TSE.2021.3101739

Debroy V, Wong WE (2013) A consensus-based strategy to improve the quality of fault localization. Softw Pract Experience 43:989–1011. https://doi.org/10.1002/spe.1146

Decan A, Mens T, Constantinou E (2018) On the impact of security vulnerabilities in the npm package dependency network. 2018 IEEE/ACM 15th International conference on mining software repositories (MSR) pp 181–191. https://doi.org/10.1145/3196398.3196401

Dependabot (2023) Dependabot. https://github.com/dependabot Accessed 30 May 2023

Dependency-Check (2023) Owasp dependency check. https://owasp.org/www-project-dependency-check Accessed 30 May 2023

Dodge Y (2008) Spearman rank correlation coefficient. In: The Concise encyclopedia of statistics, Springer, New York, NY, pp 502–505. https://doi.org/10.1007/978-0-387-32833-1_379

Enck W, Williams L (2022) Top five challenges in software supply chain security: observations from 30 industry and government organizations. IEEE Secur Priv 20(2):96–100. https://doi.org/10.1109/MSEC.2022.3142338

Foundation TL (2020) Open source software supply chain security. https://www.linuxfoundation.org/tools/open-source-software-supply-chain-security/ Accessed 06 May 2022

Gao X, Wang B, Duck GJ, Ji R, Xiong Y, Roychoudhury A (2020) Beyond tests: program vulnerability repair via crash constraint extraction. ACM Trans Softw Eng Methodol (TOSEM) 30(2). https://doi.org/10.1145/3418461

Gartner (2021) Application development will shift to application assembly and integration. https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences Accessed 12 Apr 2023

Gkortzis A, Feitosa D, Spinellis D (2021) Software reuse cuts both ways: an empirical analysis of its relationship with security vulnerabilities. J Syst Softw 172:110653. https://doi.org/10.1016/j.jss.2020.110653

Goues CL, Pradel M, Roychoudhury A (2019) Automated program repair. Commun ACM 62:56–65. https://doi.org/10.1145/3318162

Grieco G, Grinblat GL, Uzal LC, Rawat S, Feist J, Mounier L (2016) Toward large-scale vulnerability discovery using machine learning. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. https://doi.org/10.1145/2857705.2857720

Huang Z, Lie D, Tan G, Jaeger T (2019) Using safety properties to generate vulnerability patches. In: Proceedings of the 40th IEEE symposium on security and privacy (S&P), pp 539–554. https://doi.org/10.1109/SP.2019.00071

Imtiaz N, Thorn S, Williams L (2021) A comparative study of vulnerability reporting by software composition analysis tools. In: Proceedings of the 15th ACM / IEEE International symposium on empirical software engineering and measurement (ESEM). https://doi.org/10.1145/3475716.347576

Islam MJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: fix patterns and challenges. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 1135–1146. https://doi.org/10.1145/3377811.3380378

Jang J, Agrawal A, Brumley D (2012) Redebug: finding unpatched code clones in entire os distributions. In: 2012 IEEE Symposium on security and privacy, pp 48–62. https://doi.org/10.1109/SP.2012.13

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. Empir Softw Eng 23:384–417. https://doi.org/10.1007/s10664-017-9521-5

Lam AN, Nguyen AT, Nguyen HA, Nguyen TN (2017) Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International conference on program comprehension (ICPC), pp 218–229. https://doi.org/10.1109/ICPC.2017.24

Lauinger T, Chaabane A, Arshad S, Robertson W, Wilson C, Kirda E (2017) Thou shalt not depend on me: analysing the use of outdated javascript libraries on the web. https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_02B-1_Lauinger_paper.pdf Accessed 02 July 2024

Li F, Paxson V (2017) A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2201–2215. https://doi.org/10.1145/3133956.3134072

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: a deep learning-based system for vulnerability detection. In: Network and Distributed Systems Security (NDSS) symposium 2018. https://doi.org/10.14722/ndss.2018.23158

Liu C, Chen S, Fan L, Chen B, Liu Y, Peng X (2022) Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. 2022 IEEE/ACM 44th International conference on software engineering (ICSE) pp 672–684. https://doi.org/10.1145/3510003.3510142

Liu K, Koyuncu A, Bissyande TF, Kim D, Klein J, Le Traon Y (2019) You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference on software testing, validation and verification (ICST), pp 102–113. https://doi.org/10.1109/ICST.2019.00020

Mirhosseini S, Parnin C (2017) 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), pp 84–94. https://doi.org/10.1109/ASE.2017.8115621

Moreno L, Treadway JJ, Marcus A, Shen W (2014) On the use of stack traces to improve text retrieval-based bug localization. In: 2014 IEEE International conference on software maintenance and evolution, pp 151–160. https://doi.org/10.1109/ICSME.2014.37

Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) Vccfinder: finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, pp 426–437. https://doi.org/10.1145/2810103.2813604

Pfretzschner B, ben Othmane L (2017) Identification of dependency-based attacks on node.js. In: Proceedings of the 12th international conference on availability, reliability and security, pp 1–6. https://doi.org/10.1145/3098954.3120928

Pham NH, Nguyen TT, Nguyen HA, Nguyen TN (2010) Detection of recurring software vulnerabilities. Proceedings of the 25th IEEE/ACM international conference on automated software engineering pp 447–456. https://doi.org/10.1145/1858996.1859089

Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C (2019) A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proceedings of the 16th international conference on mining software repositories, pp 383–387. https://doi.org/10.1109/MSR.2019.00064

Prana GAA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D (2021) Out of sight, out of mind? how vulnerable dependencies affect open-source projects. Empir Softw Eng 26(4):1–34. https://doi.org/10.1007/s10664-021-09959-3

Reid D, Jahanshahi M, Mockus A (2022) The extent of orphan vulnerabilities from code reuse in open source software. In: Proceedings of the 44th international conference on software engineering, pp 2104–2115. https://doi.org/10.1145/3510003.3510216

Reps T, Ball T, Das M, Larus J (1997) The use of program profiling for software maintenance with applications to the year 2000 problem. In: Joint european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE), pp 432–449. https://doi.org/10.1145/267896.267925

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International conference on automated software engineering (ASE), pp 345–355. https://doi.org/10.1109/ASE.2013.6693093

Santelices R, Jones JA, Yu Y, Harrold MJ (2009) Lightweight fault-localization using multiple coverage types. In: 2009 IEEE 31st International conference on software engineering, pp 56–66. https://doi.org/10.1109/ICSE.2009.5070508

Sejfia A, Schäfer M (2022) Practical automated detection of malicious npm packages. In: 2022 IEEE/ACM 44th International conference on software engineering (ICSE), pp 1681–1692. https://doi.org/10.1145/3510003.3510104

Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) Addresssanitizer: a fast address sanity checker. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany Accessed 30 June 2024

Sheng VS, Provost F, Ipeirotis PG (2008) Get another label? improving data quality and data mining using multiple, noisy labelers. In: Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining, pp 614–622. https://doi.org/10.1145/1401890.1401965

Sonatype (2021) 2021 state of the software supply chain. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021 Accessed 06 May 2022

Soto-Valero C, Harrand N, Monperrus M, Baudry B (2021) A comprehensive study of bloated dependencies in the maven ecosystem. Empir Softw Eng 26:45. https://doi.org/10.1007/s10664-020-09914-8

Staicu CA, Pradel M, Livshits B (2016) Understanding and automatically preventing injection attacks on node.js. https://www.doc.ic.ac.uk/~livshits/papers/tr/nodejs_tr.pdf Accessed 02 July 2024

Synopsys (2022) 2022 open source security and risk analysis report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html Accessed 06 May 2022

Synopsys (2024) 2024 open source security and risk analysis report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html Accessed 12 June 2024

Tan X, Gao K, Zhou M, Zhang L (2022) An exploratory study of deep learning supply chain. In: 2022 IEEE/ACM 43rd International conference on software engineering (ICSE), pp 86–98. https://doi.org/10.1145/3510003.3510199

Viega J, Bloch J, Kohno Y, McGraw G (2000) Its4: a static vulnerability scanner for c and c++ code. In: Proceedings 16th annual computer security applications conference (ACSAC'00), pp 257–267. https://doi.org/10.1109/ACSAC.2000.898880

Vu DL, Pashchenko I, Massacci F, Plate H, Sabetta A (2020) Towards using source code repositories to identify software supply chain attacks. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, pp 2093–2095. https://doi.org/10.1145/3372297.3420015

Wang S, Lo D, Lawall J (2014) Compositional vector space models for improved bug localization. In: 2014 IEEE International conference on software maintenance and evolution, pp 171–180. https://doi.org/10.1109/ICSME.2014.39

Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y (2020) 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), pp 35–45. https://doi.org/10.1109/ICSME46990.2020.00014

Wijayasekara D, Manic M, McQueen M (2014) Vulnerability identification and classification via text mining bug databases. In: IECON 2014-40th Annual conference of the IEEE industrial electronics society, pp 3612–3618. https://doi.org/10.1109/IECON.2014.7049035

Wong CP, Xiong Y, Zhang H, Hao D, Zhang L, Mei H (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International conference on software maintenance and evolution, pp 181–190. https://doi.org/10.1109/ICSME.2014.40

Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng 42(8):707–740. https://doi.org/10.1007/10.1109/TSE.2016.2521368

Wu Y, Yu Z, Wen M, Li Q, Zou D, Jin H (2023) Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In: 2023 IEEE/ACM 45th International conference on software engineering (ICSE), pp 1046–105 https://doi.org/10.1109/ICSE48619.2023.00095

Wyss E, De Carli L, Davidson D (2022) What the fork? finding hidden code clones in npm. In: Proceedings of the 44th international conference on software engineering, pp 2415–2426. https://doi.org/10.1145/3510003.3510168

Xia P, Matsushita M, Yoshida N, Inoue K (2013) Studying reuse of out-dated third-party code in open source projects. J Inf Process 9:155–161. https://doi.org/10.11309/JSSST.30.4_98

Xie X, Chen TY, Kuo FC, Xu B (2013) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Methodol (TOSEM) 22:1–40. https://doi.org/10.1145/2522920.2522924

Xu Z, Chen B, Chandramohan M, Liu Y, Song F (2017) Spain: security patch analysis for binaries towards understanding the pain and pills. In: 2017 IEEE/ACM 39th International conference on software engineering (ICSE), pp 462–472. https://doi.org/10.1109/ICSE.2017.49

Xuan J, Monperrus M (2014) Learning to combine multiple ranking metrics for fault localization. In: 2014 IEEE International conference on software maintenance and evolution, pp 191–200. https://doi.org/10.1109/ICSME.2014.41

Yamaguchi F, Wressnegger C, Gascon H, Rieck K (2013) Chucky: exposing missing checks in source code for vulnerability discovery. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security https://doi.org/10.1145/2508859.2516665

Yasumatsu T, Watanabe T, Kanei F, Shioji E, Akiyama M, Mori T (2019) Understanding the responsiveness of mobile app developers to software library updates. In: Proceedings of the ninth ACM conference on data and application security and privacy, pp 13–24. https://doi.org/10.1145/3292006.3300020

Youm KC, Ahn J, Kim J, Lee E (2015) Bug localization based on code change histories and bug reports. In: 2015 Asia-Pacific software engineering conference (APSEC), pp 190–197. https://doi.org/10.1109/APSEC.2015.23

Zapata RE, Kula RG, Chinthanet B, Ishio T, Matsumoto K, Ihara A (2018) 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), pp 559–563. https://doi.org/10.1007/10.1109/ICSME.2018.00067

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International conference on software engineering (ICSE), pp 14–24. https://doi.org/10.1109/ICSE.2012.6227210

Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Smallworld with high risks: a study of security threats in the npm ecosystem. In: Proceedings of the 28th USENIX conference on security symposium, pp 995–1010. https://doi.org/10.5555/3361338.3361407

Authors and Affiliations

Yijun Shen[1] · Xiang Gao[1,2] · Hailong Sun[1,2] · Yu Guo[1]

✉ Xiang Gao
xiang_gao@buaa.edu.cn

Yijun Shen
shenyijun@buaa.edu.cn

Hailong Sun
sunhl@buaa.edu.cn

Yu Guo
yuguo@buaa.edu.cn

1 State Key Laboratory of Complex & Critical Software Environment (SKLCCSE), Beihang University, Beijing, China

2 Hangzhou Innovation Institute of Beihang University, Beijing, China