Empirical Software Engineering (2022) 27: 90
https://doi.org/10.1007/s10664-022-10131-8

Characterizing usages, updates and risks of third-party libraries in Java projects

Kaifeng Huang[1] · Bihuan Chen[1] · Congying Xu[1] · Ying Wang[1] · Bowen Shi[1] · Xin Peng[1] · Yijian Wu[1] · Yang Liu[2]

Accepted: 11 February 2022 / Published online: 14 April 2022

Abstract

Third-party libraries are a key building block in software development as they allow developers to reuse common functionalities instead of reinventing the wheel. However, third-party libraries and client projects are developed and continuously evolving in an asynchronous way. As a result, outdated third-party libraries might be commonly used in client projects, while developers are unaware of the potential risk (e.g., security bugs) in usages. Outdated third-party libraries might be updated in client projects in a delayed way, while developers are less aware of the potential risk (e.g., API incompatibilities) in updates. Developers of third-party libraries may be unaware of how their third-party libraries are used or updated in client projects. Therefore, a *quantitative* and *holistic* study on usages, updates and risks of third-party libraries in open-source projects can provide concrete evidence on these problems, and practical insights to improve the ecosystem sustainably. In this paper, we make the first contribution towards such a study in the Java ecosystem. First, using 806 open-source projects and 13,565 third-party libraries, we conduct a *library usage analysis* (e.g., usage intensity and usage outdatedness), followed by a *library update analysis* (e.g., update intensity and update delay). The two analyses aim to quantify usage and update practices from the two holistic perspectives of open-source projects and third-party libraries. Then, we carry out a *library risk analysis* (e.g., usage risk and update risk) on 806 open-source projects and 544 security bugs. This analysis aims to quantify the potential risk of using and updating outdated third-party libraries with respect to security bugs. Our findings suggest practical implications to developers and researchers on problems and potential solutions in maintaining third-party libraries (e.g., smart alerting and automated updating of outdated third-party libraries). To demonstrate the usefulness of our findings, we propose a security bug-driven alerting system, named LibSecurify, for assisting developers to make confident decisions by quantifying risks and effort when updating outdated third-party libraries. 33 open-source projects have confirmed the presence of security bugs after receiving our alerts, and 24 of those 33 have updated their third-party libraries. We have released our dataset to foster valuable applications and improve the Java third-party library ecosystem.

Communicated by: Zhenchang Xing and Kelly Blincoe

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

✉ Bihuan Chen
bhchen@fudan.edu.cn

Extended author information available on the last page of the article.

Keywords Library usages · Library updates · Library risks · Security bugs

1 Introduction

Third-party libraries play a key role in software development because they allow developers to reuse common functionalities instead of reinventing the wheel and substantially improve the productivity of developers. In contrast to the benefits that third-party libraries bring to software development, some costs arise due to the asynchronous development and evolution between third-party libraries and client projects. From the perspective of client projects, outdated third-party libraries can be commonly used but seldom updated, or updated in a delayed way. Developers need to invest a tremendous amount of costs in software maintenance to keep third-party libraries up-to-date. As old third-party library versions inevitably contain bugs (e.g., functional bugs and security bugs), they might cause crashes or increase attack surfaces in client projects. As new third-party library versions refactor code, fix bugs and add features, they might break library APIs (Kim et al. 2011; Bogart et al. 2016). Even worse, developers lack effective mechanisms to be aware of these potential risks in using and updating third-party libraries. From the perspective of third-party libraries, developers lack effective channels to learn about the usages and updates of their third-party libraries in client projects. As a result, such information fails to be fed back to the library development cycle to improve their design. Therefore, the usages and updates of third-party libraries are a double-edged sword, demanding a thorough assessment of benefits and costs.

To provide concrete and comprehensive evidence on these problems, a *quantitative* and *holistic* study on usages, updates and risks of third-party libraries in open-source projects is needed. On one hand, the study should define metrics to quantify usages, updates and risks such that the severity of the problems can be concretely revealed. On the other hand, the study should take the perspective of all involved parties (i.e., open-source projects and third-party libraries) such that a holistic view can be characterized for the ecosystem. Although several studies have been proposed in the Java ecosystem, none of them can contribute such a study. For example, some studies measured the usage popularity of third-party libraries at different granularities (e.g., versions (Mileva et al. 2009; Kula et al. 2017), classes (Mileva et al. 2010; De Roover et al. 2013; Hora and Valente 2015) and methods (Lämmel et al. 2011; Qiu et al. 2016)); some studies investigated the usages of outdated third-party libraries (Kula et al. 2015, 2018b); and some studies explored the reasons for updating or not updating third-party libraries (Bavota et al. 2015; Kula et al. 2018b). However, they only analyze the usages or updates mostly from the perspective of third-party libraries; and fail to characterize the severity of and the risks in using and updating third-party libraries, e.g., outdatedness of used third-party libraries, delay when updating third-party libraries, security bugs in used third-party libraries, and incompatible library methods when updating third-party libraries. This situation hides problems in maintaining third-party libraries and hinders practical solutions to potential problems.

To improve on such situation sustainably, this paper makes the first contribution to quantitatively and holistically characterize usages, updates and risks of third-party libraries in open-source projects in the Java ecosystem by answering the following three research questions:

– **RQ1: Library Usage Analysis.** What is the usage intensity and usage outdatedness of third-party libraries?

- **RQ2: Library Update Analysis.** What is the update intensity and update delay of third-party libraries?
- **RQ3: Library Risk Analysis.** What is the potential risk in using and updating outdated third-party libraries?

Library usage analysis and *library update analysis* are conducted on 806 Java open-source projects and 13,565 third-party libraries. These two analyses aim to quantify usage and update practices from the two holistic perspectives of open-source projects and third-party libraries. *Library risk analysis* is conducted on the 806 Java open-source projects and 544 security bugs in the 13,565 third-party libraries. This analysis aims to quantify the potential risk of using and updating outdated third-party libraries with respect to security bugs from the two holistic perspectives of open-source projects and third-party libraries. These analyses are conducted based on formulated metrics to allow quantification and reproduction. The data crawling and analyses in this study takes four months on a desktop machine.

Through these analyses, we aim to provide useful findings to both developers and researchers. For example, 49.8% of projects adopt more than 20 libraries. 33.0% of projects have more than 20% of methods calling library APIs. 60.0% of libraries have at most 2% of their APIs called across projects. 37.2% of projects adopt multiple versions of the same library in different modules. Around one-third of the projects have a lag of one major version from the latest library version, and around 80% and 70% of the projects have a lag of at least one minor and patch version although minor and patch versions are supposed to be compatible. 54.9% of projects leave more than half of the library dependencies never updated. 50.5% of projects have an update delay of more than 60 days. 68.8% of library releases contain security bugs. 35.3% of buggy library releases have over 300 APIs deleted in the safe release. Our findings help to uncover problems in maintaining third-party libraries, quantify the significance of these problems to raise attention in the ecosystem, and enable follow-up research to address the problems; e.g., ecosystem-level knowledge graph, library debloat, library selection, library recommendation, library harmonization, and smart alerting and automated updating of outdated libraries.

To demonstrate the usefulness of our findings, we propose a security bug-driven alerting system, named LibSecurify, to provide fine-grained information for developers to make confident decisions about third-party library version updates. In particular, LibSecurify consists of two major steps that respectively analyze the risk in used library versions (i.e., whether the security bugs in the used library versions are in the execution path of a client project) as well as the effort in updating to safe library versions (i.e., whether the called APIs in the used library versions are deleted or modified in the safe library versions). Our evaluation shows that 89.6% of the 451 open-source projects that adopt buggy third-party library versions can be safe. For the 38 unsafe open-source projects, we quantify the risk and updating effort. 33 open-source projects have confirmed the presence of security bugs after receiving our alerts, and 24 of those 33 have updated their third-party libraries.

This work is a substantial extended version of our previous work (Wang et al. 2020). Specifically, we extend the previous work in the following aspects. First, we extend the usage intensity analysis by including a coarse-grained library level analysis. Second, we refine the usage outdatedness analysis with respect to two dimensions, i.e., usage outdatedness in terms of days, and the type of library releases in terms of semantic versioning (Preston-Werner 2013). Third, we add a detailed description of our proposed security bug-driven alerting system, and add a performance evaluation. Fourth, we add a sampling-based comparison of the domains of the intensively-used and not intensively-used libraries

as well as the most outdated and least outdated libraries. Last, we provide a more detailed discussion of related work, implications, and threats to validity.

In summary, this paper makes the following contributions:

- We conducted large-scale analyses to quantitatively and holistically characterize usages, updates and risks of third-party libraries in Java open-source projects.
- We provided practical implications to developers and researchers, and released our dataset to foster valuable applications.
- We proposed a security bug-driven altering system to assist developers for making confident decisions about third-party library version updates.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 introduces our study design. Sections 4, 5 and 6 respectively report the findings for our library usage analysis, library update analysis and library risk analysis. Section 7 discusses the implications and applications of our findings, and the threats to our study. Section 8 draws the conclusions.

2 Related Work

We review the closely related work in five aspects, i.e, usage analysis, update analysis, risk analysis, evolution and adaptation, and recommendation and migration.

2.1 Usage Analysis

Library usage analysis has been widely explored to inspire developers and be helpful for the community. Some researchers analyzed library usage trend and popularity in terms of library versions (Mileva et al. 2009; Kula et al. 2017). Taking a step further, some researchers explored usage trend and popularity of library API elements (e.g., classes and interfaces), either by mining *import* statements from source code (Mileva et al. 2010; Hora and Valente 2015) or by parsing source code into AST (De Roover et al. 2013). These approaches report coarse-grained library usage mostly for one specific library, but do not report aggregated results across all libraries. Instead, we further present method-level usage analysis for projects and libraries at the ecosystem level.

To achieve library usage analysis at a more fine-grained level, Bauer et al. (2012) and Bauer and Heinemann (2012) extracted used library APIs in a project. Zaimi et al. (2015) analyzed the number of used library versions and classes for a project. They report library usage for one project, but not across a corpus of projects. Lämmel et al. (2011) conducted library usage analysis at the method level only for one specific library or project, but did not measure the aggregated results across a spectrum of libraries and projects. Qiu et al. (2016) also studied method-level library usage but only from the perspective of libraries. However, they analyzed usage intensity only for JDK library, and reported that 41.2% of the methods and 41.6% of the fields are never adopted by any project. This situation is much more severe in third-party libraries as JDK library is used by each project.

In summary, existing library usage analysis for Java ecosystem provides partial facets about library usage. To the best of our knowledge, we are the first to *holistically* analyze library usage at a fine-grained level for both projects and libraries.

Kula et al. (2015) studied the adoption of latest library versions when developers introduced libraries, and found that 82% of projects used the latest version. Based on our usage outdatedness analysis, it seems developers seldom update libraries after introduction. Cox

et al. (2015) introduced three metrics to define the dependency freshness at the dependency and project level. We use one of the metrics to quantify usage outdatedness.

Apart from Java ecosystem, studies have been conducted for npm and Android ecosystems. Wittern et al. (2016) analyzed the popularity of npm packages and the adoption of semantic versioning in npm packages. Abdalkareem et al. (2017) studied reasons and drawbacks of using trivial npm packages. For Android apps, library code is shipped into APK files, and library detection approaches (Ma et al. 2016; Li et al. 2017; Backes et al. 2016; Zhang et al. 2018) have been developed. Li et al. (2016) analyzed the popularity of mobile libraries. It is interesting to conduct fine-grained library usage in these ecosystems.

2.2 Update Analysis

It is useful to explore the intentions of developers on when and why they update libraries. Bavota et al. (2013, 2015) found that a high number of bug fixes could encourage dependency updates, but API changes could discourage dependency updates. Fujibayashi et al. (2017) explored the relationship between library release cycle and library version updates. Kula et al. (2018b) analyzed the practice of library updates, and found that developers rarely updated libraries. They also conducted eight manual case studies to understand developer's responsiveness to new library releases and security advisories, and found that developers were not likely to respond to security advisories mostly due to the unawareness of vulnerable libraries. Different from these studies, we quantify update intensity, update delay and update risk from the perspective of projects and libraries.

Apart from Java ecosystem, library update analysis has been conducted for other ecosystems. Derr et al. (2017) studied why developers updated mobile libraries, analyzed the practice of semantic versioning, and conducted a library updatability analysis. Salza et al. (2018) analyzed mobile library categories that were more likely to be updated, and identified six update patterns. Lauinger et al. (2017) and Zerouali et al. (2018) measured the time lag of an outdated npm package from its latest release, and Decan et al. (2018a) analyzed the evolution of this time lag. Decan et al. (2017) also compared problems and solutions of library releases in three ecosystems, and found that the problems and solutions varied from one to another, and depended both on the policies and the technical aspects of each ecosystem.

2.3 Risk Analysis

Risk could be introduced into a project through the dependency network. One potential risk is dependency conflict. Patra et al. (2018) analyzed JavaScript library conflicts caused by the lack of namespaces in JavaScript, and proposed a dynamic analysis approach to first identify potential conflicts and then validate them. Wang et al. (2018) analyzed manifestation and fixing patterns of dependency conflicts in Java, and proposed DECCA to detect dependency conflicts and RIDDLE (Wang et al. 2019b) to generate crashing stack traces for the detected dependency conflicts. These approaches are focused on library version inconsistencies that might have harmful consequence (e.g., bugs). However, as reported by Huang et al. (2020), harmful consequence is one of the commonest reasons for fixing library version inconsistencies, and the other reasons are to avoid maintenance efforts in the long run and to ensure consistent library API behaviors across modules. Therefore, Huang et al. (2020) proposed LIBHARMO to detect library version inconsistencies. In this study, we focus the risk of security bugs.

Another potential risk is security bugs on the dependency network (Decan et al. 2018b) and potential threats on indirectly introduced libraries and untrusted maintainers (Zimmermann et al. 2019). We focus on security bugs in Java libraries, and take a different perspective than theirs, i.e., considering client projects that use buggy library versions and measuring usage and update risk at the method level. Dietrich et al. (2014) analyzed update risk with 109 Java programs and 212 dependencies, and found that 75% of version updates are not compatible. Differently, we actually quantify such incompatibilities. Liu et al. (2021) demystified the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem.

Alerting system has been proposed to notify developers about security bugs in libraries. Cadariu et al. (2015) proposed an alerting system to report Java library dependencies that have security bugs. Mirhosseini and Parnin (2017) studied the usage of pull requests and badges to notify outdated npm packages. Nguyen et al. (2020) proposed Up2DEP, an alerting and update assistance system in Android ecosystem to warn developers about vulnerable library versions and help to update library dependencies with library API compatibility results. Such alerting systems, similar to some existing tools like OWASP,[1] Snyk[2] and Dependabot,[3] are coarse-grained as they do not analyze whether security bugs in library versions really affect a project. This was evidenced in a recent study (Zapata et al. 2018).

To mitigate the above problem, some advances have been proposed to analyze whether security bugs in libraries are truly in the execution path of a project. Hejderup et al. (2018) constructed a versioned ecosystem-level call graph, and checked whether a security bug can be reached through the call graph. Plate et al. (2015) used dynamic analysis to check whether the methods that were changed to fix security bugs were executed by a project. Then, Ponta et al. (2018) extended Plate et al.'s work Plate et al. (2015) by combining static analysis to partially mitigate the test coverage problem. However, none of them is open-sourced. Except for Ponta et al. (2018), they only notify developers about security bugs, but leave developers unaware of potential risk (or effort) to update buggy library versions. The risk analysis in Ponta et al. (2018) only reports calls to library APIs that are deleted in the new version. Instead, we conduct fine-grained change analysis on library APIs by considering their call graphs.

2.4 Evolution and Adaptation

Many studies have been conducted on API evolution, e.g., impact of refactoring on API breaking (Dig and Johnson 2006; Kim et al. 2011; Kula et al. 2018c), developers' reaction to API evolution (Robbes et al. 2012; Hora et al. 2015; Sawant et al. 2016), API stability (Raemaekers et al. 2012; McDonnell et al. 2013; Linares-Vásquez et al. 2013), types of API changes and usages (Wu et al. 2016), adoption of semantic versioning to avoid API breaking (Raemaekers et al. 2014), and API breaking in different ecosystems (Bogart et al. 2016).

On the other hand, a number of methods have been proposed to adapt to API evolution by change rules written by developers (Chow and Notkin 1996; Balaban et al. 2005), recorded from developers (Henkel and Diwan 2005), derived by similarity matching (Xing and Stroulia 2007), mined from API usage in own libraries (Dagenais and Robillard 2009; 2011), mined from API usage in projects (Schäfer et al. 2008; Nguyen et al. 2010), and

[1] https://owasp.org/www-project-dependency-check/
[2] https://snyk.io
[3] https://dependabot.com

identified by a combination of some of these methods (Wu et al. 2010). Empirical studies have been conducted to compare these methods (Cossette and Walker 2012; Wu et al. 2015). Currently, our security bug-driven alerting system only suggests safe library versions, but does not support code-level automatic library updates. Huang et al. (2021) recently proposed RepFinder to automatically find replacement APIs for missing APIs in library update from multiple sources. These API evolution and adaptation approaches are a good starting point for achieving automatic library updates.

2.5 Recommendation and Migration

Several approaches have been proposed for library recommendation and migration, e.g., recommending libraries (Thung et al. 2013a; Ouni et al. 2017), recommending library APIs (Chan et al. 2012; Thung et al. 2013b), recommending libraries or library APIs across different programming languages (Zheng et al. 2011; Chen and Xing 2016), and migration across similar libraries (Teyton et al. 2012, 2013, 2014; Kabinna et al. 2016). However, they do not target the recommendation of and migration between versions of the same library, which instead are useful for achieving automated library updates.

3 Empirical Study Methodology

In this section, we first introduce the design of our empirical study, and then present our corpus selection process.

3.1 Study Design

Our study aims to characterize usages, updates and risks of third-party libraries in Java open-source projects. To this end, we design the three RQs as introduced in Section 1. For the ease of presentation, hereafter we refer to *third-party library* as *library* and *Java open-source project* as *project*. Before elaborating the RQs, we define library terms to avoid confusion. A *library version* is a library with the version number. A *library release* is a library version with the release information (e.g., release date). A *library dependency* is a library version declared as a dependency in a project.

Our library usage analysis in RQ1 systematically analyzes the currently used libraries in projects. It first measures how intensively a project depends on libraries (i.e., usage intensity from the perspective of projects) and how intensively a library is used across projects (i.e., usage intensity from the perspective of libraries). It aims to quantify the significance of using libraries in project development and the impact of evolving libraries. Then, it investigates how far the adopted library versions are away from the latest versions (i.e., usage outdatedness) from the perspective of projects and libraries. It aims to quantify the commonness and severity of adopting outdated libraries in projects and motivate the necessity of RQ2.

Our library update analysis in RQ2 systematically analyzes the historical library version updates in projects. It first explores how intensively a project updates library versions (i.e., update intensity from the perspective of projects) and how intensively a library's versions are updated across projects (i.e., update intensity from the perspective of libraries). It aims to quantify the practices of updating library versions. Then, it measures how long library version updates lag behind library releases (i.e., update delay) from the perspective of projects

and libraries. It aims to quantify the developers' reaction time to new library releases and motivate the necessity of RQ3.

Our library risk analysis in RQ3 systematically investigates the security bugs in libraries. It first measures how many buggy library versions a project uses (i.e., usage risk from the perspective of projects) and how many security bugs exist in a library release (i.e., usage risk from the perspective of libraries). It aims to quantify the risk in using outdated libraries and delaying library version updates with respect to security bugs. Then, it measures how many library APIs in buggy library versions a project calls (i.e., update risk from the perspective of projects) and how many library APIs in a buggy library release differs from a safe library release (i.e., update risk from the perspective of libraries). It aims to quantify the risk in updating buggy libraries in terms of potential API incompatibilities.

3.2 Corpus Selection

We conducted this study on a corpus of Java open-source projects selected from GitHub. We focused on Java because it is widely-used and hence our findings can be beneficial to a wider audience. Specifically, we first selected non-forked popular Java projects that had more than 200 stars, which resulted in an initial set of 2,216 projects. Of these projects, we selected projects that used Maven or Gradle as the automated build tool in order to ease the extraction of declared library dependencies in projects, which restricted our selection to a set of 1,828 projects. We picked active projects that had commits in the last three months with the intention to prefer a local generality of our findings at the cost of a global generality. Audiences from active projects can be more actionable given our findings, while including inactive projects would generate less representative findings for them. Finally, we had 806 projects, denoted as $\mathcal{P}$. Among these 806 projects, 10 projects only include tutorials that provide examples or samples. We also conducted a quantitative analysis on these 806 projects with respect to four dimensions, i.e., the number of commits in a project, the number of contributors in a project, the lines of code (i.e., LOC) in a project, and the years of project development. We adopted cloc[4] to measure the LOC in a project, exclusive of code comments. We report the results in boxplots in Fig. 1. We crawled their repositories and commits on master branch from GitHub using a desktop with 2.29 GHz Intel Core i5 CPU and 8 GB RAM. We conducted library crawling and library analyses on the same desktop, which took four months.

4 Library Usage Analysis

To study library usages, we develop *lib-extractor* to extract library dependencies from each project's configuration files (i.e., *pom.xml* and *build.gradle* for Maven and Gradle projects) in a commit. Maven and Gradle support various mechanisms (e.g., inheritance, version range and variable expansion) to declare library dependencies, and we support them in *lib-extractor*. Basically, for Maven projects, *lib-extractor* extracts a library dependency via parsing three fields: *groupId*, *artifactId* and *version*; and for Gradle projects, it extracts a library dependency via parsing similar fields: *group*, *name* and *version*.

A library dependency d is denoted as a 4-tuple $\langle p, f, com, v \rangle$, where p and f denote the project and configuration file where d is declared (here $p.date$ denotes the date when

[4] https://github.com/AlDanial/cloc

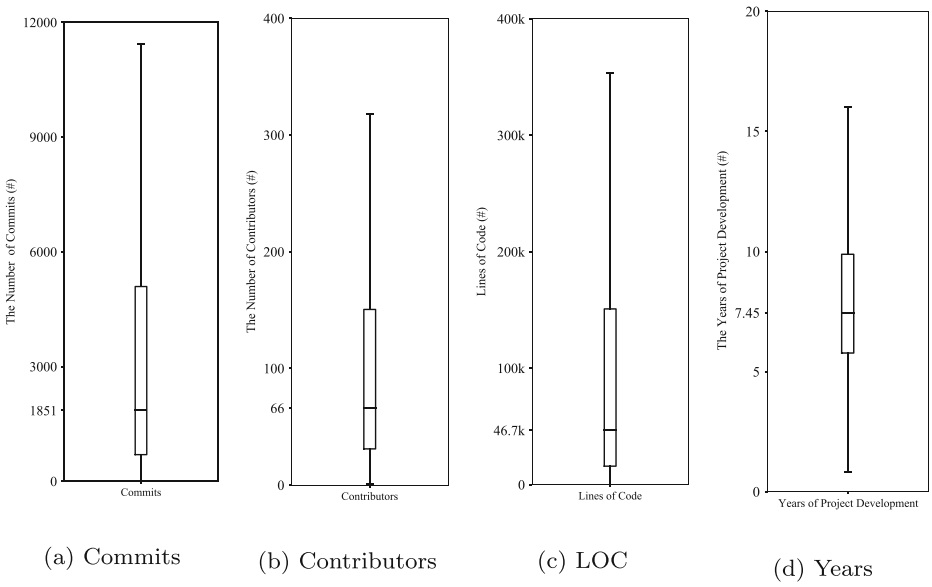


Fig. 1 Project statistics

p's repository is crawled), com denotes the commit where d is extracted (here $com.date$ denotes the submission date of com), and v denotes the library version declared in d. v is denoted as a 2-tuple $\langle l, ver \rangle$, where l denotes a library, and ver denotes a version number of l. l is denoted as a 2-tuple $\langle group, name \rangle$, where $group$ and $name$ denote l's organization and name.

As RQ1 targeted libraries currently used in projects, we ran *lib-extractor* on the latest commit of each project in $\mathcal{P}$, and obtained 164,470 library dependencies, denoted as $\mathcal{D}_{us}$, 24,205 library versions, denoted as $\mathcal{V}_{us} = \{d.v \mid d \in \mathcal{D}_{us}\}$, and 13,565 libraries, denoted as $\mathcal{L}_{us} = \{v.l \mid v \in \mathcal{V}_{us}\}$.

4.1 Library-Level Usage Intensity

Definition We define usage intensity at a coarse-grained (i.e., library) level from the perspective of a project and a library: usi_p^1, the number of libraries that are adopted in a project p, and usi_l^1, the number of projects that adopt a library l. Using $\mathcal{P}$, $\mathcal{L}_{us}$ and $\mathcal{D}_{us}$, we compute usi_p^1 and usi_l^1 by (1).

$$\begin{aligned} \forall p \in \mathcal{P}, usi_p^1 &= |\{d.v.l \mid d \in \mathcal{D}_{us} \wedge d.p = p\}| \\ \forall l \in \mathcal{L}_{us}, usi_l^1 &= |\{d.p \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\}| \end{aligned} \tag{1}$$

Findings Using usi_p^1 and usi_l^1, we show distributions of usage intensity across projects and libraries in Figs. 2 and 3, where the y-axis respectively represents the number of projects and libraries whose usage intensity falls into a range. On one hand, 28 (3.5%) projects do not adopt libraries. The domains of these 28 projects vary from algorithms to basic tools (e.g. decompiler, and toolkit for easy access to native Java shared libraries). 224 (27.8%) projects use at most 10 libraries. 401 (49.8%), 184 (22.8%) and 80 (9.9%) projects respectively adopt

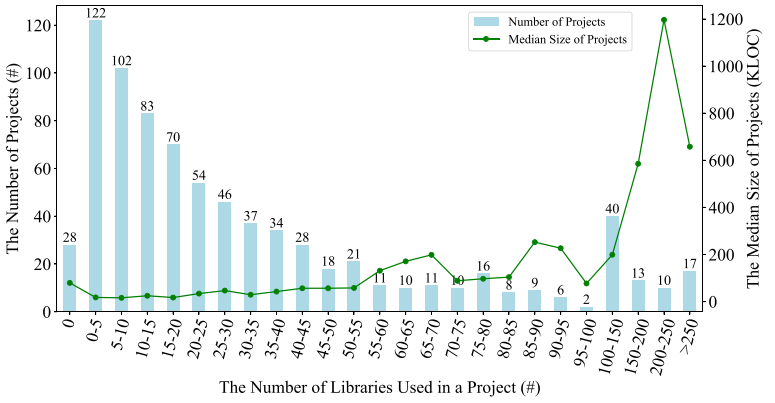


Fig. 2 Library-level usage intensity across projects

more than 20, 50 and 100 libraries. On the other hand, 10,499 (77.4%) libraries are used in only one project; and only 432 (3.2%) and 214 (1.6%) libraries are respectively adopted in more than 10 and 20 projects.

> ***Summary.*** Projects often have a moderate dependency on libraries. Only a very small portion of libraries are widely adopted across projects. Library recommendation is needed to help project developers catch reuse opportunities.

Surprisingly, 77.4% libraries are used in only one project. To understand the characteristics about these libraries, we sample 371 libraries. The sample size allows the generalization of our results at a confidence level of 95% and a margin of error of 5%, computed by a sample size calculator.[5] 192 (51.8%) of the 371 libraries are from the Maven central repository, while the other 179 libraries are from other third-party repository sites (e.g., openhab.jfrog.io and oss.sonatype.org) that are not widely used. For 177 of the 179 libraries from other third-party repository sites, their snapshot versions[6] are used. This explains why they are only used once. On the other hand, of the 192 libraries from the Maven central repository, 71 libraries are used by less than 100 libraries in the Maven central repository, and 121 libraries are used by more than 100 libraries.[7] Such a difference from our empirical result is due to two main reasons. First, our study only focuses on direct library dependencies and only includes 806 projects. Therefore, some libraries could be used as indirect library dependencies or used in other projects, but are not counted in our study. Second, if a library is used in multiple versions of a library, its usage will be counted by multiple times in the Maven central repository. However, we only analyze one version of a project.

Intuitively, the number of libraries adopted in a project is correlated to the size of the project as large projects can be complex and have high needs for libraries. To explore this conjecture, we measure the project size in thousands of lines of code (KLOC), exclusive of code comments, using cloc, and report the median size of the projects in each bar in Fig. 2.

[5] https://www.surveysystem.com/sscalc.htm

[6] a.k.a. changing versions whose features are under active development but are allowed for developers to integrate before stable versions are released.

[7] The usage count can be obtained from the "Used by" field in the Maven central repository.

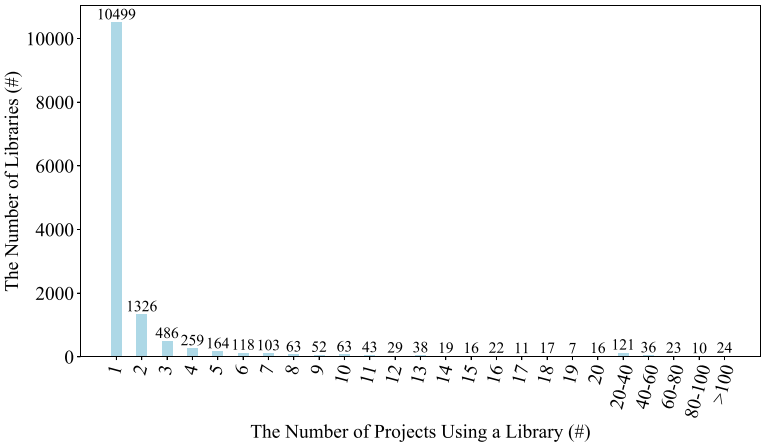


Fig. 3 Library-level usage intensity across libraries

Overall, projects using a large number of libraries are larger in size than projects using a small number of libraries. Such a difference is statistically significant (i.e., $p = 0.00003$ in one-way ANOVA test (Howell 2012)). In an extreme case, *Apache Camel* adopts 1,290 libraries, and its size is 1,326 KLOC.

To understand the library characteristics affecting usage intensity, we analyze library categories. Unfortunately, only one-fifth of the libraries in $\mathcal{L}_{us}$ (in Fig. 3) have their category specified on library repositories. Hence, we directly choose the 50 most popular libraries (as the number of popular libraries is relatively small), but sample 371 libraries from 10,499 libraries used by one project. Then, three of the authors follow an open coding procedure (Khandkar 2009) to manually categorize libraries based on the category list from Maven. Among the 50 most popular libraries, 33 are general-purpose libraries (e.g., 12 testing libraries, 10 utility libraries and 8 logging libraries) that provide functions of general interest, while 17 are domain-specific libraries (e.g., 6 web libraries and 4 database libraries) that belong to popular domains. Among the 371 unpopular libraries, only 53 share the same 17 (out of 24) categories to those 50 popular ones, and are mostly functionally-similar libraries; and the others mostly provide specific function (e.g., 129 Eclipse plugin libraries) that is only of interest for certain projects across 89 categories.

Summary. The number of libraries adopted in a project is correlated to the project size. In that sense, library maintenance becomes a non-trivial task in large projects as it is difficult for project developers to have a clear vision of the libraries used in the large code base. As multiple libraries share the same category and some of them are widely adopted but some of them are rarely adopted, library selection is needed to help developers select suitable libraries.

4.2 Method-Level Usage Intensity

Definition We further define usage intensity at a fine-grained (i.e., method) level from the perspective of a project and library: usi_p^2, the percent of a project p's methods that call library APIs, and usi_l^2, the percent of a library l's APIs that are called across projects. The

most closest work is from Qiu et al. (2016), but only considers JDK libraries. To compute usi_p^2 and usi_l^2, we need to extract library APIs, project methods, and API calls in project methods.

Therefore, we first crawled the jar file of each library version $v \in \mathcal{V}_{us}$ from library repositories (e.g., Maven and Sonatype) declared in configuration files. We successfully crawled jar files for 16,384 library versions, denoted as $\tilde{\mathcal{V}}_{us}$, but failed for the other 7,821 (32.3%) library versions. The main reason is snapshot versions (76.1%) whose jar files are no longer available; and the other reasons are very old library versions that are no longer available and private libraries that we do not have permissions to access. From $\tilde{\mathcal{V}}_{us}$, we identified 7,229 libraries, denoted as $\tilde{\mathcal{L}}_{us}$.

Then, we used Soot (Vallée-Rai et al. 1999) on the jar files for $\tilde{\mathcal{V}}_{us}$ to extract library APIs, denoted as $\mathcal{A}$. Each library API $a \in \mathcal{A}$ is denoted as a 2-tuple $\langle v, api \rangle$, where $v \in \tilde{\mathcal{V}}_{us}$ denotes a library version, and api denotes a library API. Here, we conservatively treat public methods and fields in public classes as library APIs. Next, we used JavaParser (Smith et al. 2017) with type binding on project repositories and jar files of the used library versions to extract project methods, denoted as $\mathcal{M}$, and API calls in project methods, denoted as $\mathcal{C}$. Each project method $m \in \mathcal{M}$ is denoted as a 2-tuple $\langle p, method \rangle$, where p denotes a project, and $method$ denotes a method in p. Each API call $c \in \mathcal{C}$ is denoted as a 2-tuple $\langle a, m \rangle$, where $a \in \mathcal{A}$ denotes a library API, and $m \in \mathcal{M}$ denotes the project method where a is called.

Using $\mathcal{P}$, $\tilde{\mathcal{V}}_{us}$, $\tilde{\mathcal{L}}_{us}$, $\mathcal{A}$, $\mathcal{M}$ and $\mathcal{C}$, we compute usi_p^2 and usi_l^2 by (2). Notice that $\mathcal{V}_l = \{v \mid v \in \tilde{\mathcal{V}}_{us} \wedge v.l = l\}$ denotes l's used versions, and usi_l takes their maximum usage intensity.

$$\begin{aligned} \forall p \in \mathcal{P}, usi_p^2 &= \frac{|\{c.m \mid c \in \mathcal{C} \wedge c.m.p = p\}|}{|\{m \mid m \in \mathcal{M} \wedge m.p = p\}|} \\ \forall l \in \tilde{\mathcal{L}}_{us}, usi_l^2 &= \max_{v \in \mathcal{V}_l} \frac{|\{c.a \mid c \in \mathcal{C} \wedge c.a.v = v\}|}{|\{a \mid a \in \mathcal{A} \wedge a.v = v\}|} \end{aligned} \tag{2}$$

Findings Using usi_p and usi_l, we show distributions of usage intensity across projects and libraries in Figs. 4 and 5, where the y-axis respectively denotes the number of projects and libraries whose usage intensity falls into a range. On one hand, 74 (9.2%) projects do not

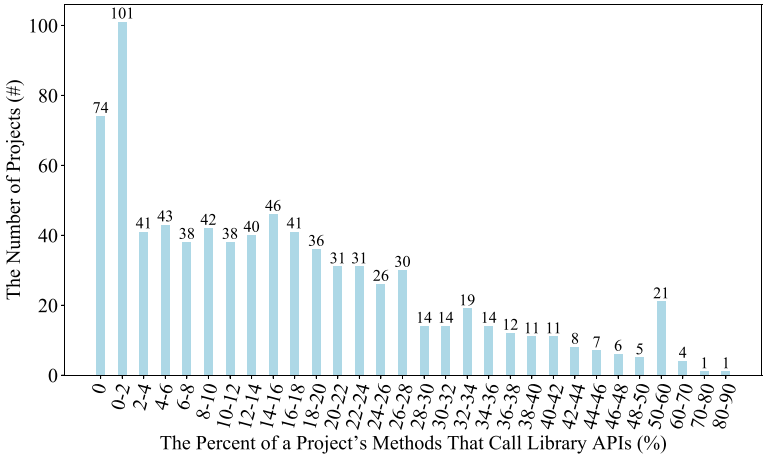


Fig. 4 Method-level usage intensity across projects

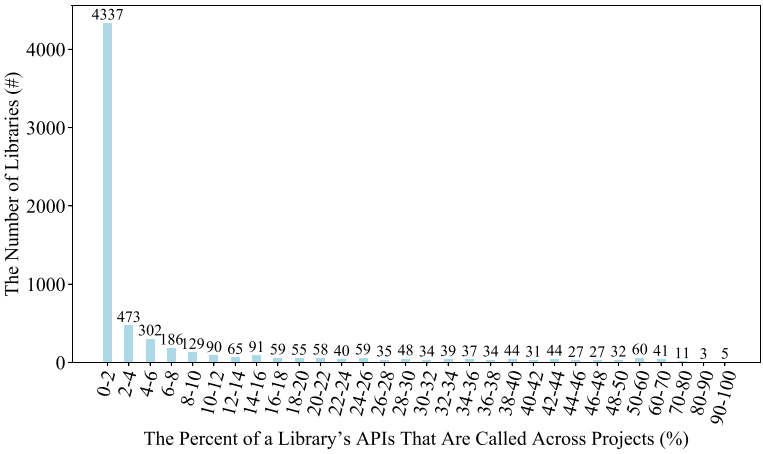


Fig. 5 Method-level usage intensity across libraries

call library APIs; i.e., 28 projects do not use libraries, 5 projects use library versions that are unavailable, and 41 projects only use the resource files in jar files. 265 (32.9%) projects have at most 10% of methods calling library APIs. 266 (33.0%) and 64 (7.9%) projects have more than 20% and 40% of methods that call library APIs, respectively. On the other hand, 4,337 (60.0%) libraries have at most 2% of their APIs called across projects; and only 281 (3.9%) libraries have more than 40% of their APIs called across projects. Notice that 733 libraries do not have class files, but only have resource files in the jar files (e.g., *Angular* only contains web assets), and are not included in Fig. 5.

We follow the same procedure as in Section 4.1 to analyze the categories of 50 libraries directly taken from the tail of Fig. 5 and 353 libraries statistically sampled from the first bar in Fig. 5. The 50 libraries mostly provide very domain-specific but relatively simple functions (e.g., XML processing, stream processing, and rule engine). On the opposite, the 353 libraries are mostly provide very domain-specific but complex functions (e.g., distributed processing libraries like *hadoop-hdfs*, and mocking libraries like *mockito-core*), require platform integration (e.g., Eclipse and Maven plugins like *confluence*), or contain a wide range of related functions (e.g., Java specification). Further, to better distinguish "complex" and "simple" libraries, we calculated the average number of classes, methods and fields in these 353 and 50 libraries. For the 353 libraries, the average number of classes, methods and fields are 3,372, 36,312 and 13,100. For the 50 libraries, the average number of classes, methods and fields are 1,234, 17,306, 7,402. We can see that "complex" libraries contain a larger number of classes, methods and field than "simple" libraries.

Interestingly, in Fig. 4, there are two projects that have more than 70% of their methods calling library APIs. One project is *jgit-cookbook*, which includes a collection of ready-to-run snippets which provide a quick start for building functionality using *jgit* (i.e., the Java Git implementation). Due to its project purpose, 90% of its methods call APIs of the library *jgit*. The other project is *symphony*, which is a modern community (e.g., forum and blog) system platform written in Java. It directly depends on 20 libraries. As *symphony* is a Web application, 77% of its methods call APIs from the library *latke*, which is a Java Web framework based on JSON. As a result, 55% of its methods call APIs from the library *json*.

Summary. Projects usually have a moderate dependency on library APIs. Such concrete usage dependency also reflects the required effort on library maintenance (e.g., library updates and migrations). Only a very small part of library APIs in most libraries are used. Library developers should utilize such usage statistics to guide API evolution, while project developers can tailor unused library features.

During our crawling, snapshot versions are the main reason for unavailable jar files. Thus, we explore the used library versions and find that 344 (42.7%) projects use snapshot versions. Therefore, we measure the severity of snapshot versions in terms of the number of snapshot versions used in a project. The result is reported in Fig. 6. Overall, 7,345 (30.3%) of the library versions in $\mathcal{V}_{us}$ are snapshot versions, and 5,951 (81.0%) of them are no longer available at library repositories. As shown in Fig. 6, 161 (20.0%) and 183 (22.7%) projects adopt at most and more than five snapshot versions, respectively.

Moreover, during our fine-grained analysis, we find that multiple versions of the same library are used in different modules of a project as different modules in a project can separately declare library dependencies to suit different needs and release schedules. Therefore, we analyze the severity of using multiple versions from two perspectives: the number of libraries whose multiple versions are used in a project, and the number of used multiple versions of the same library. The results are reported in Figs. 7 and 8, respectively. Overall, 300 (37.2%) projects adopt multiple versions of the same library in different modules. 84 (10.4%) and 57 (7.1%) projects respectively contain one and two libraries whose multiple versions are used. 84 (10.4%) projects contain more than five libraries whose multiple versions are adopted. Further, among the 2,032 cases of using multiple versions of the same library, 1,600 (78.7%) and 233 (11.5%) cases respectively involve two and three versions of the same library; and 95 (4.7%) cases use more than five versions of the same library.

Summary. Snapshot library versions and multiple versions of the same library are commonly used in one-third of the projects. They could increase maintenance cost in the long run due to incompatible APIs. The latter could even lead to dependency conflicts when modules are inter-dependent. Thus, tools are needed to better manage them.

4.3 Usage Outdatedness

Definition We define usage outdatedness of a library dependency d in terms of library releases, denoted as uso_d^r, as the number of library releases with a higher version number at the time of repository crawling. We also define usage outdatedness of a library d in terms of days, denoted as uso_d^d, as the maximum number of days between the release date of a higher library version and the repository crawling date. For each library $l \in \tilde{\mathcal{L}}_{us}$, we crawled version number and release date of l's all library releases from library repositories (e.g., Maven and Sonatype) declared in configuration files. We had 288,312 library releases, denoted as $\mathcal{R}_{us}$. Each library release $r \in \mathcal{R}_{us}$ is denoted as a 2-tuple $\langle v, date \rangle$, where v denotes a library version, and $date$ denotes v's release date. Using $\mathcal{D}_{us}$ and $\mathcal{R}_{us}$, we compute uso_d^r and uso_d^d by (3) and (4).

$$\forall d \in \mathcal{D}_{us}, uso_d^r = |\{r \mid r \in \mathcal{R}_{us} \wedge r.v.l = d.v.l \wedge r.v.ver > d.v.ver \wedge r.date < d.p.date\}| \quad (3)$$

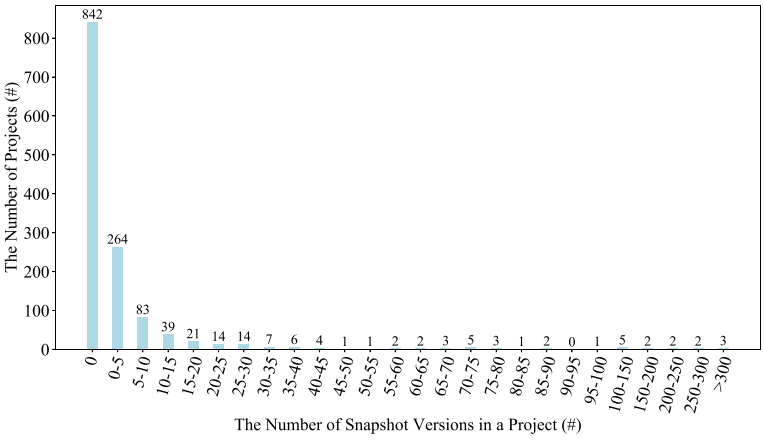


Fig. 6 Severity of using snapshot versions

$$\forall d \in \mathcal{D}_{us}, uso^d_d = max(\{r.date - d.p.date \mid r \in \mathcal{R}_{us} \wedge r.v.l = d.v.l \wedge r.v.ver > d.v.ver \wedge r.date < d.p.date\}) \quad (4)$$

Then, we define usage outdatedness from the perspective of a project and library: uso^r_p and uso^d_p, the average usage outdatedness of the library dependencies in a project p in terms of library releases and days, and uso^r_l and uso^d_l, the average usage outdatedness of the library dependencies on a library l in terms of library releases and days. Using $\mathcal{P}$, $\tilde{\mathcal{L}}_{us}$, $\mathcal{D}_{us}$, uso^r_d and uso^d_d, we get uso^r_p, uso^r_l, uso^d_p and uso^d_l by (5) and (6).

$$\begin{aligned} \forall p \in \mathcal{P}, uso^r_p &= avg_{d \in \mathcal{D}_p} uso^r_d, \mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\} \\ \forall l \in \tilde{\mathcal{L}}_{us}, uso^r_l &= avg_{d \in \mathcal{D}_l} uso^r_d, \mathcal{D}_l = \{d \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\} \end{aligned} \quad (5)$$

$$\begin{aligned} \forall p \in \mathcal{P}, uso^d_p &= avg_{d \in \mathcal{D}_p} uso^d_d, \mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\} \\ \forall l \in \tilde{\mathcal{L}}_{us}, uso^d_l &= avg_{d \in \mathcal{D}_l} uso^d_d, \mathcal{D}_l = \{d \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\} \end{aligned} \quad (6)$$

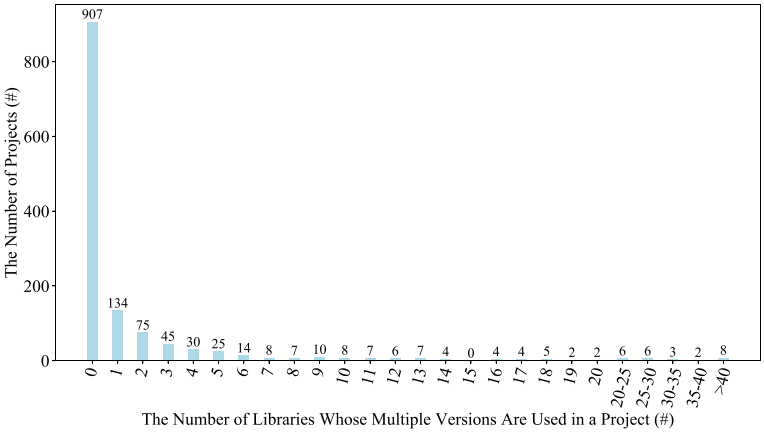


Fig. 7 Number of libraries of multiple versions

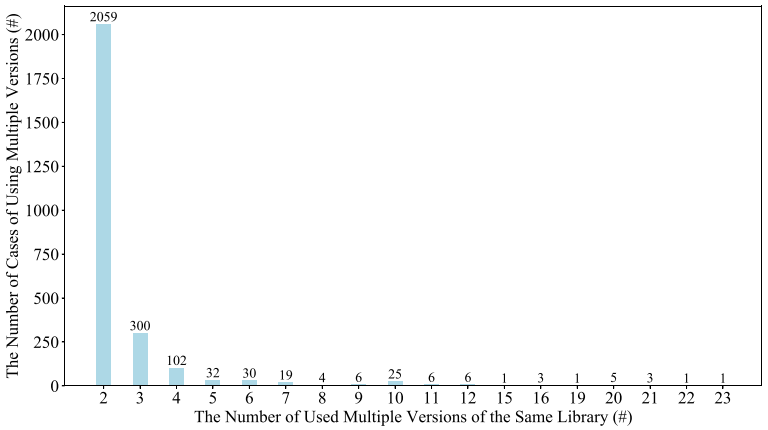


Fig. 8 Number of used multiple versions

Findings Using uso_p^r and uso_l^r, we report distributions of usage outdatedness in terms of library releases across projects and libraries in Figs. 9 and 10. On one hand, only 28 (3.5%) projects use the latest library versions. 83 (10.3%) projects use libraries that are averagely at most two versions away from the latest. 306 (38.0%), 118 (14.6%) and 19 (2.4%) projects adopt libraries that are averagely over 10, 20 and 50 versions away from the latest, respectively. 33 projects are not included in Fig. 9 as 28 projects do not adopt libraries, and the jar files of the library versions in 5 projects are all no longer available. On the other hand, in all the projects that use them, 3,269 (45.2%) libraries are already the latest, 1,419 (19.6%) libraries are averagely at most two versions away from the latest, and 1,025 (14.2%) and 134 (1.9%) libraries are averagely over 10 and 50 versions away from the latest, respectively. In an extreme case, `Confluence Core 4.0`, a very old version, is used in only one project, and `Confluence Core` has 2,374 version releases. As a result, it is 2,237 versions away from the latest version.

We follow the same procedure as in Section 4.1 to analyze the categories of 50 libraries directly taken from the tail of Fig. 10 and 344 libraries statistically sampled from the first bar in Fig. 10. Almost half of the 50 libraries are cloud computing libraries, which could need huge integration effort to switch to new versions; and 130 of the 344 libraries are Eclipse

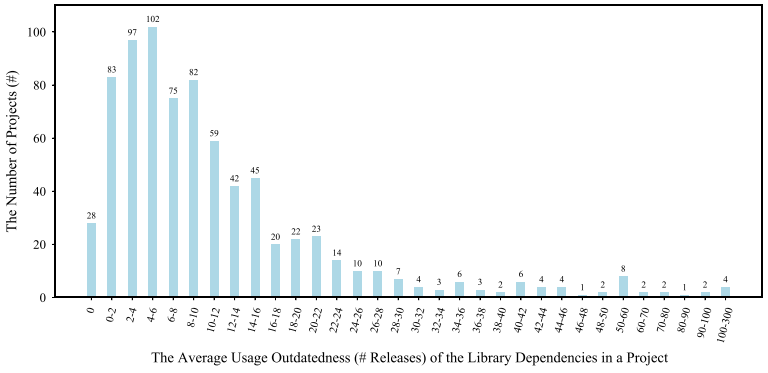


Fig. 9 Usage outdatedness (# releases) across projects

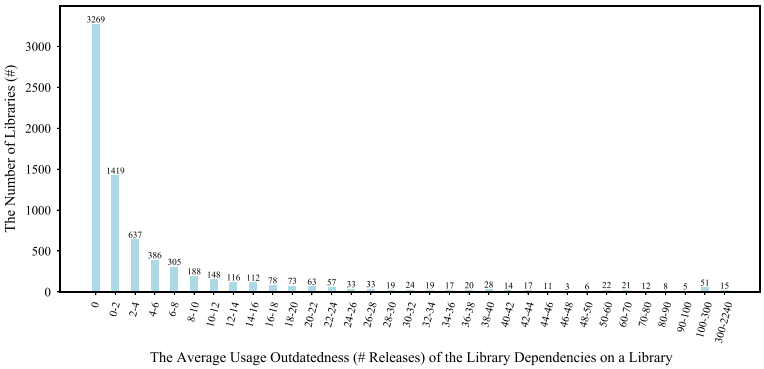


Fig. 10 Usage outdatedness (# releases) across libraries

plugin libraries, which have to be updated accordingly if developers want new features from and switch to new Eclipse versions.

Using uso_p^d and uso_l^d, we report distributions of usage outdatedness in terms of days across projects and libraries in Figs. 11 and 12. On one hand, 28 (3.5%) projects use the latest versions. 87 (10.8%) projects use libraries that are averagely at most 100 days before the release date of the latest version. 311 (38.6%) and 80 (9.9%) projects adopt libraries that are averagely over 500 and 1000 days before the release date of the latest version. On the other hand, in all the projects that use them, 3,269 (45.2%) libraries are already the latest, 1,282 (17.7%) libraries are averagely at most 100 days before the release date of the latest version and 1,231 (17.0%) and 459 (6.3%) libraries are averagely over 500 and 1000 days before the release date of the latest version.

Summary. Outdated library is nearly adopted in every project. The distance to the latest release is usually considerably large. Mechanisms are needed to make project developers aware of the risks (e.g., security bugs) of outdated libraries or the benefits (e.g., new features) of newer releases, while allowing library developers to directly notify such risks and benefits to the projects that adopt their library.

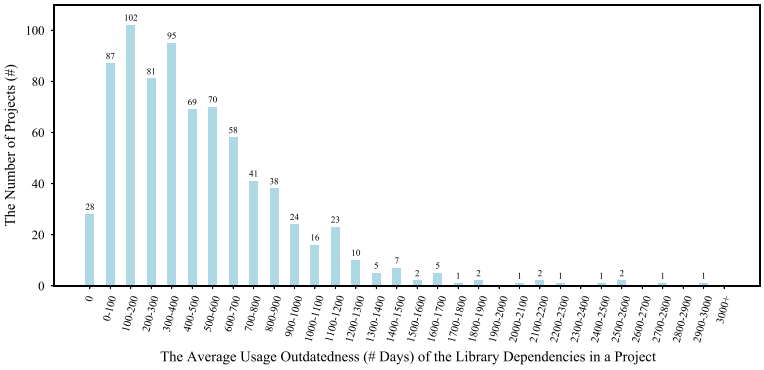


Fig. 11 Usage outdatedness (# days) across projects

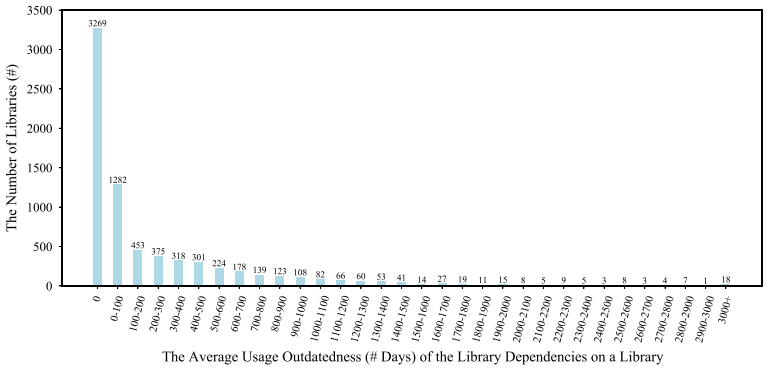
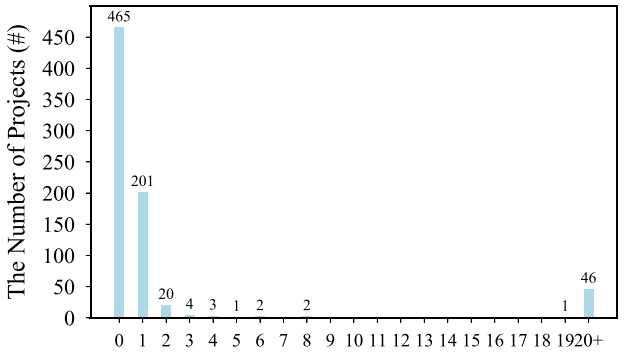


Fig. 12 Usage outdatedness (# days) across libraries

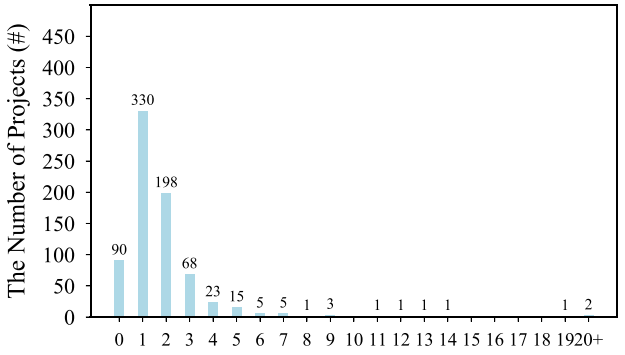
As defined by semantic versioning (Preston-Werner 2013), version numbers must take the form of $X.Y.Z$, where X, Y and Z is the major, minor and patch version. Bug fixes not affecting APIs increment Z, backwards compatible API changes or additions increment Y, and backwards incompatible API changes increment X. Generally, developers need no integration effort if updating to a patch or minor version, but need some integration effort if updating to a major version. Therefore, we re-compute uso_p^r, uso_l^r, uso_p^d and uso_l^d with respect to the type of library releases; i.e., we compute usage outdatedness by distinguishing major, minor and patch version outdatedness. Basically, we compute the number of library releases and the maximum day gap of library releases that have a higher major number as the major version-level usage outdatedness, the number of library releases and the maximum day gap of library releases that have a higher minor number while having the same major number as the minor version-level usage outdatedness, and the number of library releases and the maximum day gap of library releases that have a higher patch number while having the same major and minor numbers as the patch version-level usage outdatedness.

Figure 13 reports the distribution of usage outdatedness in terms of library releases across projects with respect to the three types of library releases. We can see that 465 (57.7%) projects share the same major version to the latest version, and 201 (24.9%) projects have a lag of one major version from the latest. This is potentially reasonable because major versions may introduce backwards incompatible API changes. However, only 90 (11.1%) projects share the same minor version to the lasted version that has the same major version. 655 (81.3%), 325 (40.3%) and 59 (7.3%) projects have a lag of at least one, two and four minor versions. Similarly, only 197 (24.4%) projects share the same patch version to the lasted version that has the same major and minor version. 548 (68.0%), 326 (40.4%) and 158 (19.6%) projects have a lag of at least one, two and four patch versions. Notice that 28 projects are not included in Fig. 13 because their declared library versions do not start with $X.Y$ or $X.Y.Z$. On the other hand, Fig. 14 reports the distribution of usage outdatedness in terms of library releases across libraries with respect to the three types of library releases. We can see that in all the projects that use them, 907 (12.5%) libraries have a lag of at least one major versions from the latest version. 2,359 (32.6%) and 1,264 (17.5%) libraries have a lag of at least one and two minor versions from the latest version that has the same major version. 1,828 (25.3%) and 1,088 (15.1%) libraries have a lag of at least one and two patch versions from the latest version that has the same major and minor version.

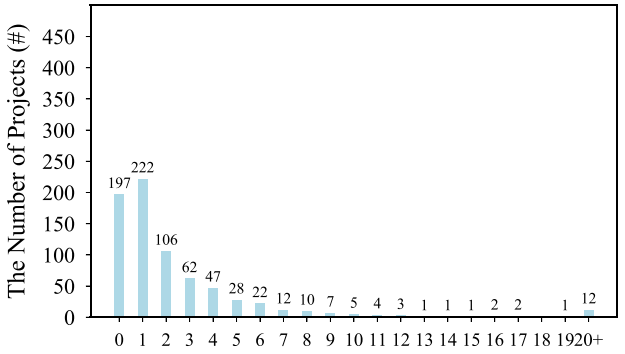
Figure 15 reports the distribution of usage outdatedness in terms of days across projects with respect to the three types of library releases. We can see that 9 (1.1%) projects share



The Average Usage Outdatedness (# Releases) of the Library Dependencies in a Project w.r.t. Major Versions



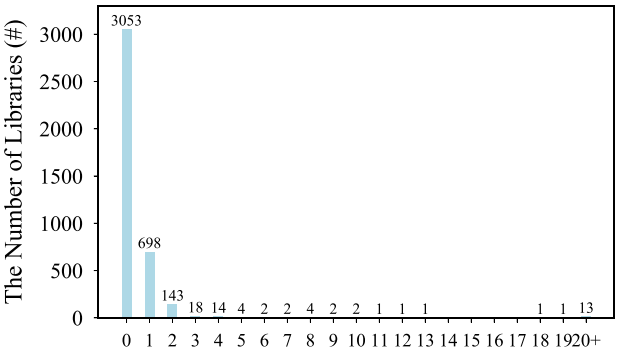
The Average Usage Outdatedness (# Releases) of the Library Dependencies in a Project w.r.t. Minor Versions



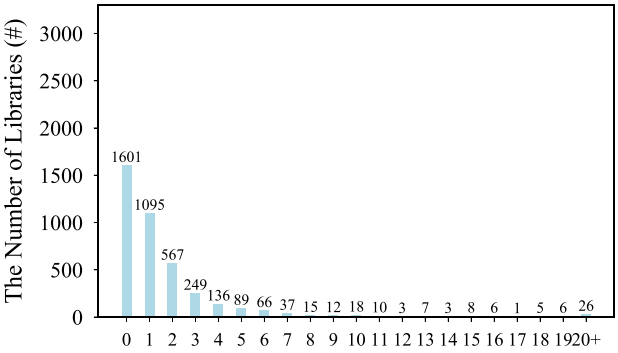
The Average Usage Outdatedness (# Releases) of the Library Dependencies in a Project w.r.t. Patch Versions

Fig. 13 Usage Outdatedness (# Releases) across Projects w.r.t. the Type of Library Releases

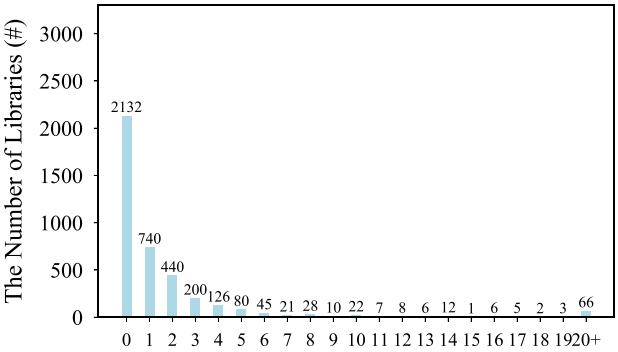
the same major version to the latest version, and 470 (58.3%) and 246 (30.5%) projects have a lag of more than 500 and 1000 days from the latest major version. For the libraries with the same major version, 9 (1.1%) projects share the same minor version to the lasted



The Average Usage Outdatedness (# Releases) of the Library Dependencies on a Library w.r.t. Major Versions



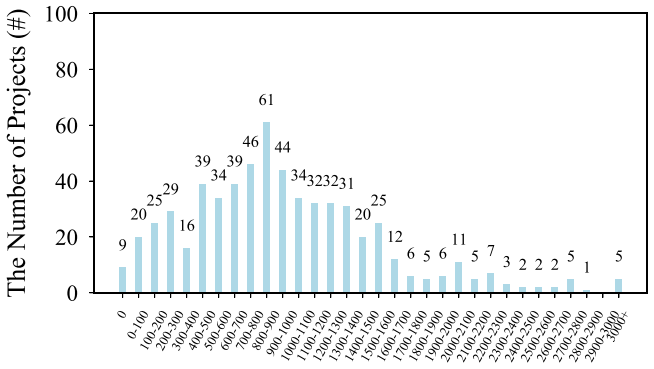
The Average Usage Outdatedness (# Releases) of the Library Dependencies on a Library w.r.t. Minor Versions



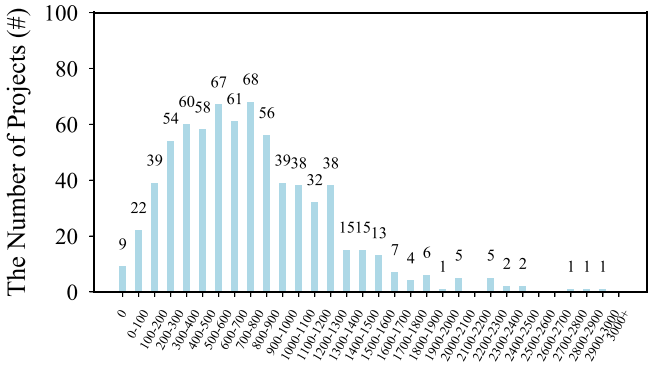
The Average Usage Outdatedness (# Releases) of the Library Dependencies on a Library w.r.t. Patch Versions

Fig. 14 Usage outdatedness (# releases) across libraries w.r.t. the type of library releases

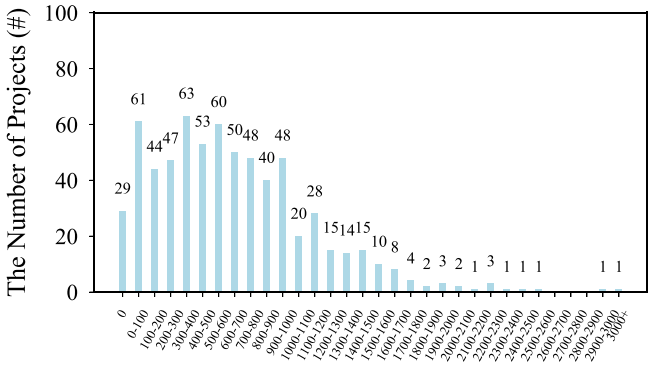
version, and 477 (59.2%) and 186 (23.1%) projects have a lag of more than 500 and 1000 days from the latest version that shares the same major version. Similarly, only 29 (3.1%) projects share the same patch version to the lasted version that has the same major and minor



The Average Usage Outdatedness (# Days) of the Library Dependencies in a Project w.r.t. Major Versions



The Average Usage Outdatedness (# Days) of the Library Dependencies in a Project w.r.t. Minor Versions



The Average Usage Outdatedness (# Days) of the Library Dependencies in a Project w.r.t. Patch Versions

Fig. 15 Usage outdatedness (# days) across projects w.r.t. the type of library releases

version, and 376 (46.7%) and 130 (16.1%) projects have a lag of more than 500 and 1000 days from the latest version that shares the same major and minor version. Notice that 28 projects are also not included in Fig. 15. On the other hand, Fig. 16 reports the distribution of usage outdatedness in terms of days across libraries with respect to the three types of library releases. We can see that in all the used libraries, 706 (9.8%) libraries have a lag of at least 500 days from the latest version. 1,081 (15.0%) and 438 (6.1%) libraries have a lag of at least 500 and 1000 days from the latest version that has the same major version. 743 (10.3%) and 289 (4.0%) libraries have a lag of at least 500 and 1000 days from the latest version that has the same major and minor version.

Summary. Around one-third of the projects have a lag of one major version from the latest library version, and around 80% and 70% of the projects have a lag of at least one minor and patch version although minor and patch versions are supposed to be compatible. More than half of the projects have a lag of 500 days from the latest version, and around 60% and 50% of the projects have a lag of more than 500 days from the latest version. Automated library version update techniques are needed for projects to keep libraries up-to-date, especially for major versions which may introduce incompatibilities.

5 Library Update Analysis

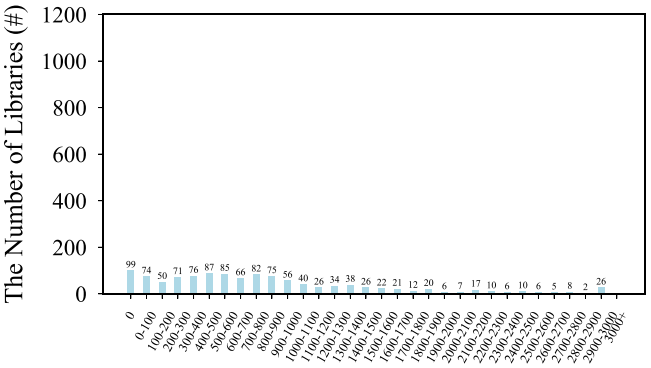
To study library updates, we develop *up-extractor* to extract library version updates from a project's commit history. It scans a project p's commits to identify each commit *com* that changes p's configuration files. It then uses *lib-extractor* (see Section 4) on *com* and *com*'s previous commit so as to respectively extract the library dependencies before and after *com*. Finally, using the two sets of library dependencies, it identifies library version updates by searching library dependencies whose version number is changed. Each library version update u is denoted as a 6-tuple $\langle p, f, com, l, ver_1, ver_2\rangle$, where p, f, *com* and l respectively denote the project, configuration file, commit and library where u occurs, and ver_1 and ver_2 respectively denote the version number before and after the update.

We used *up-extractor* to the commits of each project in $\mathcal{P}$, and extracted 5,217,348 library version updates, denoted as $\mathcal{U}$.

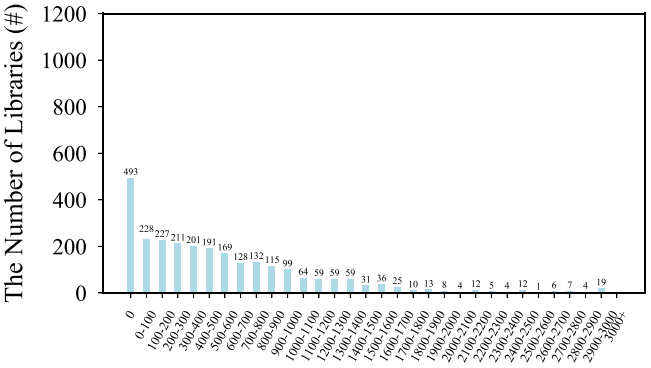
5.1 Update Intensity

Definition We define update intensity from the perspective of a project and library: upi_p, the percent of a project p's currently declared library dependencies whose version numbers were updated in p's commits, and upi_l, the percent of projects that currently contain a dependency on a library l and updated l's version number in commits. Using $\mathcal{P}$, $\mathcal{L}_{us}$, $\mathcal{D}_{us}$ and $\mathcal{U}$, we compute upi_p and upi_l by (7), where $\mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\}$ and $\mathcal{P}_l = \{d.p \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\}$.

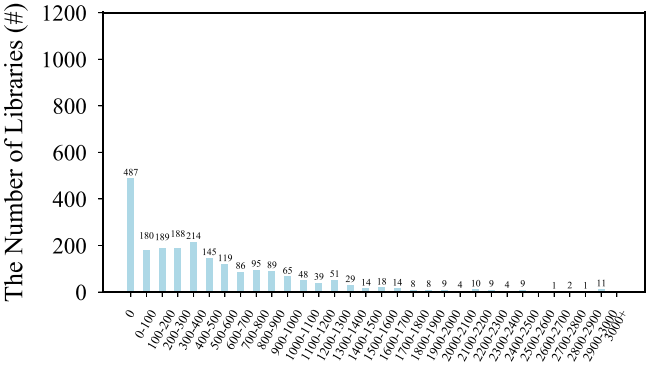
$$\begin{aligned} \forall p \in \mathcal{P}, upi_p &= |\{d \mid d \in \mathcal{D}_P \wedge (\exists u \in \mathcal{U}, u.p = d.p \wedge u.f = d.f \wedge \\ &\qquad u.l = d.v.l)\}|/|\mathcal{D}_p| \\ \forall l \in \mathcal{L}_{us}, upi_l &= |\{p \mid p \in \mathcal{P}_l \wedge (\exists u \in \mathcal{U}, u.p = p \wedge u.l = l)\}|/|\mathcal{P}_l| \end{aligned} \tag{7}$$



The Average Usage Outdatedness (# Days) of the Library Dependencies on a Library w.r.t. Major Versions



The Average Usage Outdatedness (# Days) of the Library Dependencies on a Library w.r.t. Minor Versions



The Average Usage Outdatedness (# Days) of the Library Dependencies on a Library w.r.t. Patch Versions

Fig. 16 Usage outdatedness (# days) across libraries w.r.t. the type of library releases

Findings Using upi_p and upi_l, we report distributions of update intensity across projects and libraries in Figs. 17 and 18. On the one hand, 114 (14.1%) projects did not update any currently-declared library dependency. Of them, 90 projects never updated any library dependency, and 24 projects updated library dependencies that were removed. 89 (11.0%) and 329 (40.8%) projects respectively updated at most 20% and 50% of their currently-declared library dependencies. 354 (43.9%) and 101 (12.5%) projects respectively updated more than 50% and 80% of their currently-declared library dependencies. Notice that 9 projects do not declare any library dependency and are not included in Fig. 17. On the other hand, 4,414 (32.5%) libraries were never updated in all the projects that depend on them. At the other extreme, 7,210 (53.2%) libraries were updated in more than 95% of the projects that use them. Such two extremes are mainly caused by the fact that 90.3% of these libraries are only used by one project. If excluding all 10,499 libraries only used by one project, we find that of the remaining 3,066 libraries, 2,004 (65.4%) libraries were not updated in more than half of the projects that adopt them.

Summary. Project developers do update libraries. Still, half of the projects leave more than half of the adopted libraries never updated; and one-third of the libraries are not updated in more than half of the projects that use them. Considering that the selected projects are active, the update practice is considerably poor. Awareness of the importance of updating libraries should be raised.

We further analyze the changes of version numbers in library version updates with respect to semantic versioning (Preston-Werner 2013). We identified 5,117,870 (98.1%) library version updates from $\mathcal{U}$ whose version numbers start with $X.Y$ or $X.Y.Z$, denoted as $\hat{\mathcal{U}}$. On one hand, we explore whether developers upgrade or downgrade a library version. As reported in the left chart in Fig. 19, most updates are upgrades; and a very small part (3.6%) of updates are downgrades due to incompatible APIs. 14.9% of them include diverse suffixes in version numbers, and hence are unknown due to incomparable version numbers. On the other hand, we study whether developers update major, minor, patch or snapshot versions, and give the results in the middle and left chart in Fig. 19. First, 14.5% of upgrades

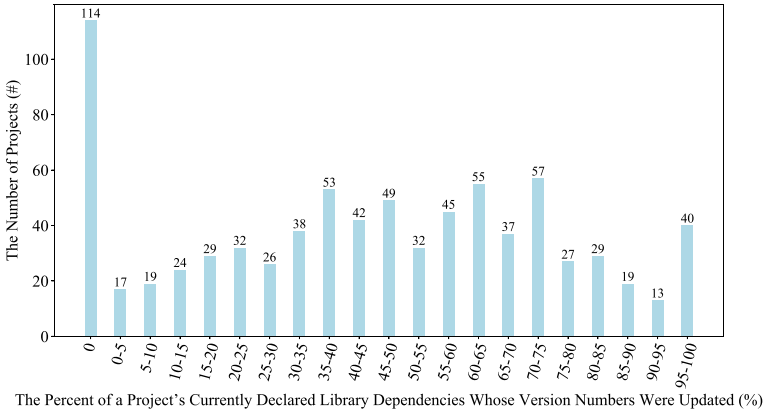


Fig. 17 Update intensity across projects

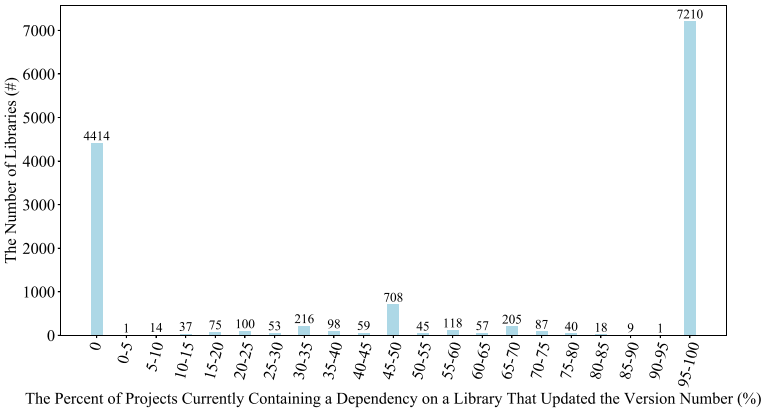


Fig. 18 Update intensity across libraries

replace snapshot versions with stable versions due to the unstable nature of snapshot versions; and 20.9% of downgrades switch back to snapshot versions due to heavy dependency on unstable APIs. Second, 79.8% upgrades are minor or patch as they are supposed to be API compatible; and 72.5% downgrades are minor or patch due to violations of semantic versioning. Third, major upgrades or downgrades are less common as incompatible APIs can be introduced in major versions.

Summary. Semantic versioning is not strictly followed. Tools are needed to analyze whether any incompatible change is introduced before a new version is released, and suggest the correct version number that follows semantic versioning. Besides, major versions deserve a mechanism to be kept updated.

5.2 Update Delay

Definition We define update delay of a library version update u, denoted as upd_u, as the delay between the commit date of u and the release date of the library version after u. For each library version update $u \in \mathcal{U}$, we crawled the release data of $\langle u.l, u.ver_2 \rangle$ from library repositories. We successfully crawled for 1,507,196 (28.9%) library version updates, denoted as $\tilde{\mathcal{U}}$, resulting in 155,969 library releases, denoted as $\mathcal{R}_{up}$. From $\mathcal{R}_{up}$, we had

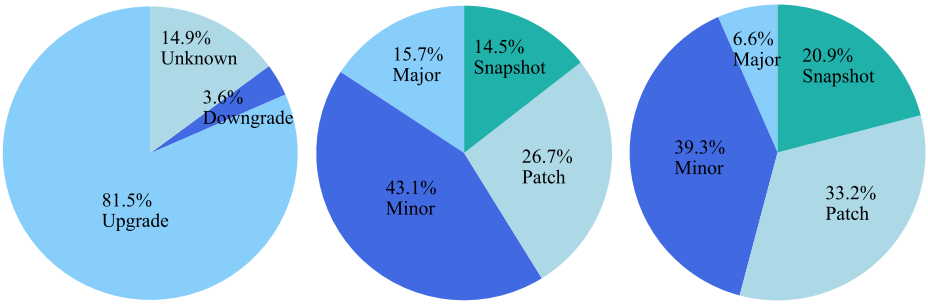


Fig. 19 Distribution of Updates (left), Upgrades (middle) and Downgrades (right)

9,438 libraries, denoted as $\mathcal{L}_{up}$ (i.e., $\mathcal{L}_{up} = \{r.v.l \mid r \in \mathcal{R}_{up}\}$). Of the library version updates we failed to crawl, 87.8% are caused by unavailable snapshot versions. Using $\tilde{\mathcal{U}}$ and $\mathcal{R}_{up}$, we compute upd_u by (8).

$$\begin{aligned} &\forall u \in \tilde{\mathcal{U}}, upd_u = u.com.date - r.date, \\ &r \in \mathcal{R}_{up} \wedge r.v.l = u.l \wedge r.v.ver = u.ver_2 \end{aligned} \tag{8}$$

Then, we define update delay from the perspective of a project and library: upd_p, the average update delay of the library version updates in a project p, and upd_l, the average update delay of the library version updates on a library l. Using $\mathcal{P}$, $\mathcal{L}_{up}$, $\tilde{\mathcal{U}}$ and upd_u, we compute upd_p and upd_l by (9).

$$\begin{aligned} \forall p \in \mathcal{P}_{up}, upd_p &= avg_{u \in \mathcal{U}_p} upd_u, \mathcal{U}_p = \{u \mid u \in \tilde{\mathcal{U}} \wedge u.p\} \\ \forall l \in \mathcal{L}_{up}, upd_l &= avg_{u \in \mathcal{U}_l} upd_u, \mathcal{U}_l = \{u \mid u \in \tilde{\mathcal{U}} \wedge u.l = l\} \end{aligned} \tag{9}$$

Findings Using upd_p and upd_l, we show distributions of update delay across projects and libraries in Figs. 20 and 21. On the one hand, 186 (23.1%) projects updated their library dependencies at a lag of at most 30 days. 407 (50.5%), 256 (31.8%) and 174 (21.6%) projects had an update delay of more than 60, 120 and 180 days, respectively. 107 (13.3%) projects are not included in Fig. 20 since we failed to compute the update delay (i.e., 90 projects never updated any library dependency; and 17 projects updated library dependencies but we failed to crawl the release date). On the other hand, 6,856 (72.6%) libraries were updated at a lag of at most 30 days. 1,951 (20.7%), 1,355 (14.4%) and 985 (10.4%) libraries had an update delay of over 60, 120 and 180 days.

Summary. Project developers have a slow reaction to new library releases. Such a wide time window could increase the risk (e.g., security bugs) of using outdated libraries, or even increase the difficulty of updating to new releases as more library APIs would be used during this time window.

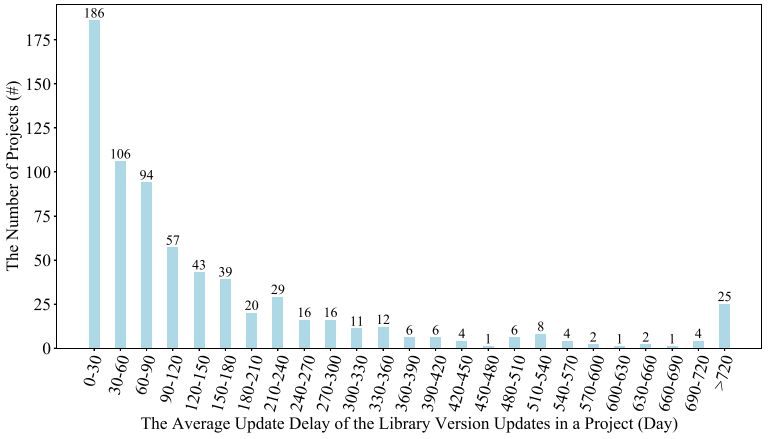


Fig. 20 Update delay across projects

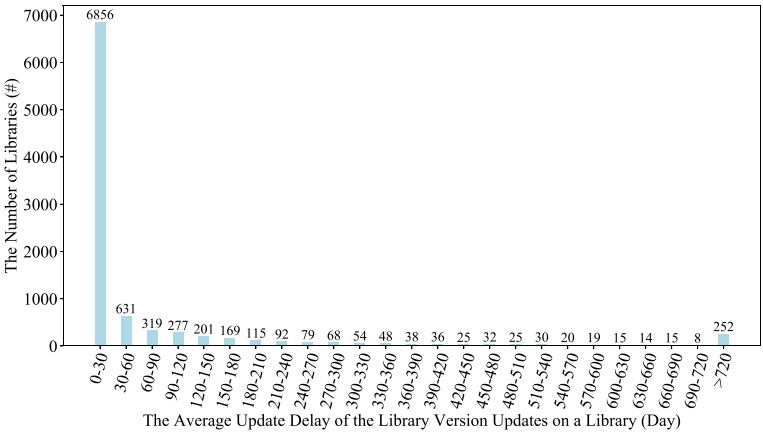


Fig. 21 Update delay across libraries

6 Library Risk Analysis

Several tools (e.g., Black Duck,[8] Veracode,[9] SAP (Ponta et al. 2018), OWASP, Snyk and Dependabot) have been recently proposed to notify developer about security bugs in used library versions. Besides, bug fixing is also recognized as the most common reason for updating libraries (Derr et al. 2017). Therefore, we study library risks with respect to security bugs. To this end, we develop *bug-crawler* to collect security bugs in a library by first searching from Veracode's vulnerability database and then crawling the metadata (e.g., affected library releases and security patch) of each security bug. Here we use Veracode's vulnerability database instead of crawling from libraries' bug trackers because i) Veracode's vulnerability database is one of the most famous open-source databases and ii) Veracode maintains the mapping between security bugs and the affected library versions, while different libraries may use different bug trackers which often do not distinguish between security bugs and non-security bugs. We represent a security bug in a library release as a bug-release pair g, denoted as a 2-tuple $\langle b, r \rangle$, where b denotes a security bug, and r denotes a library release that b affects (i.e., r is buggy).

We focused on the security bugs in the libraries (i.e., $\mathcal{L}_{us}$) used by the 806 projects. We applied *bug-crawler* to each library $l \in \mathcal{L}_{us}$, and collected 544 security bugs, affecting 252 libraries. We manually checked these 544 security bugs to ensure the correct mapping between security bugs and their affected library versions. These 252 libraries have 17,421 library releases (computed from $\mathcal{R}_{us}$), denoted as $\mathcal{R}_{ri}$. By parsing affected library releases of the 544 security bugs, we had 35,196 bug-release pairs, denoted as $\mathcal{G}$. From $\mathcal{G}$, we had 11,989 buggy library releases, denoted as $\tilde{\mathcal{R}}_{ri}$ (i.e., $\tilde{\mathcal{R}}_{ri} = \{g.r \mid g \in \mathcal{G}\}$).

6.1 Usage Risk

Definition We define usage risk of a project as two indicators: usr_p^1, the number of buggy library versions a project uses, and usr_p^2, the number of security bugs in the buggy library

[8] https://www.blackducksoftware.com
[9] https://www.veracode.com

versions a project uses. We define usage risk of a library release as usr_r, the number of security bugs in a library release. These indicators report the *upper bound* of usage risk because not all security bugs will actually affect a project. Using $\mathcal{P}$, $\mathcal{D}_{us}$, $\mathcal{G}$ and $\mathcal{R}_{ri}$, we compute usr_p^1, usr_p^2 and usr_r by (10), where $\mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\}$.

$$\begin{aligned} \forall p \in \mathcal{P}, usr_p^1 &= |\{d.v \mid d \in \mathcal{D}_p \wedge (\exists g \in \mathcal{G}, g.r.v = d.v)\}| \\ \forall p \in \mathcal{P}, usr_p^2 &= \sum_{d \in \mathcal{D}_p} |\{g.b \mid g \in \mathcal{G} \wedge g.r.v = d.v\}| \\ \forall r \in \mathcal{R}_{ri}, usr_r &= |\{g.b | g \in \mathcal{G} \wedge g.r = r\}| \end{aligned} \tag{10}$$

Findings Using usr_p^1, usr_p^2 and usr_r, we present distributions of usage risk across projects and library releases in Figs. 22 and 23. On one hand, 451 (56.0%) projects adopt buggy library versions. 328 (40.7%) projects adopt 1 to 5 buggy library versions, and 123 (15.3%) projects even use more than 5 buggy library versions. In 207 (25.7%) projects, their used buggy library versions have 1 to 5 security bugs. In 188 (23.3%) projects, their used buggy library versions even have more than 10 security bugs. On the other hand, of the 17,421 library releases, only 5,432 (31.2%) library releases do not have security bug. 5,953 (34.2%) library releases have 1 security bug, while 3,911 (22.4%) and 1,149 (6.6%) respectively have more than 2 and 5 security bugs.

Summary. More than half of the projects use library versions that contain security bugs. Two-third of the library releases contain security bugs. The relatively common existence of security bugs indicates the potential risk faced by projects if project developers are unaware of the security bugs in used libraries or delay library updates.

6.2 Update Risk

Definition We define update risk of a project as two indicators: upr_p^1, the number of called APIs in buggy library versions a project uses, and upr_p^2, the number of API calls to buggy

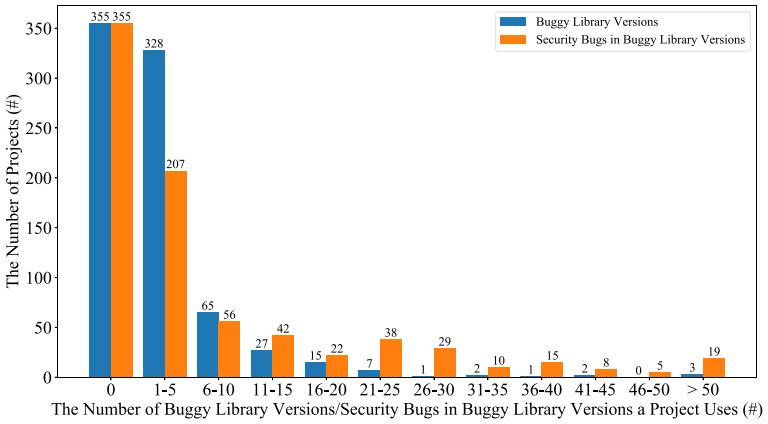


Fig. 22 Usage risk across projects

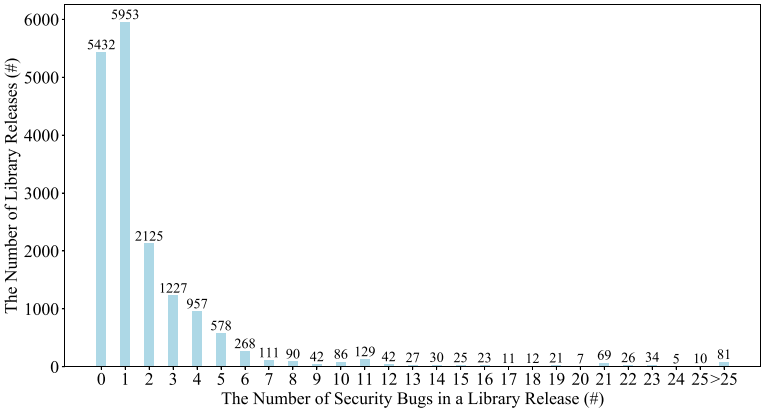


Fig. 23 Usage risk across library releases

library versions a project uses. These two indicators report the *upper bound* of update risk as not all called APIs in buggy library versions will be *deleted* (i.e., the API signature does not exist, which will fail the compilation) or *modified* (i.e., the API signature is not changed but its behavior is changed, which will pass the compilation) in safe library versions. Using $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{G}$, we compute upr_p^1 and upr_p^2 by (11).

$$\begin{aligned} \forall p \in \mathcal{P}, upr_p^1 &= |\{c.a | c \in \mathcal{C} \wedge c.m.p = p \wedge (\exists g \in \mathcal{G}, g.r.v = c.a.v)\}| \\ \forall p \in \mathcal{P}, upr_p^2 &= |\{c | c \in \mathcal{C} \wedge c.m.p = p \wedge (\exists g \in \mathcal{G}, g.r.v = c.a.v)\}| \end{aligned} \tag{11}$$

We define update risk of a buggy library release as two indicators: upr_r^1, the number of APIs in a buggy library release that are deleted in the safe library release, and upr_r^2, the number of APIs in a buggy library release that are modified in the safe library release. There can be multiple safe library releases that fix the security bugs in the buggy library release. Here, we choose the safe library release with the smallest version number. These two indicators also report the *upper bound* of update risk as not all deleted/modified APIs are used by a project. Using $\tilde{\mathcal{R}}_{ri}$ and $\mathcal{A}$, we can compute upr_r^1 and upr_r^2 by (12), where $\mathcal{A}_x = \{a \mid a \in \mathcal{A} \wedge a.v = x.v\}$ (i.e., the APIs in a library release x), r_{safe} denotes the safe library release, and $\sqcap$ computes the APIs whose signature is not changed but its code or the code of its transitively called methods is changed (i.e., potentially changing its behavior).

$$\begin{aligned} \forall r \in \tilde{\mathcal{R}}_{ri}, upr_r^1 &= |\{a.api \mid a \in \mathcal{A}_r\} - \{a.api \mid a \in \mathcal{A}_{r_{safe}}\}| \\ \forall r \in \tilde{\mathcal{R}}_{ri}, upr_r^2 &= |\{a.api \mid a \in \mathcal{A}_r\} \sqcap \{a.api \mid a \in \mathcal{A}_{r_{safe}}\}| \end{aligned} \tag{12}$$

Findings Using upr_p^1, upr_p^2, upr_r^1 and upr_r^2, we give distributions of update risk across projects and buggy library releases in Figs. 24 and 25. On one hand, of the 451 projects that adopt buggy library versions, 151 (33.5%) projects do not call APIs in used buggy library versions, 181 (40.1%) projects call at most 20 APIs, and 82 (18.2%) projects call more than 40 APIs. In addition, 133 (29.5%) projects have at most 20 API calls to used buggy library versions, and 122 (26.5%) projects have more than 40 API calls. On the other hand, of the 11,989 buggy library releases, 2,664 (22.2%) do not have safe library releases (i.e., the security bug affects all current library releases), and thus are not included in Fig. 25. 4,586 (38.3%) buggy library releases have less than 40 APIs modified in the safe library

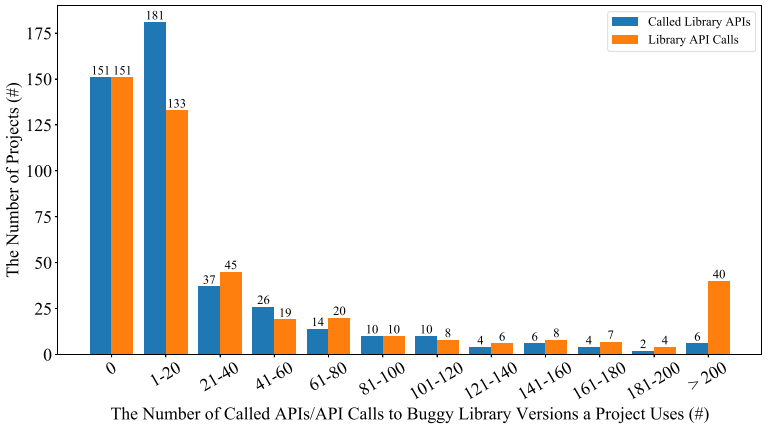


Fig. 24 Update risk across projects

release, while 2,921 (24.4%) buggy library releases have more than 100 APIs modified in the safe one. 2,065 (17.2%) buggy library releases have less than 40 APIs deleted in the safe library release. Surprisingly, 4,236 (35.3%) buggy library releases have more than 300 APIs deleted in the safe one.

Summary. These API-level upper bounds of usage and update risk present moderate risk in using and updating buggy library versions. Tools are needed to provide a tight estimation of risk via combining the library APIs used in a project and the library APIs changed in safe versions such that developers can confidently update buggy libraries.

7 Discussion

In this section, we elaborate the practical implications, demonstrate the usefulness of our findings, and discuss the threats to our study.

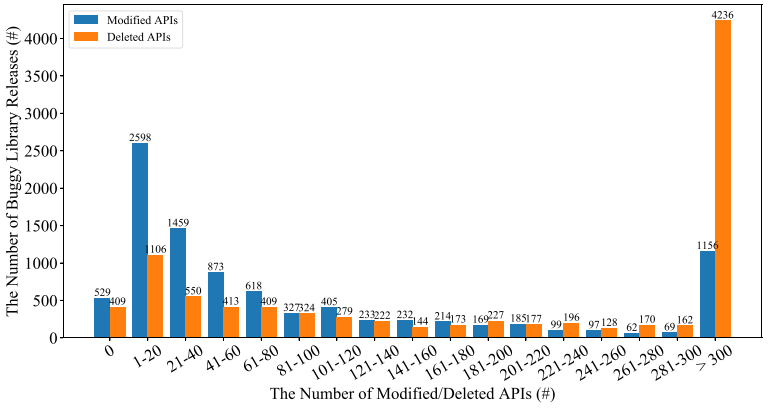


Fig. 25 Update risk across library releases

7.1 Implications to Researchers and Developers

Ecosystem-Level Knowledge Graph Our study actually attempts to put projects, libraries and security bugs in one picture and draw connections among them. Ideally, every parties in the ecosystem should be involved to establish connections and produce intelligence to foster the ecosystem. For example, if the knowledge about *how the APIs of a library are used by all the projects that use the library* is available, library developers can get instant reminders when they evolve APIs that are widely used and have high change impact. If the knowledge about *security bugs in a library and all the projects that use the library* is available, project developers can have instant alerts when new security bugs are disclosed. Towards this paradigm, knowledge graph seems to be a feasible solution. In particular, all open-source projects and their called library APIs, library releases and their APIs and call graphs, bugs (e.g., performance and security) and their affect library APIs, licenses, and developers should be connected into one whole graph. The main challenges are: establish the infrastructure to support the huge graph, collect these various kinds of data in a (nearly) real-time way, and develop various graph analysis techniques to provide online services for developers (e.g., alerting buggy library versions and analyzing ecosystem-level change impact within IDEs). This study provides a starting point to achieve such a paradigm.

Library Debloat As a very small part of library APIs are widely called across projects, many unused library features are still kept in software systems, which can cause software bloat, especially for embedded systems. Software bloat can hurt performance (Xu et al. 2010) or broaden attack surface (e.g., code reuse attack) (Wang et al. 2019a). Therefore, the low usage intensity of library APIs opens an opportunity to tailor unused features in libraries (i.e., library debloat) in a specific usage context to avoid software bloat. Some debloat techniques (Quach et al. 2018; Sharif et al. 2018) have been designed for C/C++; Soto-Valero et al. (2020) removed unused transitive libraries as a whole but did not target partially-used libraries; and Matos et al. (2019) studied the possibility of splitting libraries into bundles based on the API usages of a set of client projects.

Just-In-Time Library Recommendation A very small part of libraries are widely used across projects (Section 4.1). One potential reason is that project developers are actually unaware of the libraries that can provide the functions they need and thus miss reuse opportunities. It may also hinder the growth of ecosystem; e.g., library developers may stop maintaining libraries. Therefore, library recommendation is needed to help project developers catch reuse opportunities. However, existing recommendation techniques (e.g., Thung et al. 2013a, b; Ouni et al. 2017; Chan et al. 2012) are not directly linked to developers' programming activities, and thus lack wide adoption. One potential remedy is to develop just-in-time library recommendation, which can instantly suggest libraries and library APIs based on the code context when developers are programming. Its difficulty is to summarize descriptions or even code samples to make developers take the suggestions confidently.

Library Selection While different libraries might provide similar functionalities, it is critical for developers to select the suitable library at the first place. Otherwise, it will become a tedious task to migrate to another library later. A potential way is to select suitable libraries based on information about multiple dimensions, e.g., library risk (e.g., security bugs and license conflicts), library maintenance (e.g., commit frequency and issue fixing rate), library maturity (e.g., number of developers and community usages). Therefore, a model needs to be established to capture various facets about libraries and facilitate library selection.

Multiple Version Harmonization Multiple versions of the same library are commonly used in multi-module project. It may increase the burden of project developers as it may cause inconsistent library API behaviors across different modules, or even lead to dependency conflicts when modules are inter-dependent (Wang et al. 2018; Patra et al. 2018). Therefore, techniques are needed to automatically detect multiple versions, analyze their differences in client usage context, and refactor client code and configuration files to harmonize into a single version. While it is a tedious task to unify multiple versions, we believe it brings sustainable benefits in library maintenance in the long run. Inspired by our work, Huang et al. (2020) have proposed an interactive, effort-aware library version harmonization approach.

Snapshot Version Management Snapshot library versions are commonly used. They are in fact unfinished versions that are still under heavy development, which increases the maintenance cost due to incompatible APIs. In that sense, it is a double-edged sword; i.e., it can keep projects using the latest library versions, but may cause projects depend on incompatible APIs. Therefore, it is worthwhile for researchers deeply investigating the benefits and costs of snapshot versions to shed light on how to better manage snapshot versions.

Smart Alerting and Automated Updating Given the wide existence of buggy, outdated libraries in projects, it is urgent to propose techniques to alert and update buggy, outdated libraries. On one hand, alerts should be raised only when security bugs in library versions are in execution paths of projects. Otherwise, buggy library versions are safe and would cause false positives if alerted, and developers would become overwhelmed by the increasing number of alerts they receive. Moreover, multiple fine-grained information should be provided to assist developers to make confident decisions in updating buggy library versions. Specifically, alerts should indicate the statistics about the library API calls affected by security bugs, such that developers can assess the risk of using buggy library versions. Alerts should also report the statistics about the calls to library APIs that are deleted or modified in the new library version, such that developers can assess the effort to complete the update. On the other hand, automated library updating techniques are needed to analyze behavior changes of modified library APIs and locate replacements of deleted library APIs. Another potential solution to automated updating is to learn updating patterns from projects that finish the update. However, it may be limited to the size of such update data.

Usage-Driven Library Evolution Our library usage analysis presents an opportunity for library developers to conduct usage-driven library evolution, e.g., giving high fix priority to bugs in widely-used library APIs, carefully evolving widely-used library APIs based on change impacts on client projects, redesigning or optimizing library APIs based on their usage statistics, and assessing whether new library APIs are adopted.

Dataset and Visualization Our study produces a lot of data: declared library dependencies, library APIs, library API calls, library releases, library version updates, and security bugs in library versions. We released them at https://3rdpartylibs.github.io together with our analysis tools to ease the reproduction and foster valuable applications. One potential application is to provide an online service to keep usage, update and risk results up-to-date, showing real-time statistics of these projects, libraries and security bugs, and develop visualization techniques (Kula et al. 2014, 2018a) to visualize our data to assist project and library developers to make decisions in evolving, adopting and updating libraries.

7.2 Application for Usefulness Demonstration

Based on the implication on smart alerting, we propose a security bug-driven alerting system for buggy libraries, named LIBSECURIFY. The goal of LIBSECURIFY is to quantify the risk with respect to the security bugs in used library versions for a project, and to recommend safe library versions with quantified updating effort.

7.2.1 Approach Overview

Figure 26 presents the approach overview of LIBSECURIFY, which consists of four steps, i.e., *dependency analysis*, *library API analysis*, *risk analysis* and *effort analysis*. It relies on two databases, i.e., security bug database and library database. The security bug database currently has 544 security bugs collected in our risk analysis (see Section 6) and the corresponding buggy library methods (i.e., the methods that are changed in the security patch to fix a security bug) in affected library versions. The library database currently has all the released versions of the 252 libraries affected by 544 security bugs. To keep the two databases up-to-date, we implement a pipeline to regularly crawl security bugs, affected library versions, and their patches from Veracode and all the released versions of libraries that are affected by security bugs from Maven.

7.2.2 Approach Details

Our dependency analysis first uses *lib-extractor* (see Section 4) to extract library dependencies for a target project, and then determines the set of extracted library dependencies, D_b, that contain security bugs by querying the library version from the security bug database. If D_b is empty, LIBSECURIFY returns and reports that the target project is safe; otherwise, it means that the security bugs in D_b may potentially affect the target project, and thus LIBSECURIFY continues to the next library API analysis step.

Our library API analysis is to determine the library API calls and the library APIs called in the target projects. To this end, for each library dependency in D_b, it checks whether its library APIs are already in the cache (i.e., its library APIs have been extracted during the analysis for other projects). The representation for library APIs' cache is the same to Section 5, which is denoted as $\mathcal{A}$. If not, it uses Soot (Vallée-Rai et al. 1999) on the jar file (obtained by querying the library database) of the library dependency to extract its library APIs and store them to the cache. In this way, the library API extraction for each library version is conducted once and reused across projects. Then, it uses JavaParser (Smith et al.

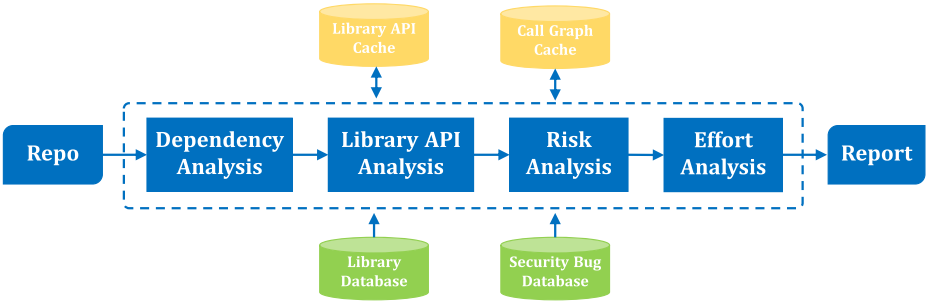


Fig. 26 Approach overview of LIBSECURIFY

2017) on the repository of the target project with type binding to extract library API calls in the target project, and summaries the library APIs called in the target project.

Our risk analysis is to determine whether the target project directly or indirectly calls buggy library methods. For each of the called library APIs in $\mathcal{A}$, if its call graph is not in the cache, our risk analysis first uses Soot (Vallée-Rai et al. 1999) to construct the call graph, and stores the call graph to the cache for reuse across projects. The call graph $\mathcal{CG}$ is denoted as a set of edges and nodes $\langle \mathcal{N}, \mathcal{E} \rangle$, where for each $n \in \mathcal{N}$, n denotes the library method in the call graph, and for each $e \in \mathcal{E}$, e denotes the caller method n_{caller} and the callee method n_{callee} from $\mathcal{N}$. Then, it checks whether the library methods $\mathcal{N}$ in each call graph contain buggy library methods by querying our security bug database. If yes, we consider the called library API as *risky* and affecting the target project (i.e., the corresponding security bug could in the execution path of the target project). After this analysis, we can report the number of security bugs that affect the target project in each buggy library version (i.e., NB), the number of risky library APIs called in the target project (i.e., NA), and the number of calls to risky library APIs in the target project (i.e., NC). These three metrics provide developers with the risk and impact of adopted buggy library versions.

Our effort analysis is to recommend the safe library versions and the updating effort. For a risky library and its called APIs, it queries the library database for safe and higher library versions as candidates for recommendation. For each higher library version than the buggy library version, it first determines whether the called library APIs are deleted or modified in the higher library version. Here, an API is modified if the body code of the API or the code of the library methods in its call graph is modified. It uses Soot (Vallée-Rai et al. 1999) to get the method body and compare with the higher library version via computing their hash. The call graph construction and caching for higher library versions are conducted in the same way to risk analysis. The process of comparing call graphs follows a breath-first search where methods are compared layer by layer and are ranked by their signatures in each layer. An API is considered as deleted if the we cannot find the declaration of that API via its original signature in a higher version. Then, it checks whether the called library APIs (that are not deleted) directly or indirectly call buggy library methods in the higher library version. If yes, we skip this higher library version because it still contains security bugs affecting the target project. If no, we can report the number of called library APIs that are deleted (i.e., NAD), the number of called library APIs that are modified (i.e., NAM), the number of calls to the deleted library APIs (i.e., NCD), and the number of calls to the modified library APIs (i.e., NCM). By extracting the location and counting the occurence of library APIs in the target project, we have NCD and NCM. These metrics measure the updating effort on the recommended library version. Thus, with the summary and detailed location of these APIs in the target project provided by LibSecurify, developers can hence interact with LibSecurify to make easier decision on which library version to update, and inspect code which may be affected by version update. It is worth mentioning that these metrics capture partial updating effort because effort like figuring out how to adapt the source code based on the API changes and running test cases are hard to measure and are not included here.

7.2.3 Approach Evaluation

We have run our alerting system against the 451 projects that use buggy libraries (see Section 6) to determine whether the buggy libraries affect the projects. We find that 413 projects are not affected by the buggy libraries and can be safe. For the 38 unsafe projects, we list the detailed results in Table 1, where column *P* lists the projects, *BL* reports the

Table 1 Results of applying our alerting system

P	BL	NB	NA	NC	SL	NAD	NAM	NCD	NCM
1	1	1(5)	3(19)	4(36)	5	0	6	0	10
2	1	1(7)	2(6)	7(18)	5	0	5	0	11
3	1	1(2)	1(3)	1(3)	23	0	2	0	2
4	1	1(7)	7(40)	24(119)	5	0	20	0	81
5	1	1(2)	3(40)	4(68)	6	27	6	41	10
6	1	1(2)	2(26)	21(190)	15	0	6	0	49
7	1	1(4)	1(1)	3(3)	4	1	0	3	0
8	1	1(4)	3(16)	24(51)	5	1	7	1	36
9	1	1(3)	1(7)	1(7)	16	0	2	0	2
10	1	2(5)	1(4)	1(4)	4	4	0	4	0
11	1	1(2)	3(18)	6(44)	2	0	11	0	23
12	1	1(4)	1(6)	2(10)	4	6	0	10	0
13	1	1(1)	9(19)	11(23)	9	0	0	0	0
14	1	1(4)	1(61)	1(136)	4	0	11	0	30
15	1	1(2)	2(61)	2(142)	5	4	14	5	44
16	1	1(3)	1(32)	1(35)	5	2	3	2	3
17	1	1(7)	6(56)	39(104)	15	0	28	0	70
18	1	1(3)	1(5)	1(7)	16	0	2	0	3
19	1	1(1)	1(68)	1(221)	9	0	31	0	115
20	1	1(5)	1(5)	1(11)	5	0	3	0	9
21	1	1(2)	3(33)	5(76)	6	9	18	20	46
22	1	1(3)	1(4)	1(5)	5	0	2	0	3
23	1	1(2)	2(13)	2(16)	6	0	6	0	7
24	1	1(7)	3(22)	5(66)	5	0	12	0	38
25	1	1(4)	4(28)	10(50)	5	0	11	0	30
26	1	1(5)	19(107)	475(1,918)	5	1	53	1	1,653
27	1	1(8)	2(11)	6(19)	10	0	4	0	8
28	1	1(7)	6(51)	45(303)	5	0	17	0	84
29	1	1(3)	3(5)	3(5)	5	0	3	0	3
30	1	1(7)	2(5)	3(6)	5	0	5	0	6
31	1	2(5)	1(5)	1(6)	4	5	0	6	0
32	1	1(5)	5(11)	6(20)	5	0	8	0	11
33	2	2(8)	2(34)	2(98)	9	0	14	0	35
34	2	3(3)	9(118)	28(296)	200	9	16	20	47
35	1	1(4)	2(16)	5(36)	2	0	3	0	6
36	1	1(2)	1(14)	3(43)	15	0	1	0	3
37	1	1(7)	11(23)	16(37)	10	0	16	0	24
38	1	1(7)	12(81)	47(337)	5	1	37	31	162

number of buggy libraries affecting the project, *NB*, *NA* and *NC* are the reported metrics in risk analysis (where the total number of security bugs, called library APIs, and calls to the library APIs are listed in parentheses), *SL* reports the number of suggested library versions,

and the other columns list the metrics in effort analysis for one suggested library version. Results for other suggested library versions are at https://3rdpartylibs.github.io.

Our evaluation indicates that all 38 projects are mostly affected by one buggy library version. While many security bugs exist in the buggy library version, mostly only one security bug affects a median of 2 library APIs which are called by a median of 4 times. For example, in the fourth project, 7 of the 40 called library APIs are affected by 1 security bug, and are called by 24 times. Multiple safe library versions are suggested for developers to choose according to updating effort. For example, 5 versions are suggested for the fourth project. In one of them, 20 of the 40 called library APIs are modified, affecting 81 library API calls; and no called library API is deleted.

We submitted issues to 274 of the 413 safe projects by reporting the used buggy library versions, the security bugs in them and safe versions. We did not submit for the remaining 139 projects as they already updated the buggy library versions at the time of our issue reporting, disabled issues and also did not use other issue trackers, or were read-only projects. Finally, we got replies from 78 projects in a week. Specifically, 29 issues have been confirmed and 23 of them have been fixed. For example, in *Alluxio/alluxio*, the developer responded that "thank you for this comprehensive list of CVEs! We recently merged a PR that updated a few dependencies. I think now would be a good time to address these as well. We will look into it shortly". 28 issues do not affect the projects as they affect test code only or are not in the execution path, while developers will still fix it in 4 issues. For example, in *eclipse/jetty.project*, the developer replied that *"Good catch regarding Hazelcast but except Hazelcast everything else is used for test. Maybe you should probably update your robot to not report error regarding dependencies with scope test?"*. 14 issues are still under analysis. Developers ask for pull request or thank for our issue without any further action in 7 issues. For example, in *apache/rocketmq*, the developer replied that *"we are grateful to you for pointing out these security holes, and we will check carefully and eliminate the hidden dangers. and it would be nice if you can submit a PR to make improvements together*. These results indicate that developers tend to update buggy libraries as long as they contain security bugs.

Then, we submitted issues to 31 of the 38 unsafe projects by reporting the execution path to the security bug and our fine-grained information in Table 1. We did not submit for the remaining 7 projects as they already updated the buggy library versions at the time of our issue reporting or were read-only projects. Finally, we received replies from 10 projects. 4 issues have been confirmed and 1 of them has been fixed. 4 issues are still under analysis. Developers decide to not fix the issue for backward compatibility in 1 issue, and ask for pull request in 1 issue. We are enlarging our security bug and library database, and collaborating with our two interested industrial partners to deploy our tool into continuous integration.

Moreover, we measured the time overhead of each step in LibSecurify. On average, LibSecurify took 39.24 min for a project, where dependency analysis took 0.23 min, library API analysis took 7.29 min, risk analysis took 15.79 min, and effort analysis took 15.94 min. The most time-consuming step in library API analysis is to extract library APIs, and the most time-consuming step in risk analysis and effort analysis is to construct call graphs. It is worth mentioning that when we re-ran LibSecurify against these projects (e.g., in the scenario that we need to recheck these projects after some code changes), the time overhead of LibSecurify was reduced to 15.24 min, where the time overhead of library API analysis, risk analysis and effort analysis was respectively reduced to 2.92, 5.05 and 7.04 min because some library APIs and call graphs are cached and can be reused.

7.3 Threats to Validity

The validity of this study may subject to some threats.

Indirect Library Dependencies Our study is focused on direct library dependencies, i.e., libraries that are directly declared in projects' configuration files. Libraries can depend on other libraries, i.e., indirect library dependencies. It can be expected that, if indirect library dependencies are considered, the dependency on libraries can be heavier, the potential risk in terms of security bugs can be higher, and the problem of using multiple versions of the same library can be more severe. We believe our findings are still meaningful if only considering direct library dependencies.

Subject Representativity We use active projects because library usage and update are software maintenance activities and inactive projects might contain less representative maintenance data and bias our findings. An independent study on those not active projects is needed to compare the findings. Besides, we focus on open-source projects in this study, and it is interesting to investigate whether industrial projects use libraries in the same way as in open-source projects. Moreover, we fail to crawl jar files and release dates of some libraries, and hence they are excluded from some of our analyses, but we have tried our best and clarified the data for each analysis. We believe our data is still representative and meaningful due to the large size.

Library APIs We conservatively regard public methods and fields in public classes as library APIs. Thus, some public methods and fields that are not meant to be used by client projects are also regarded as library APIs. Therefore, the real usage intensity can be higher than reported. However, the only ground truth is in the documentations for library releases, and is not always available.

Soundness of Program Analysis We use JavaParser (Smith et al. 2017) to extract library API calls and Soot (Vallée-Rai et al. 1999) to extract library APIs and construct call graphs. It is well known that such program analysis tools can inevitably produce unsound results. As both JavaParser and Soot are the state-of-the-art, we believe our results and findings can have a high confidence level. We are investigating the combination of static and dynamic analysis to improve the precision.

8 Conclusions

We conducted a quantitative and holistic study to characterize usages, updates and risks of third-party libraries in Java open-source projects. Specifically, we quantified the usage and update practices from the perspective of open-source projects and third-party libraries, and analyzed risks in terms of security bugs in third-party libraries. We provided implications to developers and researchers on problems and remedies in maintaining third-party libraries. We designed a security bug-driven alerting system prototype for buggy libraries. We also released our dataset and source code of our tools at https://3rdpartylibs.github.io to foster valuable applications and improve the ecosystem of third-party libraries more sustainably. In future, we plan to developer various techniques to achieve ecosystem-level knowledge graph, library debloat, library recommendation, library selection, automated library update, and visualization.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 61802067). Bihuan Chen is the corresponding author of this paper.

Declarations

Conflict of Interest The authors have no competing interests to declare that are relevant to the content of this article.

References

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? An empirical case study on npm. In: FSE, pp 385–395

Backes M, Bugiel S, Derr E (2016) Reliable third-party library detection in android and its security applications. In: CCS, pp 356–367

Balaban I, Tip F, Fuhrer R (2005) Refactoring support for class library migration. In: OOPSLA, pp 265–279

Bauer V, Heinemann L (2012) Understanding api usage to support informed decision making in software maintenance. In: CSMR, pp 435–440

Bauer V, Heinemann L, Deissenboeck F (2012) A structured approach to assess third-party library usage. In: ICSM, pp 483–492

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2013) The evolution of project inter-dependencies in a software ecosystem: the case of apache. In: ICSM, pp 280–289

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the apache community upgrades dependencies: an evolutionary study. Empir Softw Eng 20(5):1275–1317

Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to break an api: cost negotiation and community values in three software ecosystems. In: FSE, pp 109–120

Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: SANER, pp 516–519

Chan WK, Cheng H, Lo D (2012) Searching connected api subgraph via text phrases. In: FSE, pp 10:1–10:11

Chen C, Xing Z (2016) Similartech: automatically recommend analogical libraries across different programming languages. In: ASE, pp 834–839

Chow K, Notkin D (1996) Semi-automatic update of applications in response to library changes. In: ICSM, pp 359–368

Cossette BE, Walker RJ (2012) Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: FSE, p 55

Cox J, Bouwers E, van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: ICSE, vol 2, pp 109–118

Dagenais B, Robillard MP (2009) Semdiff: Analysis and recommendation support for api evolution. In: ICSE, pp 599–602

Dagenais B, Robillard MP (2011) Recommending adaptive changes for framework evolution. ACM Trans Softw Eng Methodol 20(4):19

De Roover C, Lammel R, Pek E (2013) Multi-dimensional exploration of api usage. In: ICPC, pp 152–161

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in oss packaging ecosystems. In: SANER, pp 2–12

Decan A, Mens T, Constantinou E (2018a) On the evolution of technical lag in the npm package dependency network. In: ICSME, pp 404–414

Decan A, Mens T, Constantinou E (2018b) On the impact of security vulnerabilities in the npm package dependency network. In: MSR, pp 181–191

Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep me updated: an empirical study of third-party library updatability on android. In: CCS, pp 2187–2200

Dietrich J, Jezek K, Brada P (2014) Broken promises: an empirical study into evolution problems in java programs caused by library upgrades. In: CSMR-WCRE, pp 64–73

Dig D, Johnson R (2006) How do apis evolve? A story of refactoring: research articles. J Softw Maint Evol 18(2):83–107

Fujibayashi D, Ihara A, Suwa H, Kula RG, Matsumoto K (2017) Does the release cycle of a library project influence when it is adopted by a client project? In: SANER, pp 569–570

Hejderup J, van Deursen A, Gousios G (2018) Software ecosystem call graph for dependency management. In: ICSE-NIER, pp 101–104

Henkel J, Diwan A (2005) Catchup! Capturing and replaying refactorings to support api evolution. In: ICSE, pp 274–283
Hora A, Valente MT (2015) Apiwave: keeping track of api popularity and migration. In: ICSME, pp 321–323
Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How do developers react to api evolution? The pharo ecosystem case. In: ICSME, pp 251–260
Howell DC (2012) Statistical methods for psychology, 8th edn. Cengage Learning
Huang K, Chen B, Shi B, Wang Y, Xu C, Peng X (2020) Interactive, effort-aware library version harmonization. In: ESEC/FSE, pp 518–529
Huang K, Chen B, Pan L, Wu S, Peng X (2021) Repfinder: finding replacements for missing apis in library update. In: ASE
Kabinna S, Bezemer CP, Shang W, Hassan AE (2016) Logging library migrations: a case study for the apache software foundation projects. In: MSR, pp 154–164
Khandkar SH (2009) Open coding. Tech. rep. University of Calgary
Kim M, Cai D, Kim S (2011) An empirical investigation into the role of api-level refactorings during software evolution. In: ICSE, pp 151–160
Kula RG, Roover CD, German D, Ishio T, Inoue K (2014) Visualizing the evolution of systems and their library dependencies. In: VISSOFT, pp 127–136
Kula RG, German DM, Ishio T, Inoue K (2015) Trusting a library: a study of the latency to adopt the latest maven release. In: SANER, pp 520–524
Kula RG, German DM, Ishio T, Ouni A, Inoue K (2017) An exploratory study on library aging by monitoring client usage in a software ecosystem. In: SANER, pp 407–411
Kula RG, De Roover C, German DM, Ishio T, Inoue K (2018a) A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In: SANER, pp 288–299
Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018b) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417
Kula RG, Ouni A, German DM, Inoue K (2018c) An empirical study on the impact of refactoring activities on evolving client-used apis. Inf Softw Technol 93(C):186–199
Lämmel R, Pek E, Starek J (2011) Large-scale, ast-based api-usage analysis of open-source java projects. In: SAC, pp 1317–1324
Lauinger T, Chaabane A, Arshad S, Robertson W, Wilson C, Kirda E (2017) Thou shalt not depend on me: analysing the use of outdated javascript libraries on the web. In: NDSS
Li L, Bissyandé TF, Klein J, Le Traon Y (2016) An investigation into the use of common libraries in android apps. In: SANER, pp 403–414
Li M, Wang W, Wang P, Wang S, Wu D, Liu J, Xue R, Huo W (2017) Libd: Scalable and precise third-party library detection in android markets. In: ICSE, pp 335–346
Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Di Penta M, Oliveto R, Poshyvanyk D (2013) Api change and fault proneness: a threat to the success of android apps. In: ESEC/FSE, pp 477–487
Liu C, Chen S, Fan L, Chen B, Liu Y, Peng X (2021) Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In: ICSE
Ma Z, Wang H, Guo Y, Chen X (2016) Libradar: fast and accurate detection of third-party libraries in android apps. In: ICSE, pp 653–656
Matos AS, Filho JBF, Rocha LS (2019) Splitting apis: an exploratory study of software unbundling. In: MSR, pp 360–370
McDonnell T, Ray B, Kim M (2013) An empirical study of api stability and adoption in the android ecosystem. In: ICSM, pp 70–79
Mileva YM, Dallmeier V, Burger M, Zeller A (2009) Mining trends of library usage. In: IWPSE-Evol, pp 57–62
Mileva YM, Dallmeier V, Zeller A (2010) Mining api popularity. In: Testing—practice and research techniques, pp 173–180
Mirhosseini S, Parnin C (2017) Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: ASE, pp 84–94
Nguyen HA, Nguyen TT, Wilson JrG, Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to api usage adaptation. In: OOPSLA, pp 302–321
Nguyen DC, Derr E, Backes M, Bugiel S (2020) Up2dep: android tool support to fix insecure code dependencies. In: ACSAC, pp 263–276
Ouni A, Kula RG, Kessentini M, Ishio T, German DM, Inoue K (2017) Search-based software library recommendation using multi-objective optimization. Inf Softw Technol 83(C):55–75
Patra J, Dixit PN, Pradel M (2018) Conflictjs: finding and understanding conflicts between javascript libraries. In: ICSE, pp 741–751

Plate H, Ponta SE, Sabetta A (2015) Impact assessment for vulnerabilities in open-source software libraries. In: ICSME, pp 411–420
Ponta SE, Plate H, Sabetta A (2018) Beyond metadata: code-centric and usage-based analysis of known vulnerabilities in open-source software. In: ICSME, pp 449–460
Preston-Werner T (2013) Semantic versioning 2.0.0. http://semver.org
Qiu D, Li B, Leung H (2016) Understanding the api usage in java. Inf Softw Technol 73:81–100
Quach A, Prakash A, Yan LK (2018) Debloating software through piece-wise compilation and loading. In: USENIX Security
Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: ICSM, pp 378–387
Raemaekers S, Van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: a study of the maven repository. In: SCAM, pp 215–224
Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to api deprecation?: the case of a smalltalk ecosystem. In: FSE, pp 56:1–56:11
Salza P, Palomba F, Di Nucci D, D'Uva C, De Lucia A, Ferrucci F (2018) Do developers update third-party libraries in mobile apps? In: ICPC, pp 255–265
Sawant AA, Robbes R, Bacchelli A (2016) On the reaction to deprecation of 25,357 clients of 4 + 1 popular java apis. In: ICSME, pp 400–410
Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: ICSE, pp 471–480
Sharif H, Abubakar M, Gehani A, Zaffar F (2018) Trimmer: application specialization for code debloating. In: ASE, pp 329–339
Smith N, van Bruggen D, Tomassetti F (2017) Javaparser: visited. Leanpub, oct de
Soto-Valero C, Harrand N, Monperrus M, Baudry B (2020) A comprehensive study of bloated dependencies in the maven ecosystem. CoRR arXiv:2001.07808
Teyton C, Falleri JR, Blanc X (2012) Mining library migration graphs. In: WCRE, pp 289–298
Teyton C, Falleri JR, Blanc X (2013) Automatic discovery of function mappings between similar libraries. In: WCRE, pp 192–201
Teyton C, Falleri JR, Palyart M, Blanc X (2014) A study of library migrations in java. J Softw: Evol Process 26(11):1030–1052
Thung F, Lo D, Lawall J (2013a) Automated library recommendation. In: WCRE, pp 182–191
Thung F, Wang S, Lo D, Lawall J (2013b) Automatic recommendation of api methods from feature requests. In: ASE, pp 290–300
Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot: a java bytecode optimization framework. In: CASCON, p 13
Wang Y, Wen M, Liu Z, Wu R, Wang R, Yang B, Yu H, Zhu Z, Cheung SC (2018) Do the dependency conflicts in my project matter? In: ESEC/FSE, pp 319–330
Wang C, Chen B, Liu Y, Wu H (2019a) Layered object-oriented programming: advanced vtable reuse attacks on binary-level defense. IEEE Trans Inf Forensics Secur 14(3):693–708
Wang Y, Wen M, Wu R, Liu Z, Tan SH, Zhu Z, Yu H, Cheung SC (2019b) Could I have a stack trace to examine the dependency conflict issue. In: ICSE, pp 572–583
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: ICSME, pp 35–45
Wittern E, Suter P, Rajagopalan S (2016) A look at the dynamics of the javascript package ecosystem. In: MSR, pp 351–361
Wu W, Guéhéneuc YG, Antoniol G, Kim M (2010) Aura: a hybrid approach to identify framework evolution. In: ICSE, pp 325–334
Wu W, Serveaux A, Guéhéneuc YG, Antoniol G (2015) The impact of imperfect change rules on framework api evolution identification: an empirical study. Empir Softw Eng 20(4):1126–1158
Wu W, Khomh F, Adams B, Guéhéneuc YG, Antoniol G (2016) An exploratory study of api changes and usages based on apache and eclipse ecosystems. Empir Softw Eng 21(6):2366–2412
Xing Z, Stroulia E (2007) Api-evolution support with diff-catchup. IEEE Trans Softw Eng 33(12):818–836
Xu G, Mitchell N, Arnold M, Rountev A, Sevitsky G (2010) Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: FoSER, pp 421–426
Zaimi A, Ampatzoglou A, Triantafyllidou N, Chatzigeorgiou A, Mavridis A, Chaikalis T, Deligiannis I, Sfetsos P, Stamelos I (2015) An empirical study on the reuse of third-party libraries in open-source software development. In: BCIC, pp 4:1–4:8

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: ICSME, pp 559–563

Zerouali A, Constantinou E, Mens T, Robles G, González-Barahona J (2018) An empirical analysis of technical lag in npm package dependencies. In: ICSR, pp 95–110

Zhang Y, Dai J, Zhang X, Huang S, Yang Z, Yang M, Chen H (2018) Detecting third-party libraries in android applications with high precision and recall. In: SANER, pp 141–152

Zheng W, Zhang Q, Lyu M (2011) Cross-library api recommendation using web search engines. In: ESEC/FSE, pp 480–483

Zimmermann M, Staicu CA, Tenny C, Pradel M (2019) Small world with high risks: a study of security threats in the npm ecosystem. In: USENIX Security, pp 995–1010

Affiliations

Kaifeng Huang[1] · Bihuan Chen[1] · Congying Xu[1] · Ying Wang[1] · Bowen Shi[1] · Xin Peng[1] · Yijian Wu[1] · Yang Liu[2]

Kaifeng Huang
kfhuang16@fudan.edu.cn

Congying Xu
XuCY19@fudan.edu.cn

Ying Wang
wangying18@fudan.edu.cn

Bowen Shi
18210240163@fudan.edu.cn

Xin Peng
pengxin@fudan.edu.cn

Yijian Wu
wuyijian@fudan.edu.cn

Yang Liu
yangliu@ntu.edu.sg

[1] School of Computer Science, Fudan University, D2023, Interdisciplinary Building No. 2, Songhu Road No. 2005, Yangpu District, Shanghai 201203, China

[2] School of Computer Science and Engineering, Nanyang Technological University, 02C-84, Block N4, 50 Nanyang Avenue, Singapore 639798, Singapore