An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects

Ying Wang*, Bihuan Chen*, Kaifeng Huang*, Bowen Shi*, Congying Xu*, Xin Peng*, Yijian Wu*, Yang Liu†
*School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
†School of Computer Science and Engineering, Nanyang Technological University, Singapore

***Abstract*—Third-party libraries play a key role in software development as they can relieve developers of the heavy burden of re-implementing common functionalities. However, third-party libraries and client projects evolve asynchronously. As a result, outdated third-party libraries might be used in client projects while developers are not aware of the potential risk (e.g., security bug). Outdated third-party libraries may be updated in client projects in a delayed way, and developers may be less aware of the potential risk (e.g., API incompatibility) in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 evidences on these problems, and practical insights to improve the ecosystem.**

In this paper, we contribute such a study in Java ecosystem. In particular, we conduct a *library usage analysis* (e.g., usage intensity and outdatedness) and *library update analysis* (e.g., update intensity and delay) on 806 open-source projects and 13,565 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. These analyses aim to quantify the usage and update practices and the potential risk of using and updating outdated third-party libraries with respect to security bugs from two holistic perspectives (i.e., open-source projects and third-party libraries). 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problems and potential solutions in maintaining third-party libraries (e.g., smart alerting and automated updating of outdated third-party libraries).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 indicate the usefulness of our findings, we design a smart alerting system for assisting developers to make confident decisions when updating third-party libraries. 33 and 24 open-source projects have confirmed and updated third-party libraries after receiving our alerts.

***Index Terms*—outdated libraries, security bugs**

I. Introduction

Third-party libraries allow developers to reuse common functionalities instead of reinventing the wheel, and substantially improve developers' productivity. In contrast to the benefits third-party libraries bring to software development, some problems arise due to the asynchronous 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. Old third-party library versions contain bugs, which might cause crashes or increase attack surfaces in client projects. New third-party library versions refactor code, fix bugs and add features, which might break library APIs [12, 25]. 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 also 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.

To provide concrete and comprehensive evidences 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 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 [27, 35], classes [15, 23, 36] and methods [29, 42]); some studies investigated the usages of outdated third-party libraries [26, 28]; and some studies explored the reasons for updating or not updating third-party libraries [11, 28]. 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 risk 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 improve on such situation sustainably, this paper contributes to quantitatively and holistically characterize usages, updates and risks of third-party libraries in open-source projects in Java ecosystem by answering 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?

We conduct *library usage analysis* and *library update analysis* on 806 well-maintained Java open-source projects and 13,565 third-party libraries, and conduct *library risk analysis* on the 806 projects and 544 security bugs in the 13,565 third-party libraries. These analyses were conducted based on formulated

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00014

metrics, and the data crawling and analyses in this study took about four months on a desktop machine.

Through these analyses, we aim to provide useful findings to developers and researchers. For example, 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. 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, and smart alerting and automated updating of outdated libraries.

To demonstrate the usefulness of our findings, we propose a security bug-driven alerting system to provide multiple fine-grained information for developers to make confident decisions about third-party library version updates. Our preliminary 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 and 24 open-source projects have confirmed and updated buggy third-party libraries after receiving our alerts.

In summary, this paper makes the following contributions:

- We conducted large-scale empirical analyses to quantitatively and holistically analyze usages, updates and risks of third-party libraries in Java open-source projects.
- We provided practical implications to developers and researchers, released our dataset, and proposed a prototype system to demonstrate the usefulness of our findings.

II. Related Work

Usage Analysis. Mileva et al. [35] and Kula et al. [27] analyzed usage trend and popularity of library versions. Then, Mileva et al. [36] and Hora and Valente [23] explored usage trend and popularity of library API elements (e.g., classes and interfaces) by mining *import* statements. De Roover et al. [15] analyzed library usage at API element level by AST parsing. These approaches report coarse-grained library usage mostly for one specific library, but do not report aggregated results across all libraries. Instead, we aim at method-level usage analysis for projects and libraries at the ecosystem level.

Bauer et al. [8, 9] extracted used library APIs in a project. Zaimi et al. [53] 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.

Lammel et al. [29] 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. [42] 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. [26] 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. [14] 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. [51] analyzed the popularity of npm packages and the adoption of semantic versioning in npm packages. Abdalkareem et al. [6] studied reasons and drawbacks of using trivial npm packages. For Android apps, library code is shipped into APK files, and library detection approaches [7, 32, 33, 56] have been developed. Li et al. [31] analyzed the popularity of mobile libraries. It is interesting to conduct fine-grained library usage in these ecosystems.

Update Analysis. Bavota et al. [10, 11] analyzed when and why developers updated inter-dependencies, and they found that a high number of bug fixes could encourage dependency updates, but API changes could discourage dependency updates. Fujibayashi et al. [21] explored the relationship between library release cycle and library version updates. Kula et al. [28] 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. [19] studied why developers updated mobile libraries, analyzed the practice of semantic versioning, and conducted a library updatability analysis. Salza et al. [44] analyzed mobile library categories that were more likely to be updated, and identified six update patterns. Lauinger et al. [30] and Zerouali et al. [55] measured the time lag of an outdated npm package from its latest release, and Decan et al. [17] analyzed the evolution of this time lag. Decan et al. [16] 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.

Risk Analysis. Decan et al. [18] analyzed the risk of security bugs on the npm package dependency network. Zimmermann et al. [57] measured security threats of security bugs and maintainers on the npm package dependency network. 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. [20] 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.

Cadariu et al. [13] proposed an alerting system to report Java library dependencies that have security bugs. Mirhosseini and Parnin [37] studied the usage of pull requests and badges to notify outdated npm packages. Such alerting systems, similar to some existing tools like OWASP [3], Snyk [4] and Dependabot [2], 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 [54].

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. [22] constructed a versioned ecosystem-level call graph, and checked whether a security bug can be reached through the call graph. Plate et al. [39] used dynamic analysis to check whether the methods that were changed to fix security bugs were executed by a project. Then, Ponta et al. [40] extended Plate et al.'s work [39] by combining static analysis to partially mitigate the test coverage problem. However, none of them is open-sourced. Except for Ponta et al.'s work [40], they only notify developers about security bugs, but leave developers unaware of potential risk (or effort) to update buggy library versions. Ponta et al.'s risk analysis [40] 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.

III. Empirical Study Methodology

A. Study Design

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 (see Sec. I), 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 investigates 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.

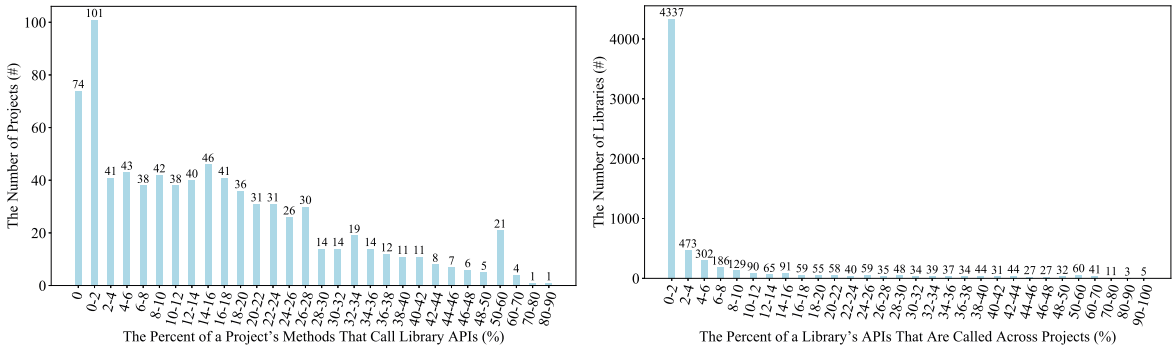
B. Corpus Selection

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 Java projects that had more than 200 stars to ensure the project quality, 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 and well-maintained projects that had commit 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 well-maintained 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 $\mathcal{P}$. 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 a total of four months.

IV. 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 p's



(a) Usage Intensity across Projects (b) Usage Intensity across Libraries

Fig. 1: Distributions of Usage Intensity across Projects and Libraries

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.

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 $\mathcal{L}_{us} = \{v.l \mid v \in \mathcal{V}_{us}\}$.

A. Usage Intensity

Definition. We define usage intensity at a fine-grained level from the perspective of a project and library: usi_p, the percent of a project p's methods that call library APIs, and usi_l, the percent of a library l's APIs that are called across projects. Different from previous studies (e.g., [23, 27, 35, 36]), we explore library usage at the method level. The most closest work is from Qiu et al. [42], but only considers JDK libraries.

To compute usi_p and usi_l, we need to extract library APIs, project methods, and API calls in project methods. We 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[1] (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 [48] 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 [46] 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 and usi_l by Eq. 1. $\mathcal{V}_l = \{v \mid v \in \tilde{\mathcal{V}}_{us} \wedge v.l = l\}$ denotes l's used versions, and usi_l takes their maximum usage intensity.

$$\begin{aligned} \forall p \in \mathcal{P}, usi_p &= \frac{|\{c.m \mid c \in \mathcal{C} \wedge c.m.p = p\}|}{|\{m \mid m \in \mathcal{M} \wedge m.p = p\}|} \\ \forall l \in \tilde{\mathcal{L}}_{us}, usi_l &= \max_{v \in \mathcal{V}_l} \frac{|\{c.a \mid c \in \mathcal{C} \wedge c.a.v = v\}|}{|\{a \mid a \in \mathcal{A} \wedge a.v = v\}|} \end{aligned} \quad (1)$$

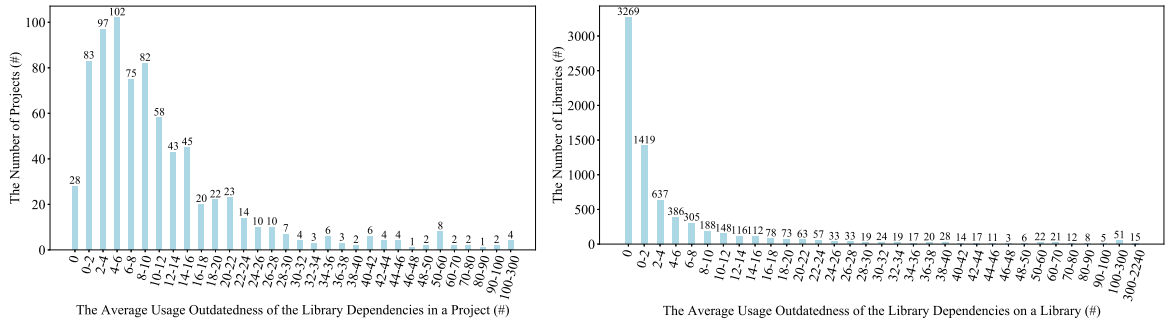
Findings. Using usi_p and usi_l, we show distributions of usage intensity across projects and libraries in Fig. 1a and 1b, where the y-axis respectively denotes the number of projects and libraries whose usage intensity falls into a range. On one hand, 74 (9.2%) projects do not call library APIs; i.e., 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. 1b.

Summary. Projects usually have a moderate dependency on library APIs. Such concrete usage dependency also reflects the required efforts 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. 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 7,345 (30.3%) library versions in $\mathcal{V}_{us}$ are snapshot versions, and 5,951 (81.0%) of them are no longer available. 344 (42.7%) projects use snapshot versions, and 161 (20.0%) and 183 (22.7%) projects respectively adopt at most and more than five snapshot versions.

Moreover, during our fine-grained analysis, we find that mul-

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



(a) Usage Outdatedness across Projects (b) Usage Outdatedness across Libraries

Fig. 2: Distributions of Usage Outdatedness across Projects and Libraries

tiple 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. 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.

B. Usage Outdatedness

Definition. We define usage outdatedness of a library dependency d, denoted as uso_d, as the number of library releases with a higher version number at the time of repository crawling. 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. 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 by Eq. 2.

$$\begin{aligned} \forall d \in \mathcal{D}_{us}, uso_d = &|\{r \mid r \in \mathcal{R}_{us} \wedge r.v.l = d.v.l \wedge \\ &r.v.ver > d.v.ver \wedge r.date < d.p.date\}| \end{aligned} \tag{2}$$

Then, we define usage outdatedness from the perspective of a project and library: uso_p, the average usage outdatedness of the library dependencies in a project p, and uso_l, the average usage outdatedness of the library dependencies on a library l. Using $\mathcal{P}$, $\tilde{\mathcal{L}}_{us}$, $\mathcal{D}_{us}$ and uso_d, we get uso_p and uso_l by Eq. 3.

$$\begin{aligned} &\forall p \in \mathcal{P}, uso_p = avg_{d \in \mathcal{D}_p} uso_d, \mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\} \\ &\forall l \in \tilde{\mathcal{L}}_{us}, uso_l = avg_{d \in \mathcal{D}_l} uso_d, \mathcal{D}_l = \{d \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\} \end{aligned} \tag{3}$$

Findings. Using uso_p and uso_l, we report distributions of usage outdatedness across projects and libraries in Fig. 2a and 2b. 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. 2a 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.

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

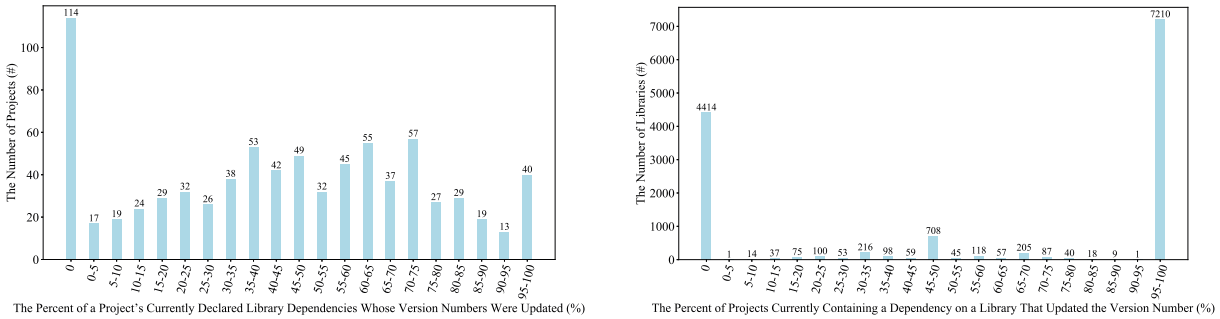
V. Library Update Analysis

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 Sec. IV) 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 $\mathcal{U}$.

A. 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



(a) Update Intensity across Projects (b) Update Intensity across Libraries

Fig. 3: Distributions of Update Intensity across Projects and Libraries

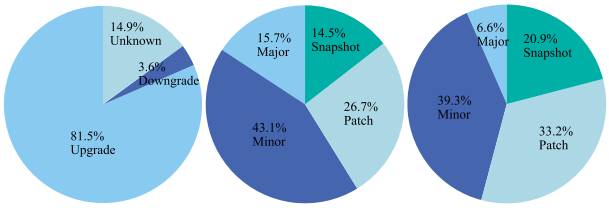
l's version number in commits. Using $\mathcal{P}$, $\mathcal{L}_{us}$, $\mathcal{D}_{us}$ and $\mathcal{U}$, we compute upi_p and upi_l by Eq 4, where $\mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\}$ and $\mathcal{P}_l = \{d.p \mid d \in \mathcal{D}_{us} \wedge d.v.l = l\}$.

$$\begin{aligned} \forall p \in \mathcal{P}, upi_p =& |\{d \mid d \in \mathcal{D}_P \wedge (\exists u \in \mathcal{U}, u.p = d.p \wedge u.f = d.f \wedge \\ & u.l = d.v.l)\}|/|\mathcal{D}_p| \\ \forall l \in \mathcal{L}_{us}, upi_l =& |\{p \mid p \in \mathcal{P}_l \wedge (\exists u \in \mathcal{U}, u.p = p \wedge u.l = l)\}|/|\mathcal{P}_l| \end{aligned} \quad (4)$$

Findings. Using upi_p and upi_l, we report distributions of update intensity across projects and libraries in Fig. 3a and 3b. 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. 3a. 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 well-maintained, 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. Defined by semantic versioning [41], 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. We identified 5,117,870



(a) Updates (b) Upgrades (c) Downgrades

Fig. 4: Update, Upgrade and Downgrade Distributions

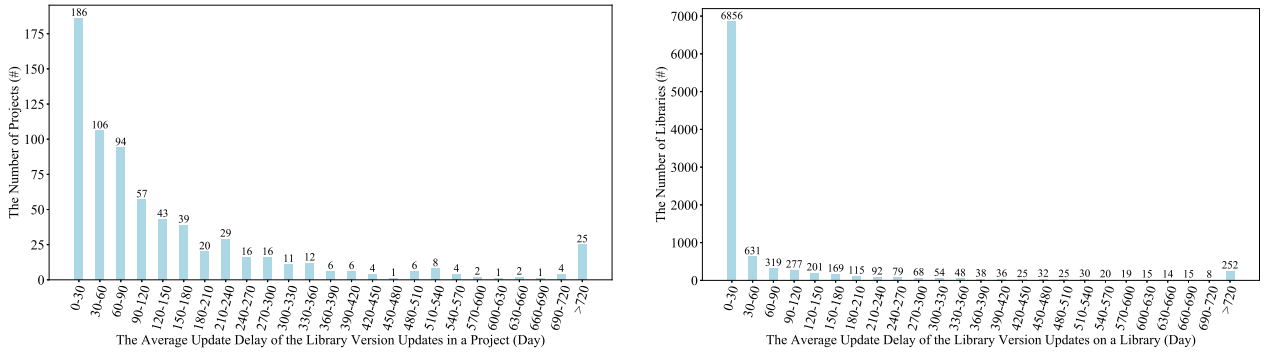
(98.1%) library version updates from $\mathcal{U}$ whose version numbers start with $X.Y$ or $X.Y.Z$, denoted as $\hat{\mathcal{U}}$.

On one hand, we explore whether developers upgrade or downgrade a library version. As reported in Fig. 4a, 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 Fig. 4b and 4c. First, 14.5% of upgrades 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.

B. 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 $\mathcal{R}_{up}$.



(a) Update Delay across Projects (b) Update Delay across Libraries

Fig. 5: Distributions of Update Delay across Projects and Libraries

From $\mathcal{R}_{up}$, we had 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 Eq. 5.

$$\begin{aligned} &\forall u \in \tilde{\mathcal{U}}, upd_u = u.com.date - r.date, \\ &r \in \mathcal{R}_{up} \wedge r.v.l = u.l \wedge r.v.ver = u.ver_2 \end{aligned} \tag{5}$$

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 Eq. 6.

$$\begin{aligned} \forall p \in \mathcal{P}_{up}, upd_p = &avg_{u \in \mathcal{U}_p} upd_u, \mathcal{U}_p = \{u \mid u \in \tilde{\mathcal{U}} \wedge u.p\} \\ \forall l \in \mathcal{L}_{up}, upd_l = &avg_{u \in \mathcal{U}_l} upd_u, \mathcal{U}_l = \{u \mid u \in \tilde{\mathcal{U}} \wedge u.l = l\} \end{aligned} \tag{6}$$

Findings. Using upd_p and upd_l, we show distributions of update delay across projects and libraries in Fig. 5a and 5b. 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. 5a 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.

VI. Library Risk Analysis

Several tools (e.g., Black Duck [1], Veracode [5], SAP [40], OWASP [3], Snyk [4] and Dependabot [2]) 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 [19]. 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 [5] and then crawling the metadata (e.g., affected library releases and security patch) of each security bug. 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).

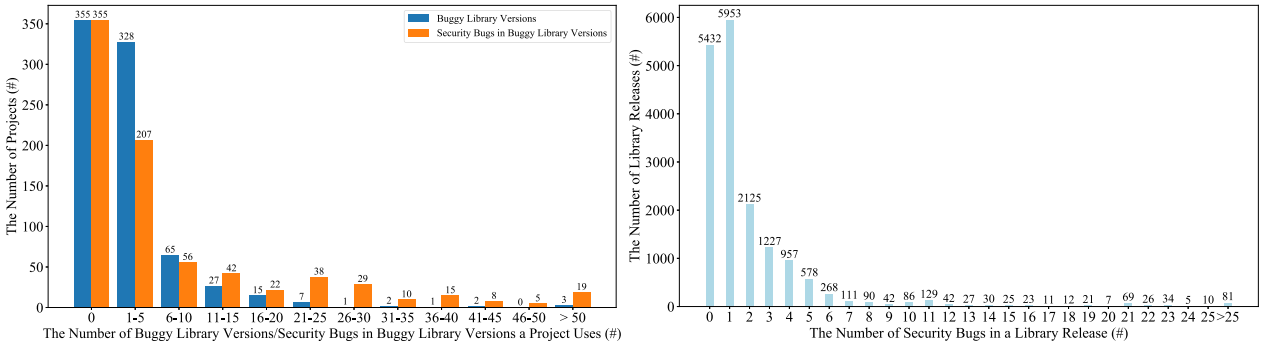
We focused on the security bugs in the libraries (i.e., $\mathcal{L}_{us}$) used by the 806 projects. We applied *bug-crawler* to each library $l \in \mathcal{L}_{us}$, and collected 544 security bugs, affecting 252 libraries. These 252 libraries have 17,421 library releases (computed from $\mathcal{R}_{us}$), 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., $\tilde{\mathcal{R}}_{ri} = \{g.r \mid g \in \mathcal{G}\}$).

A. 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 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 Eq. 7, where $\mathcal{D}_p = \{d \mid d \in \mathcal{D}_{us} \wedge d.p = p\}$.

$$\begin{aligned} &\forall p \in \mathcal{P}, usr_p^1 = |\{d.v \mid d \in \mathcal{D}_p \wedge (\exists g \in \mathcal{G}, g.r.v = d.v)\}| \\ &\forall p \in \mathcal{P}, usr_p^2 = \sum_{d \in \mathcal{D}_p} |\{g.b \mid g \in \mathcal{G} \wedge g.r.v = d.v\}| \\ &\forall r \in \mathcal{R}_{ri}, usr_r = |\{g.b | g \in \mathcal{G} \wedge g.r = r\}| \end{aligned} \tag{7}$$

Findings. Using usr_p^1, usr_p^2 and usr_r, we present distributions of usage risk across projects and library releases in Fig. 6a and 6b. 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.



(a) Usage Risk across Projects (b) Usage Risk across Library Releases

Fig. 6: Distribution of Usage Risk across Projects and Library Releases

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.

B. 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 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 Eq. 8.

$$\begin{aligned}\forall p \in \mathcal{P}, upr_p^1 &= |\{c.a | c \in \mathcal{C} \wedge c.m.p = p \wedge (\exists g \in \mathcal{G}, g.r.v = c.a.v)\}| \\ \forall p \in \mathcal{P}, upr_p^2 &= |\{c | c \in \mathcal{C} \wedge c.m.p = p \wedge (\exists g \in \mathcal{G}, g.r.v = c.a.v)\}|\end{aligned} \tag{8}$$

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 Eq. 9, 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).

$$\begin{aligned}\forall r \in \tilde{\mathcal{R}}_{ri}, upr_r^1 &= |\{a.api \mid a \in \mathcal{A}_r\} - \{a.api \mid a \in \mathcal{A}_{r_{safe}}\}| \\ \forall r \in \tilde{\mathcal{R}}_{ri}, upr_r^2 &= |\{a.api \mid a \in \mathcal{A}_r\} \sqcap \{a.api \mid a \in \mathcal{A}_{r_{safe}}\}|\end{aligned} \tag{9}$$

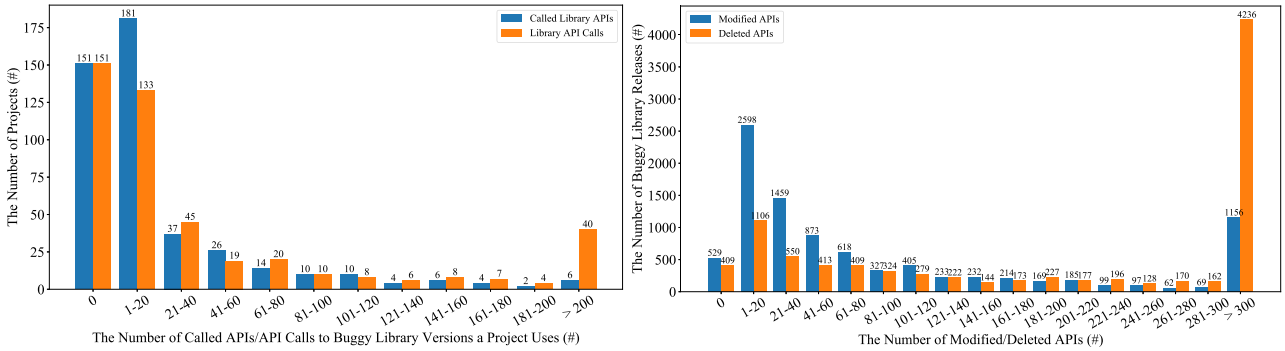
Findings. Using upr_p^1, upr_p^2, upr_r^1 and upr_r^2, we give distributions of update risk across projects and buggy library releases in Fig. 7a and 7b. On one hand, of the 451 projects that adopt buggy library versions, 151 (33.5%) projects do not call APIs in used buggy library versions, 181 (40.1%) 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. 7b. 4,586 (38.3%) buggy library releases have less than 40 APIs modified in the safe library 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.

VII. Discussion

A. 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



(a) Update Risk across Projects (b) Update Risk across Library Releases

Fig. 7: Distribution of Update Risk across Projects and Library Releases

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 [52] or broaden attack surface (e.g., code reuse attack) [49]. 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 [43, 45] have been designed for C/C++; Soto-Valero et al. [47] removed unused transitive libraries as a whole but did not target partially-used libraries; and Matos et al. [34] studied the possibility of splitting libraries into bundles based on the API usages of a set of client projects.

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 [38, 50]. 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. Inspired by our work, Huang et al. [24] have proposed an interactive, effort-aware library version harmonization approach.

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. 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.

B. Application for Usefulness Demonstration

Based on the implication on smart alerting, we design a prototype of a security bug-driven alerting system for buggy libraries. It consists of two main components: *risk analysis* and *effort analysis*; and it has two databases: bug database and library database. The bug database currently has 544 security bugs collected in our risk analysis (see Sec. VI) 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.

Our risk analysis is implemented to decide whether a project directly or indirectly calls buggy library methods. It first uses JavaParser [46] with type binding to extract library API calls in the project. Then, it uses Soot [48] to construct the call graphs of these called library APIs. Finally, it checks whether the library methods in each call graph contain buggy library methods from our bug database. If yes, we consider the called library API as *risky* and affecting the project (i.e., the corresponding security bug could in the execution path of the project). After this analysis, we can report the number of security bugs that affect the project in each buggy library version (i.e., NB), the number of risky library APIs called in the project (i.e., NA), and the number of calls to risky library APIs in the project (i.e., NC). These three metrics provide developers with the risk and impact of adopted buggy library versions.

Our effort analysis is implemented to suggest the new library versions and the updating effort. 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. 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 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). These metrics measure the updating effort on the suggested library version.

We have run our alerting system against the 451 projects that use buggy libraries (see Sec. VI) 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 I for 10 projects due to space limitation, where column *P* lists the projects, *BL* reports the 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 projects and other suggested library versions are at https://3rdpartylibs.github.io.

Our preliminary 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. 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. 14 issues are still under analysis. Developers ask for pull request or thank for our issue without any further action in 7 issues. 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 I. 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.

TABLE I: 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

C. Threats to Validity

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.

Subject Representativity. We use well-maintained 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 well-maintained projects is needed to compare the findings. Besides, 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.

VIII. 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 released our dataset and source code of our tools at https://3rdpartylibs.github.io.

Acknowledgment

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

REFERENCES

[1] Black duck. [Online]. Available: https://www.blackducksoftware.com
[2] Dependabot. [Online]. Available: https://dependabot.com
[3] Owasp dependency-check. [Online]. Available: https://owasp.org/www-project-dependency-check/
[4] Snyk. [Online]. Available: https://snyk.io
[5] Veracode. [Online]. Available: https://www.veracode.com
[6] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *FSE*, 2017, pp. 385–395.
[7] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *CCS*, 2016, pp. 356–367.
[8] V. Bauer and L. Heinemann, "Understanding api usage to support informed decision making in software maintenance," in *CSMR*, 2012, pp. 435–440.
[9] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *ICSM*, 2012, pp. 483–492.
[10] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *ICSM*, 2013, pp. 280–289.
[11] ——, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.
[12] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *FSE*, 2016, pp. 109–120.
[13] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *SANER*, 2015, pp. 516–519.
[14] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *ICSE*, vol. 2, 2015, pp. 109–118.
[15] C. De Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *ICPC*, 2013, pp. 152–161.
[16] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *SANER*, 2017, pp. 2–12.
[17] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *ICSME*, 2018, pp. 404–414.
[18] ——, "On the impact of security vulnerabilities in the npm package dependency network," in *MSR*, 2018, pp. 181–191.
[19] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *CCS*, 2017, pp. 2187–2200.
[20] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *CSMR-WCRE*, 2014, pp. 64–73.
[21] D. Fujibayashi, A. Ihara, H. Suwa, R. G. Kula, and K. Matsumoto, "Does the release cycle of a library project influence when it is adopted by a client project?" in *SANER*, 2017, pp. 569–570.
[22] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *ICSE-NIER*, 2018, pp. 101–104.
[23] A. Hora and M. T. Valente, "apiwave: Keeping track of api popularity and migration," in *ICSME*, 2015, pp. 321–323.
[24] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," in *ESEC/FSE*, 2020.
[25] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *ICSE*, 2011, pp. 151–160.
[26] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *SANER*, 2015, pp. 520–524.
[27] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, "An exploratory study on library aging by monitoring client usage in a software ecosystem," in *SANER*, 2017, pp. 407–411.
[28] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.
[29] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *SAC*, 2011, pp. 1317–1324.
[30] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS*, 2017.
[31] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *SANER*, 2016, pp. 403–414.
[32] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: Scalable and precise third-party library detection in android markets," in *ICSE*, 2017, pp. 335–346.
[33] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *ICSE*, 2016, pp. 653–656.
[34] A. S. Matos, J. B. F. Filho, and L. S. Rocha, "Splitting apis: An exploratory study of software unbundling," in *MSR*, 2019, pp. 360–370.
[35] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *IWPSE-Evol*, 2009, pp. 57–62.
[36] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Testing – Practice and Research Techniques*, 2010, pp. 173–180.
[37] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *ASE*, 2017, pp. 84–94.
[38] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," in *ICSE*, 2018, pp. 741–751.
[39] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *ICSME*, 2015, pp. 411–420.
[40] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *ICSME*, 2018, pp. 449–460.
[41] T. Preston-Werner, "Semantic versioning 2.0. 0," *http://semver. org*, 2013.
[42] D. Qiu, B. Li, and H. Leung, "Understanding the api usage in java," *Information and software technology*, vol. 73, pp. 81–100, 2016.
[43] A. Quach, A. Prakash, and L. K. Yan, "Debloating software through piece-wise compilation and loading," in *USENIX Security*, 2018.
[44] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *ICPC*, 2018, pp. 255–265.
[45] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *ASE*, 2018, pp. 329–339.
[46] N. Smith, D. van Bruggen, and F. Tomassetti, "Javaparser: Visited," *Leanpub, oct. de*, 2017.
[47] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *CoRR*, vol. abs/2001.07808, 2020.
[48] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON*, 1999, pp. 13–.
[49] C. Wang, B. Chen, Y. Liu, and H. Wu, "Layered object-oriented programming: Advanced vtable reuse attacks on binary-level defense," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 693–708, 2019.
[50] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *ESEC/FSE*, 2018, pp. 319–330.
[51] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *MSR*, 2016, pp. 351–361.
[52] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *FoSER*, 2010, pp. 421–426.
[53] A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, A. Mavridis, T. Chaikalis, I. Deligiannis, P. Sfetsos, and I. Stamelos, "An empirical study on the reuse of third-party libraries in open-source software development," in *BCIC*, 2015, pp. 4:1–4:8.
[54] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *ICSME*, 2018, pp. 559–563.
[55] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *ICSR*, 2018, pp. 95–110.
[56] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *SANER*, 2018, pp. 141–152.
[57] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security*, 2019, pp. 995–1010.