A Qualitative Study of Dependency Management and Its Security Implications

Ivan Pashchenko
ivan.pashchenko@unitn.it
University of Trento, IT

Duc-Ly Vu
ducly.vu@unitn.it
University of Trento, IT

Fabio Massacci
fabio.massacci@unitn.it
University of Trento, IT

ABSTRACT

Several large scale studies on the Maven, NPM, and Android ecosystems point out that many developers do not often update their vulnerable software libraries thus exposing the user of their code to security risks. The purpose of this study is to qualitatively investigate the choices and the interplay of *functional and security concerns* on the developers' overall decision-making strategies for *selecting, managing, and updating software dependencies.*

We run 25 semi-structured interviews with developers of both large and small-medium enterprises located in nine countries. All interviews were transcribed, coded, and analyzed according to applied thematic analysis. They highlight the trade-offs that developers are facing and that security researchers must understand to provide an effective support to mitigate vulnerabilities (for example bundling security fixes with functional changes might hinder adoption due to lack of resources to fix functional breaking changes).

We further distill our observations to actionable implications on what algorithms and automated tools should achieve to effectively support (semi-)automatic dependency management.

KEYWORDS

Dependency management, security, vulnerable dependencies, qualitative study, interviews

ACM Reference Format:
Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications . In *to appear in Conf. on Comp. and Comm. Sec. - CCS'20.* ACM, New York, NY, USA, 20 pages. https://doi.org/tbc

1 INTRODUCTION

Vulnerable dependencies are a known problem in the software ecosystems [25, 33], because free and open-source software (FOSS) libraries are highly interconnected, and developers often do not update their project dependencies, even if they affected by known security vulnerabilities [11, 25].

A handful of studies report that developers do not update dependencies in their projects since they are not aware of dependency issues [6] or do not want to break their projects [7, 16]. Although functionality and security appear to be essential factors that affect developers' decisions [3], those studies mainly focus on functionality aspects, and therefore, provide limited insights on the impact of security concerns on developers' reasoning.

Other studies also show this tension between functionality and security. On the Android ecosystem, mobile app developers do not consider security as a top-priority task [11]. A later study by the same group [20] explained the reason behind it as a major clash with functionality: the 'easy' updates would actually break around 50% of dependent projects.

A key observation is that several of those studies are about ecosystems that do not feature a central place for storing and managing app dependencies. Developers with central dependency management system, like Maven, NPM, or PyPI, might have a very different approach towards dependencies of their applications.

For example, an initial quantitative study of the Maven ecosystem [25] analyzed more than 4600 Github repositories and provided yet another evidence that developers keep their project dependencies outdated. However, a later study [33] showed that several of the reported vulnerabilities were in test/development libraries (i.e., not shipped with the product), and therefore, irrelevant. So, not updating the library was not due to a breaking conflict with functionality but a perfectly rational decision.

The goal of our paper is to provide a *sound qualitative analysis* of the motivation of developers between the rigid format of surveys (e.g., [11]) and the anecdotal examples that complement quantitative studies on dependencies (e.g., [25]).

Following the process of semi-structured interviews we have investigated the following research questions:

- RQ1: How do developers select dependencies to include into their projects, and where (if at all) does security play a role?
- RQ2: Why do developers decide to update software dependencies and how do security concerns affect their decisions?
- RQ3: Which methods, techniques, or automated analysis tools (e.g., Github Security Alerts) do developers apply while managing (vulnerable) software dependencies?
- RQ4: Which mitigations do developers apply for vulnerable dependencies with no fixed version available?

This paper has the following contributions:

- qualitative investigation of the choices and the interplay of *functional and security concerns* on the developers' overall decision-making strategies for *selecting, managing, and updating software dependencies*
- possible implications for research and practice to help improving the security and the support of (semi-)automatic dependency management.

Our qualitative study is based on semi-structured interviews with 25 enterprise developers, who are involved in development of web, embedded, mobile, or desktop applications. Some interviewees

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/tbc

also create their own libraries (i.e. supply dependencies to other projects) but, to keep focus, our interviews investigated their role in the demand of libraries. The interviewees have at least three years of professional experience at various positions spanning from regular developers to company CTOs, including the coordinator of a Java Users' Group and a lead developer of a Linux distribution. They come from 25 companies located in nine different countries.

Each interview (lasting 30' on average) was recorded and transcribed. The transcripts were anonymized and sent back to the interviewees for confirmation. Each conversation was then coded along the lines of applied thematic analysis to provide a quantitative assessment of the collected qualitative data.

This paper illustrates the insights with quotations from the interviewees to provide a better grasp of developers' reasoning while managing software dependencies and how security concerns affect their decisions[1]. After completing the analysis, we also returned the overall findings to the participating developers to check that we have not misinterpreted their thoughts.

2 TERMINOLOGY

We rely on the terminology established between practitioners:

- A *library* is a separately distributed software component, which functionality might be reused by other software projects.
- A *dependency* is a library some functionality of which is reused by other software projects. Although "dependency" logically relates to a relation, we adopt the term as it is used (and abused) by software developers[2] so we can correctly communicate the meaning of their thoughts delivered in the form of quotations later in the paper.
- *Dependency management* is the process of modification of the configuration of a project by updating (i.e., adoption of new versions of currently used dependencies) or adding/removing dependencies. To manage dependencies, software developers only need to modify own code of their project.
- *Dependency maintenance* implies access and modification of the source code of project dependencies. For dependency maintenance, developers typically have to access the dependency source code repositories (e.g., Github repositories) and contribute to the dependency projects (e.g., by creating pull requests of the proposed changes).

3 BACKGROUND

To understand the state-of-the-art we looked in Elsevier Scopus for papers published between 2006 and 2019 that report findings on one of the code groups identified in Section 2 and that mention surveys, interviews, case or qualitative studies, etc. After a preliminary selection of 159 articles, we narrowed it down to 25 (including suggestions from anonymous reviewers). A comparative analysis of all papers is available in Table 7 in the Appendix.

3.1 Dependency management and mitigation of dependency issues.

Many empirical studies [2, 8, 9, 19, 22, 25, 26, 28, 33, 35, 36, 44] investigate the topic of security vulnerabilities introduced by software dependencies. Cox et al. [9] introduced the notion of "dependency freshness" and reported that fresh dependencies are more likely to be free from security vulnerabilities. However, various studies of different dependency ecosystems, i.e., Java [25, 33], JavaScript [19, 22, 44], Ruby [22], Rust [22], etc., provide the evidence that developers often do not update software dependencies.

Derr et al. [11] surveyed Android developers to identify their usage of libraries and requirements for more effective library updates. When updating their app libraries, developers consider bug fixing to be the most important reason while security played a minor role. Developers are wary of updating their dependencies if they work as intended. A follow up quantitative study [20] found that the most likely reason that stops developers from updating dependencies are breaking changes due to deprecated functions, changed data structures, or entangled dependencies between different libraries and even the host app. Limited insights are provided on the developers' motivations for performing an update of each kind (functionality or security). Moreover, since the study presented the findings from the Android ecosystem that does not have a central dependency management system, like Maven Central, NPM, or PyPI, the results might not generalize to the developers of other ecosystems.

Considering the ecosystems that have a centered dependency management system, Haenni et al. [16] reported the impact of changes to be one of the main developers' concerns when updating their dependencies. Later, Bogart et al. [6] observed that developers often find it challenging to be aware of potentially significant changes to the dependencies of their projects and prefer to wait for the dependencies to break rather than act proactively about them. In their later study [7], breaking changes are the main factor that prevents developers from updating their project dependencies. Also, the authors observed that developers sometimes do not update dependencies in their projects even though this is recommended by the policy of their company. However, the studies took into account only the effect of functionality issues introduced by dependencies and did not consider the impact of security concerns.

Kula et al. [25] is the only paper to study the influence of security advisories on dependency updatability we are aware of. The authors found no correlation between the presence of security advisories and dependencies update on FOSS projects in Github. An anecdotal survey of developers showed that some were not aware of security advisories and existing security fixes. However, the authors only surveyed FOSS developers who did not update dependencies of their projects, and therefore, the reported results might not generalize when applied to all developers (e.g., enterprise developers). Also, the study reported no in-depth qualitative analysis (e.g., no coding, publishing only some quotations from email responses). Moreover, a recent study by Pashchenko et al. [33] suggested that the results presented in [25] might be affected by false positives as the authors considered vulnerable dependencies used only for testing purposes.

[1]As this is a purely qualitative study, the presented findings may not necessarily generalize to other ecosystems and the proposed implications encourage additional investigations to confirm their validity.
[2]https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Summary: Current qualitative dependency studies suggest that dependency issues might affect developers' decisions, however, the studies focus mainly on functionality issues, and therefore, provide limited insights on whether security concerns have any impact on the developers' decisions for the selection of new dependencies to be included in software projects (RQ1), their further management (RQ2), and how developers mitigate bugs and vulnerabilities in case there is no fixed version of a dependency available (RQ4).

3.2 Technologies/tools for automating the software development process.

Several papers studied the adoption of static analysis tools that allow developers to identify both functionality and security issues in the own code of their software projects. For example, Vassallo et al. [43] investigated the impact of the development context on the selection of static analysis tools. Tools are adopted in three primary development contexts: local environment, code review, and continuous integration. However, Johnson et al. [21] identified that lack of or weak support for teamwork or collaboration, a high number of false positives, and low-level warnings are the main barriers that prevent developers' adoption. These studies clarify some issues that developers face while using automated tools but might not apply to the developers' perceptions of using dependency analysis tools that do not actually analyze code.

Mirhosseini and Parnin [31] is the only study that analyzed how developers use dependency analysis tools. The authors quantitatively studied whether automated pull requests encourage developers to update their dependencies: projects that used automatically generated pull requests or badges updated dependencies more frequently, but developers also *ignored* almost two-thirds of such pull requests due to potential breaking changes. As the study considered functionality aspects, we don't know whether security may change the developers' reactions to automated notifications. The study focused on JavaScript developers who used *greenkeeper.io* as a dependency management tool, so its findings might not apply to other dependency management tools.

Summary: The qualitative studies of technologies and tools for automating the software engineering process report interesting observations regarding the developers' experience, but current studies that involve dependency analysis tools focus mostly on functionality aspects, and therefore, provide limited insights on how developers can use them to discover and mitigate security issues introduced by software dependencies (RQ3, RQ4).

3.3 Information needs and decision making during software development

Several studies [5, 16, 23, 25, 27, 34, 38, 39] describe the information needs and decision-making strategies of industrial practitioners. For example, Unphon and Dittrich [41] observed that an architect or a key developer plays a central role in designing and revising software architecture. Pano et al. [32] reported that a combination of four actors (customer, developer, team, and team leader), performance size, and automation drive the choice of a JavaScript framework. Again these papers capture information needs and behavioral patterns of enterprise developers but do not report security concerns on decision-making preferences.

Assal and Chiasson [3] surveyed software developers to study the interplay between developers and software security processes. The authors observed that the security effort allocated to the implementation stage is significantly higher than in the code analysis, testing, and review stages. The paper provides a good insight into human aspects of developers' behavior towards their own code but does not tackle software dependencies (i.e., other people's code).

Linden [42] studied the developers' perception of security in various development activities, both with surveys and in a laboratory exercise. The authors found that developers mainly consider security in coding activities, such as writing code or selection of external SDKs. However, the study provides limited insights about developers' reasoning while working with dependencies. Moreover, the findings are reported based on observing and surveying only Android developers, and therefore, might not apply for other development environments, especially those having a central dependency management system, like NPM or PyPI.

Summary: The studies on information needs provide useful insights on developers' decision-making strategies, however, the existing studies do not show how the developers' actions and decisions change in the presence of security issues introduced by software dependencies (RQ1 and RQ2).

Table 1: Descriptive statistics of the number of interview participants in the selected papers

Note, that we do not report the data for the mailing lists study type, since we have participants number only for one study: Kula et al. [25] involved 16 developers in their study, while Sharif et al. [38] studied mailing lists from 6 FOSS projects but did not report the number of participants.

	#Papers	#Developers				
Study type		μ	σ	median	Q25%	Q75%
interviews	16	12.1	6.4	12	6.8	15
surveys	7	119.1	92.5	116	52	163
observations	3	8.7	6.4	6	5	11

4 METHODOLOGY

Our goal is to study the developers' perceptions of software dependencies and the effect of security concerns on their decisions. Online surveys or controlled experiments force the investigator's point of view on the arguments of interest, and therefore, may blur the developers' opinions. Instead, semi-structured interviews suited best for our goals [46]. Being open, they allow new ideas to be brought up during the interview as a result of what an interviewee says, and it is indeed used by most of the selected studies (15 out of 22 studies in Table 7).

Table 1 shows the descriptive statistics of the number of participants in the papers discussed in Section 3. We observe that an interview-based study, on average, employs 13 developers. At the same time, 75% of the selected papers report results from less than 17 interviews. Moreover, the studies typically report interview results from developers of a single company or the same community of developers. This may potentially introduce some bias since developers may share the same development strategies and approaches.

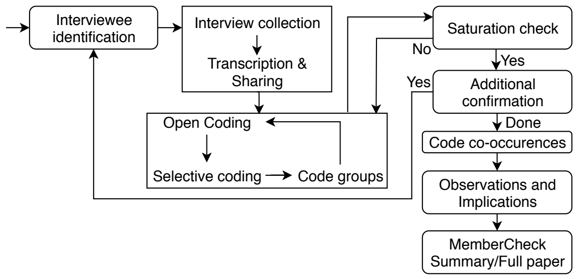

Figure 1: Research Stream

Table 2: Interviewees in our sample

The table describes interviewees in our sample. We report positions, professional experience, and primary languages as communicated during the interviews. By location, we specify the current country of the developer workplace. We have clustered the companies as follows: free and open-source project (FOSS project), large enterprise (LE), small and medium-sized enterprise (SME), and user group (UG).

#	position	comp. type	country	exper. (years)	dev. type	primary languages
#1	CTO	SME	DE	3+	web	Python, JS
#2	Moderator	UG	IT	10+	web	Java
#3	Developer	LE	IT	10+	web	Java, JS
#4	CEO	SME	SI	7+	web/desktop	Python, JS
#5	Developer	SME	NL	3+	web/desktop	Python
#6	Freelancer	SME	RU	3+	mobile	Python, JS
#7	Developer	SME	DE	5+	web/desktop	Python, JS
#8	Developer	LE	RU	4+	web	Python, JS
#9	CTO	SME	IT	4+	web	JS
#10	Developer	LE	DE	10+	embedded	C/C++
#11	Developer	LE	VN	5+	embedded	C/C++
#12	Developer	SME	DE	4+	web	Java, Python
#13	Team lead	LE	RU	10+	desktop	JS
#14	Developer	SME	RU	4+	web	Java
#15	Project Leader	FOSS	UK	10+	embedded	Python, C/C++
#16	Developer	SME	IT	8+	web	Java
#17	Developer	LE	VN	3+	web/desktop	Java
#18	Senior Software Engineer	LE	IT	10+	embedded	Python, C/C++
#19	Developer	SME	RU	3+	web	Java
#20	Security Engineer	LE	DE	3+	web/desktop	JS
#21	Developer	SME	HR	3+	web	JS
#22	Developer	SME	IT	8+	web	JS
#23	Developer	LE	IT	9+	web	Java
#24	Full stack developer	SME	IT	3+	web	JS, Python
#25	Developer	SME	ES	3+	embedded	C/C++

4.1 Recruitment of participants

As a source for finding software developers, we referred to local development communities. We used the public channels for these groups to post our call for interviews as well as contacted their reference people. Then we applied the snowball sampling approach [12] to increase the number of interviewees by asking the respondents to distribute our call within their friends and other development communities they are involved in. To overcome the potential bias of the snowballing approach, for our interviews, we selected developers with various roles and responsibilities, each representing a different company and often a different country.

In our study, we recruited enterprise developers working in at least one of the following programming languages: C/C++, Java, JavaScript, or Python. The interviewees have at least three years of professional working experience (with more than ten years for six developers) and held various positions, spanning from regular and senior developers to team leaders and CTOs. Some of the participants are involved in internal/corporate development, while others are working on web, embedded, mobile, or desktop applications. In total, we interviewed 30 developers[3] and eventually retained 25 for the analysis distributed over 25 different companies located in nine countries[4]. Table 2 summarizes the key demographics of the interviewees in our sample.

4.2 Interview process

To collect primary data, we had interview sessions lasting approximately 30 minutes. We met personally the interviewees who reside in our city and scheduled remote meetings with others via Skype or Webex. We offered no monetary compensation for the interviewees as the interviewed developers are highly skilled professionals who are very unlikely to be motivated by a compensation we could offer. Instead, we proposed them to share their expert opinions on the topic interesting for them. We followed XXX Ethical Review Board procedure for the management of consent and processing of data[5]. We explained that all interviews would be reported anonymously, and neither personal nor company identifiable data would be made available. No personal data was collected.

We adopted the semi-structured interview type for our research and framed our questions to allow developers to define the flow of the discussion, i.e., followed the “grand tour interviews” principle [17]. Still, we made sure all interviews included the following parts[6] (not necessarily in that order):

- *Introduction* - interviewer describes the context and motivation for the study;
- *Developer's self-presentation* - developer (D) presents her professional experience and the context of her current activities;
- *Selection of new dependencies* - D describes selection and inclusion of new dependencies into her software projects;
- *Updating dependencies* - D explains the motivations and insights of updating dependencies in her projects, i.e., when it is the right time to update, how often she updates dependencies, and if there is any routine or regulation regarding the dependency update process in her company;
- *Usage of some automated tool for dependency analysis* - D describes an automatic tool (if it is used) that facilitates dependency analysis process in her projects, and provides some general details about the integration of this tool into her development process;
- *Mitigation of dependency issues* - D describes how she addresses issues in dependencies (e.g., bugs or vulnerabilities);

[3]We could have three more developers to participate in our study. They initially agreed to let us observe their actions while analyzing software dependencies, but then the process got stuck at the stage of selecting the analysis target. Their companies were unwilling to let us study their internal libraries without a legal agreement in place, while analysis of third-party FOSS libraries was not interested in the developers.

[4]Four interviewees were not confident enough to speak about software dependencies in their projects since they just came into the company. Another developer said that due to the company policy, they do not use software dependencies. Hence, we discarded five interviews from our analysis.

[5]Url not provided as this would disclose the institution.

[6]After the interviews were completed, two researchers checked that an individual interview contains all elements mentioned above, by coding the interview transcripts with the codes corresponding to each interview part. Five interviews (#6, #12, #14, #21, #23) do not contain the *Usage of some automated tool for dependency analysis* part, since the interviewees mentioned that they perform dependency management manually.

- *Other general comments regarding dependency management* - this includes some general perceptions, comments, or recommendations that D may give on the process of dependency management and, in particular, about the security issues introduced by software dependencies.

There were two interviewers at each interview session. Each interviewer had a list of the interview parts mentioned above and crossed off a part if she subjectively counted it as discussed. An interview finished as soon as all the parts became crossed.

Each interview was recorded and transcribed. The transcripts were anonymized and sent back to the developers for confirmation[7]. The recordings were then destroyed for preserving the possibility of identifying the interviewees.

4.3 Interview coding and analysis

To analyse the interviews, we have adopted the *applied thematic analysis* [14]. Figure 1 summarizes our approach. It follows the principle of emergence [13], according to which data gain their relevance in the analysis through a systematic generation and iterative conceptualization of codes, concepts, and code groups. Data is analyzed, broken into manageable pieces (codes), and compared for similarities and differences. Similar concepts are grouped under the same conceptual heading (code group). Code groups are composed in terms of their properties and dimensions, and finally, they provide the structure of the analysis [40].

The first phase of analysis (open coding) consists of collecting the critical point statements from each interviewee transcript; a code summarizing the key points in a few words is assigned to each key point statement. The interviewees are numbered #1 to #25. Two researchers independently followed the "iterative process" described by Saldaña [37] to code the transcribed interviews[8]. Then they looked together at the resulting codes and agreed on the common code structure, which was reviewed by a third researcher not involved in the preliminary coding process. So after each iteration, we had a complete agreement on the codes and code groups by the three researchers. Each time we reviewed the resulted codes, we have also performed a check whether we have achieved a saturation of the reported observations [30], i.e., if the interviewees discuss the same concepts. After concluding that saturation is achieved, we interviewed additional developers to control the stability of our observations (Additional confirmation step in Figure 1).

We started the coding process as soon as we had ten interviews. At first, we created 345 quotations and assigned 138 codes to them. During the first six iterations, we were consolidating both quotations and codes by looking at quotations and merging codes on close topics. This resulted in 151 quotations with 28 codes assigned to them. On the next stages, we have added 15 more interviews, which significantly enlarged the number of quotations (533 quotations on the 11th iteration). While adding them, we realized that there was one irrelevant code (Scala)[9], so we deleted it. Hence, there were 27 codes on the 10th iteration. Then we have added quotations and codes for the developer roles (SME, LE, FOSS, or UG developer), which resulted in 31 codes and 574 quotations on the 11th iteration. On the last step of the coding process, we have added the codes corresponding to the interview process parts. Hence, we have ended up with four codes that correspond to developer roles, six codes for interview process parts, and 27 codes for developer answers assigned to 829 interview quotes.

To validate our observations and implications, we shared the one-page summary of this study, along with the full version of the paper with the interviewees. We asked them to validate if the results correspond to their expectations (last step in Figure 1).

4.4 Final Code Book

To analyze the developer interviews, we introduce the following code groups that tag a topic of a conversation:

- *Dependencies* code group indicates that a fragment of a conversation is specific to software dependencies rather than, for example, to own code of a software project.
- *Language* code group labels conversations specific to a particular programming language (e.g., Java vs. Python) rather than discussions of common issues relevant to the software engineering process in general. A different code is used for each programming language (C/C++, Java, JS, Python).

Additionally, we cluster similar topics in the conversations and assign them to the corresponding code groups as follows:

- The *Attitude* code group captures a qualitative assessment of a fact reported by a developer. E.g., a developer expresses her likes, dislikes, or recommendations regarding particular steps of dependency management.
- The *Context* code group captures background information about the reported issues, such as whether an issue relates to functionality or security.
- The *Issues* code group includes discussions about functionality flaws or weaknesses, like bugs or breaking changes.
- The *Operations* code group captures specific modifications of project own code or its dependencies. For example, a conversation fragment discusses dependency management or dependency maintenance.
- The *Process* code group captures the presence of established development practices followed by developers. For example, a conversation fragment describes how a developer team automates the dependency management of their project.

Table 3 summarizes the resulted list of codes in our study while Figure 2 shows number of code occurrences. Notice that *the same sentence may be labeled by several codes*:

```
We have a contract that we inform our clients once a month. If
we have discovered vulnerability today the client would know
about it in a month. Of course, if the vulnerability is not
critical. If it is critical, we inform our client immediately
as soon as we gather the information.. (#5)
```

is associated to codes: *dependency management*, *python* (as the developer is talking about Python), *requirements*, *security*.

5 FINDINGS

We have checked[10] whether practices established within development communities affect our findings. Considering the per-language code distributions, we observed that Java, JavaScript, and Python

[7]Except for the cases when the developer explicitly told us that she believed us to transcribe everything correctly and did not want to check the transcript.

[8]For coding we have used the *Atlas.ti* software.

[9]The code *Scala* was mentioned by only one developer as an example of her subproject

[10]For detailed analysis, please, refer to Appendix D.

Table 3: Codes used in the study

The final code book consists of 27 codes grouped into 7 code groups. Figure 2 shows the frequency of occurrences of the resulted codes.

Code group	Code	Description	Example
Dependencies	dependency	operations with dependencies	`We are using enough number of libraries.` (#14)
Language	C/C++	discussion specific to C/C++	`[...] with the C++ you have to include the libraries yourself.` (#10)
	Java	discussion specific to Java	`Well, this is a Java story` (#14)
	JavaScript	discussion specific to JavaScript	`Well, the JavaScript world is a mess.` (#7)
	Python	discussion specific to Python	`[...] but we are working with Python.` (#24)
Attitude	like	positive assessment	`If we can apply automation test, it would be good for us [...]` (#17)
	dislike	negative assessment	`[...] but we are also afraid of its effect on the other flows.` (#17)
	recommendation	suggestion of improvements	`[...] having a SonarQube plug-in - it would be great.` (#3)
Context	functionality	project functionalities or features	`[...] and we integrated that functionality in our project.` (#8)
	requirements	policies or requirements	`We have a contract that we inform our clients once a month.` (#5)
	security	security related statement	`It's very complicated to figure out that your code has such a vulnerability.` (#3)
Issues	broken	something not working	`[...] to avoid all service to go down.` (#9)
	bugs	programming error description	`Well, bugs of course.` (#12)
	resources	human or time resources	`I cannot address every smallest issue [...]` (#2)
	licenses	rights to use software	`[...] it is difficult to control compatibility of licenses` (#2)
	fix availability	availability of a bug fix	`Simply we used another library [...]` (#23)
Operations	maintenance	changes that involve modifications of source code	`We suggest fixes to the contributors.` (#7)
	management	changes that involve modifications of project configuration	`Every couple of days I would upgrade all of the packages.` (#15)
	dependency selection	selection of new dependencies	`When we select them, we have a discussion.` (#5)
	direct deps	dependencies introduced directly	`[...] our direct dependency was Jenkins.` (#9)
	looking for info	check 3rd-party sources for info	`I still go to Github, read sources[...]` (#5)
	transitive deps	dependencies of dependencies	`If you have a transitive dependency [...]` (#3)
Process	automated	solutions that automate software engineering tasks	`Thanks to various tools, bots, which just sit in your repository[...]` (#7)
	code tool	tool for analysis of quality and security of code	`It produces a report on the [] server.` (#3)
	workflow	company practices discussion	`[ironically] Yes, we have a weekly reminder [...]` (#5)
	dependency tool	tool for analysis of quality and security of dependencies	`We are using the [] scanner and it is the only one[...]` (#20)
	manual	solving a task without application of any automation tools	`We do not use any tools to check security.` (#16)

Table 4: Developers' attitudes: likes vs dislikes

The table shows the co-occurrence of codes like and dislike with other codes of issues, process, operations, and context code groups. For example, codes dislike and management have 86 co-occurrences, which means the depelopers often expressed negative attitudes towards dependency management. We mark (underline and bold) the number of co-occurrences exceeding 18 (sum of the mean and one standard deviation of code co-occurrences). The full co-occurrence table is available in the Apendix E.

	issues			process		operations			context	
	broken	bugs	resources	automated	workflow	management	looking for info	trans deps	functionality	security
dislike	**21**	**29**	**23**	14	16	**86**	15	12	**23**	**36**
like	6	**31**	3	4	6	**44**	9	1	8	**44**

developers shared similar attitudes regarding dependency management: most frequent codes are *management*, *security*, and *bugs*. Most concerns of C/C++ developers were on the co-occurrences of these codes with code *dislike*. Hence, below we present our findings without distinguishing by programming language.

5.1 RQ1: rationale for selection

To understand the developers' rationale for the selection of new dependencies for their projects and whether security aspects affect their choices, we have studied the developers' answers simultaneously marked by the codes *management* and *looking for info*.

Observation 1: *Security is considered for selection if it is enforced by company policy: some companies have a pull of homegrown or preapproved FOSS libraries, so developers are encouraged or even sometimes restricted to use them in their projects.*

Three of the interviewed developers (#5, #10, #28) directly communicated, that they considered security while selecting software dependencies. However, for them, this was forced by the policy of their companies: #10 has to use only the dependencies approved by an internal dependency assessment tool that as well ensures that the libraries are secure, while #5 checks security history of a library in case it is planned to be included in the core of their project.

The developers #10, #12, and #13 mentioned that their companies have a pull of preapproved FOSS and homegrown libraries. These libraries and their dependencies are checked for the presence of security issues and functionality bugs, and therefore, have a higher priority to be used in comparison to their FOSS alternatives.

```
We are trying to use them [preapproved libraries] actively. This
is highly appreciated and sometimes is even forced due to code
reuse [...] (#13)
```

Discussion. Derr et al. [11] reported that Android developers consider security among the least important criteria for selecting new dependencies, while several recent papers underline the impact of company policies on developers' decisions to consider security. The early dependency studies [7, 9] reported that company policies might encourage developers to consider security, but these policies are not always followed in practice. More recent studies (e.g., [3, 42]) observed the stronger impact of the company policies on the developers' decisions regarding considerations of security, however, these studies provide limited insights on the impact of company policies on the dependency selection process. Hence, our observation clarifies if company security policies also impact the developers' decisions regarding software dependencies.

Observation 2: *Developers mostly rely on community support of a library: if a vulnerability or a bug is discovered in a well-supported library, the fix appears quickly, it is easy to adopt, and it does not break the dependent library.*

The other interviewed developers instead relied on community support of considered libraries, as the community can be leveraged for troubleshooting both functionality and security issues: in case of a vulnerability is discovered in a well-supported library it will be quickly fixed, and the security fix is usually easy to adopt as it does not break the dependent project.

```
I maybe do a quick google and select the thing that works
best for a lot of people[...] if there're bugs, it's going to
be easier to work [them] out just by using, let's say, the
canonical package [and asking the community for support.] (#15)
```

Discussion.The previous studies of Android developers [11, 21, 42] reported that developers lack community support and a central package manager. We fill the gap by studying the ecosystem of developers working in the context of established central package managers (Maven, NPM, PyPI). Previous papers [7, 16] suggested that developers prefer libraries that are popular and well-supported to include into their projects as they are more reliable from the functionality perspective. Hence, we add to these observations by providing evidence that developers perceive community support to be a 'guarantee' for a library to be secure.

Observation 3: *Dependency selection is often assigned to a skilled developer or a software architect.*

The task of selection of new dependencies is often assigned to software architects (#10, #14, and #17) or to "someone who has experience" (#12):

```
The most difficult case is to decide which dependencies should
be used, how dependencies should be used, or in general design
the structure of a project. That is the reason why the task of
designing the structure of software is assigned to the software
architect: because they have a lot of experience. They have to
check the project before developers actually work. (#17)
```

Discussion.Pano et al. [32] reported that a combination of developers, customers, team, and team leader often leads to the selection of a development technology/framework. In this perspective, we clarify that the dependency selection (i.e., specific libraries to be used within a preselected framework) in big SMEs and LEs are often assigned to a skilled developer or a software architect.

Observation 4: *For dependency selection, developers mainly focus on functionality support of a library, rather than its security.*

Interviewed developers mentioned functionality aspects twice more often rather than security while selecting new software dependencies for their projects: 27 co-occurrences of *functionality* and *selection of new dependencies* codes in the interviews of 12 developers in comparison to 11 co-occurences of *security* and *selection of new dependencies* codes in the interviews of 7 developers.

Observation 5: *For dependency selection, developers refer to high-level information that demonstrates community support of a library, rather than low-level details of a library source code.*

When we asked questions about the selection of new dependencies, developers often reported that they rely on third-party resources to get additional information about new dependencies: 22 out of 25 developers (everybody, except #2, #3, and #20) shared additional sources of information that they refer to before including a new dependency into their projects.

14 out of 25 developers (#1, #4, #5, #6, #8, #9, #13, #15, #17, #19, #22, #23, #24, and #25) named *Github.com* as the primary information source since Github allows them to both understand whether there exists a strong community behind a particular library, and, if necessary, have additional details about library code. As for high level information, the interviewees may refer to the number of stars (#1, #4, #6, #9, #22, and #23), project contributors (#4, #15, and #23), and library users (#4, #5, #9, #15, #22, and #25). Additionally, developers were interested in the code style of a project (#5, #8, #9, and #22), commit frequency (#4, #5, #8, #9, #17, #23, and #24), as well as the number of issues resolved (#5, #9, and #17), still open (#17), and how quickly an open issue is fixed (#4, #5, and #17).

```
If a library has thousands of issues that are open, then you
need to be careful. [Once] integrated, you may experience the
same problems. (#9)
```

Additional sources of information mentioned by developers were Google (#4, #6, #15, #16, and #25), dependency repositories, like Maven Central (#4, #12, #17, and #19), Node.js, or PyPI (#9, #24), and StackOverflow (#22). The developers referred to these sources to find the most popular dependencies that solve particular tasks.

According to the most referred sources and types of information, the interviewed developers pay little attention to security aspects (as unpalatable as this observation might be) and instead look for excellent community support of the library: if a library features quick security fixes, but fixes of its functionality issues linger, such a library will likely not be selected.

Discussion.We complement the existing observations (e.g., [7, 9, 11]) on the information sources developers refer to while selecting new dependencies and provide specific insights into why particular information source is referred to from the security perspective.

Observation 6: *To avoid legal issues, enterprise developers check software licenses while selecting new project dependencies.*

Besides security and functionality, developers of every type of organization we covered specified that one needs to be careful while selecting software dependencies, since there also exist license issues of using them as part of a proprietary software project: FOSS (#13, #24), SME (#14, #24), LE (#3, #10, #13), UG (#2)

```
[...] if you sell some software, and inside your software you
have a restricted license, like GPLv3, you could have a lot of
legal issues, because the owner of the library may discover that
and you may have a lot of legal problems. (#3)
```

Discussion.Current qualitative studies of FOSS ecosystem [16, 42] provided limited insights on the impact of legal concerns on developers' decisions for selecting software dependencies. For example, Linden et al. [42] reported that individual developers recruited for a laboratory task have a limited understanding of (and little patience to understand) legal issues behind the usage of third-party software. In contrast, we observe that developers belonging to each organization type we covered (FOSS, SME, LE, UG) have reported that they consider licenses of dependencies before including them into their projects.

5.2 RQ2: motivations for (not) updating

To answer this research question, we have looked into the particularity of the dependency management process. More specifically, we have considered the conversation fragments labeled by the codes of the *attitude* code group (Table 4).

Observation 7: *In general, developers have mixed perceptions about dependency management process, while few developers have strongly negative and strongly positive attitudes.*

Developers expressed different perceptions of the dependency management process: they have mentioned negative aspects (86 co-occurrences of codes *dislike* and *management* in the interviews of 22 developers), as well as expressed positive attitudes towards dependency management (44 co-occurrences of codes *like* and *management* in the interviews of 18 developers). Six developers mentioned only problematic aspects, two reported only positive attitudes, and 16 developers expressed mixed perceptions of the dependency management process (i.e., their interviews contained co-occurrences of both *dislike* and *like* codes with *management* code).

```
Yeah, it was really hard to switch from AngularJS [...] to
Angular2. But they did a great job, so every other update, like
Angular2, 4, 5, 6, [...] the switch is really smooth. You don't
have to do lots of crazy things. (#21)
```

Discussion.While several previous studies [6, 7, 9, 25, 31] reported developers to have mainly negative attitudes towards dependency management process, we observe that enterprise developers have mixed perceptions while several developers expressed only positive attitudes.

Observation 8: *If developers update dependencies of their projects, they pay attention to vulnerabilities.*

The most important and discussed issue for the developers in our sample were *bugs* (84 occurrences in the interviews of 22 developers). When the developers spoke about bugs, often they discussed vulnerabilities (61 co-occurrences of codes *bugs* and *security*).

Observation 9: *Developers perceive security-related fixes as easy to adopt, as for widely-used and well-supported libraries such fixes appear fast and do not break the dependent projects.*

Developers do not have negative concerns about fixing vulnerabilities in dependencies since they either use only well-known stable libraries that rarely introduce vulnerabilities and quickly fix them (#5, #6, #16, and #17); or their projects are not security critical, i.e., used only for internal purposes, hence even if a vulnerability appears in their dependencies, they will not be exploited (#3, #4, #9, and #24). Also, the adoption of fixed dependency versions typically does not break the dependent projects (#1, #4, #11, and #14).

Discussion.Developers are reported to be less proactive about dependencies [6] as they felt it is difficult to manage the dependencies or the lack of support in providing updates from vendors [25]. However, we observe a generally positive attitude of developers to security fixes in software dependencies, since fixed versions of well-supported dependencies appear fast, and their adoption does not break the dependent projects.

Observation 10: *Developers tend to avoid updating dependencies of their projects since they lack resources to cope with the breaking changes (possibly hidden in transitive dependencies) introduced by new dependency versions.*

Many interviewees reported that, generally, they try to avoid updates of dependencies in their projects. 14 developers (#1, #4, #7, #8, #9, #10, #11, #12, #14, #15, #16, #17, #18, and #23) said that they do not have enough resources to perform proper dependency management, while 11 developers (#4, #7, #8, #9, #12, #13, #14, #16, #17, #19, and #23) mentioned that they avoid updating dependencies of their projects since updates might introduce breaking changes:

```
Our project is huge. We tried once, and 1000 tests became down.
To fix it[...] We just do not have time for that. Hence everything
became frozen. (#8)
```

Eight developers (#1, #2, #3, #7, #13, #14, #17, and #23) said that they experienced problems with dependency management due to a high number of transitive dependencies that are difficult to control.

Discussion.The previous studies of developers perceptions on dependencies [7, 11, 31] reported breaking changes to be the main factor that stops developers from updating dependencies of their projects. Our finding complements these studies and also suggests project stability to be the highest priority for developers. I.e., they are not updating dependencies for security reasons unless developers are confident that this update is free from breaking changes (or developers have enough time and resources to thoroughly test their projects). Also, our observation shows that the lack of control over the high number of transitive dependencies causes a significant strain in managing and updating dependencies. It can be one of the main reasons for not updating dependencies, in addition to technical debts, performance reasons, or bug fixes [11].

Observation 11: *Company policy significantly affects developers' decisions about updating software dependencies by splitting the field in two: adopt every new version or ignore all updates.*

Developers #7 and #19 said that the established practice and company mindset might force developers to follow different dependency management strategies. For example, developers #7, #15, #19, and #21 said they keep dependencies of their projects fresh and perform "small" updates every time the new dependency version appears. The update process seems "quite smooth" for them.

```
I faced dependency updates in [company name]. And there such
task appeared maybe twice a month. (#19)
```

On the other hand, developers #7, #8, #12, #15, and #19 mentioned that they try to avoid updates of software dependencies as much as possible due to the risk-averse mindset and lack of proper motivation for updating software dependencies (as new does not mean bug-free): although they did not express any problematic aspect in it, developers #8, #12, and #19 reported that they do not update dependencies in their projects since their company policies suggest keeping versions of dependencies unchanged.

Table 5: Dependency operations vs Process

The table shows the number of co-occurrences of codes of dependency operations and process code groups. For example, codes workflow and management have 45 co-occurrences, which means the depelopers often discussed how they integrated dependency management into their workflow. We mark (underline and bold) the number of co-occurrences exceeding 18 (mean + one standard deviation). The full co-occurrence table is in Apendix E.

	Dependency operations				
	mainte-nance	manage-ment	direct deps	look for info	trans deps
automated	1	**18**	0	7	2
code tool	0	5	0	2	0
workflow	1	**45**	2	13	2
dependency tool	3	**26**	0	9	1
manual	7	13	1	7	1

```
I faced at this job, that most people do not understand why it's
needed to update libraries, why we need to refactor code. If
everything works, do not touch it, do you need that most? And if
I start to fix everything by myself, I would just become crazy
to convince everyone. Actually, I had a not so good experience,
when I tried to increase the code quality a bit. And people
started to complain: why did you touch that? (#8)
```

Discussion.Developers are reported to be not always encouraged to update a library as it works as intended [11, 31], the update contains only minor improvements [3], or there are not enough development resources available [25]. In contrast with the previous studies, we observe that several enterprise developers have an opposite approach: they update dependencies of their projects as soon as the new version of a dependency appears. Our interviewees suggest the company policy to be the key factor for such a change in the dependency management practice.

5.3 RQ3: automation of dependency management

To answer RQ3, we have looked at the developers' answers that were marked by one of the codes from the *process* code group.

Observation 12: *Dependency analysis tools (if used) are applied for identification of arising issues within dependencies, so developers can assess the findings to decide whether to adopt a new dependency version. The dependency update itself is performed manually.*

On dependency management (see Table 5), developers often referred to the contextual information established within their companies: the codes *management* and *workflow* co-occurred 45 times in the interviews of 16 developers (#3, #5, #7, #9, #10, #12–14, #18–25).

Developers #3, #5, #7, and #10 reported that they apply dependency analysis tools in their day-by-day work to identify possible problems within dependencies of their projects (26 co-occurrences of codes *dependency tool* and *management*). They have the automatic dependency scanning tools integrated with their workflow, and they have to check the generated issues manually. If they decide to update a dependency, developers #3, #7, #9, #17, and #18 prefer to manually configure the project to use the new version and then manually test the project to ensure that it functions correctly.

```
You add a request and say: "I would like to have this library".
There is a process for that and someone will investigate this and
will run the [Dependency Tool], and you will get an automatic
report. And so the [library] will be cleared or not. (#10)
```

Discussion.Several studies [6, 9, 20] reported that developers do not update dependencies due to the lack of awareness about security issues affecting their projects. There are some reasons for this: the absence of proper security knowledge, lack of plans for security assessment, and appropriate tools [3]. But the studies did not investigate the roles of dependency analysis tools. We observe that enterprise developers are aware of existence of dependency analysis tools, and (if applicable) use them as the supporting source of information for planning manual dependency management tasks. However, they do not rely on the tools for sensitive operations, like automatically updating dependencies of their projects. The last observation aligns and complements the finding reported by Mirhosseini and Parnin [31].

Observation 13: *Developers recommend introducing high-level metrics that show that a library is safe to use (security badge), mature, and does not bring too many transitive dependencies.*

To facilitate the selection of new dependencies, developer #6 recommends having badges in Github (or one's dependency management system) that show whether usage of a particular dependency is safe. Besides checking for vulnerabilities in a specific version of a dependency, the developers #16 and #25 suggest defining whether the dependency is mature (See 5.1), while the developer #13 would like to see if the new dependency increases the technology stack or introduces new transitive dependencies.

Discussion.Mirhosseini and Parnin [31] reported that developers would like to see some supporting and explanatory arguments for an automated bug fixing suggestions to be accepted. Also, the authors found that developers prefer to have passive notifications (e.g., badges) about changes in dependencies. We observe similar developers' desire regarding the information about software dependencies – developers would like to have a high-level metric (i.e., an argument) showing if a library should be adopted.

Observation 14: *Developers think that dependency analysis tools generate many irrelevant or low priority alerts.*

The developers #9, #15, and #22 tried dependency analysis tools, but decided not to introduce them into their work process due to a significant number of unrelated alerts:

```
I had one [dep. analysis tool] and it tended to spamming, and I
turned it off. For example, it reported minor vulnerabilities,
so I was kind of annoyed by them. (#15)
```

Observation 15: *Several developers tried dependency analysis tools but decided to rely on the information about vulnerability fixes and functionality improvements distributed via social channels.*

Many developers (#1, #2, #3, #7, #9, #10, #11, #17, and #18) perform manual analysis of their dependencies. Five developers (#1, #2, #4, #18, and #24) said that they use social channels, like Twitter or dependency mailing lists, to receive information about discovered issues and new versions of their project dependencies.

Discussion.Observation 14 suggests that dependency analysis tools share the well-known weakness of static analysis tools (e.g., [21, 43]) used to find security issues in the own code of software projects: false-positive and low-level alerts annoy developers. Hence, they abandon the tools and prefer to seek social support, although the information it sometimes provides is too much to digest [7].

Observation 16: *Developers recommend dependency analysis tools to report only relevant alerts, work offline, be easily integrated into company workflow, and report both recent and early safe versions of vulnerable dependencies.*

Regarding the dependency analysis tools, developer #18 suggests the tools to report only the findings that really affect the analyzed project (reduce the number of false positives if possible). Developer #9 suggests that security tools should work offline, since otherwise, they may disclose some sensitive information about the analyzed projects. Developer #19 suggests that the tools for analyzing software dependencies should be easy to integrate with development pipelines, while developer #22 would like to have reported both early and recent safe versions of the identified vulnerable dependencies, so there will be a possibility to consider several versions to update to.

Discussion.Johnson et al. [21] reported that developers want code analysis tools that provide faster feedback in an efficient way that does not disrupt their workflows and allow them to ignore specific defects about their own code. We observe similar requirements for dependency analysis tools.

Observation 17: *Developers consider dependency analysis tools to be similar to static (or dynamic) analysis tools and recommend these tools to be integrated so that they could be applied simultaneously.*

Developers #2, #3, #8, #9, and #13 considered dependency analysis tools to be similar to code analysis tools (i.e., static or dynamic analysis tools). Hence, they could be applied to the same stage of the software development process.

```
Security assessment of your dependencies should stay near the
security assessment of your code, because it's part of the
security assessment of your code. (#3)
```

Developers #3 and #13 even gave us the recommendation to augment the reports from a code analysis tool (for example, SonarQube) with alerts generated by a dependency analysis tool:

```
Maybe it's possible to plug the results of dependency analysis
to SonarQube? So we would be able to use it later on in our
continuous integration and do continuous code analysis. It would
be cool to have this. (#13)
```

Discussion.We do not find other related works that discuss the integration of dependency analysis tools into the development workflow. Since enterprise developers often perceive the dependency analysis tools to be integration-wise similar to static analysis tools, the tools could be applied at the same time during the development process, e.g., build or compile time [21, 43], integrated in an IDE [15], or into a code review [43].

5.4 RQ4: Mitigating unfixed vulnerabilities

To answer RQ4, we had examined the developers' answers, where they described the mitigations of the cases when no newer version of a vulnerable dependency had a fix for a vulnerability (the interview fragments tagged with codes *fix availability* and *dislike*).

Observation 18: *When discovered a vulnerable dependency that does not have a fix, developers first try to understand whether this vulnerability affects their project. If its fix requires significant effort, then developers will likely decide to stay with the vulnerability.*

Although the interviewees #1, #3, #7, #11, and #23 said that they always were able to find a fixed version of a vulnerable dependency, the others considered such a situation as probable and problematic.

When discovered a case of a vulnerable dependency that does not have a fix, the developers #3, #5, #7, and #14 firstly assess whether this vulnerability impacts their projects since maybe they do not use the affected functionality. In case a vulnerable dependency does not impact their project, developers may just decide to leave the project unchanged (for example, #16). Even if a project depends on the affected functionality, but the vulnerability fix requires significant development effort, developers #1, #2, #12, and #15 prefer to stay with the vulnerability.

```
If I have to rewrite all the application and the cost is huge,
then maybe we will stay with the vulnerability. (#2)
```

Discussion.Several developers' studies (e.g., [11, 25, 29]) reported the evidence that developers try to avoid changing dependencies unless they understand the absolute necessity of this operation. Hence, this finding aligns with these studies, as the first step for developers is to understand if the vulnerability impacts their project [16, 25] and estimate the effort required to mitigate the vulnerability.

Observation 19: *If vulnerability affects their project, some developers may decide to temporarily disable the affected functionality and wait for an "official" patch.*

Developers #2 and #12 said that they could just roll back to a previous unaffected version of a vulnerable dependency.

If developers decide to address the security issue without a fix in the dependencies of their projects, then they are likely to check the solutions suggested by other library users or maintainers (for example, #4 and #15). In case they discover that the maintainers are working on the problem and are going to release a fix soon, the developers #4, #17, and #20 temporarily disable the project functionality that is exposed to the vulnerability:

```
We had to change the configuration of [image] library to totally
disallow that particular attack vector. (#20)
```

Discussion.Bogart et el. [7] observed that developers act less proactive about dealing with (functionality) bugs in their dependencies: sometimes developers decide to do nothing with their own project but wait for the fixed version of the dependency [29]. We observe that in case of vulnerability disclosures, developers are more proactive: they check the impact of the vulnerability on their projects and provide immediate solutions by disabling affected functionality of their projects.

Observation 20: *Skilled developers fix vulnerabilities in their dependencies and contribute to the dependency projects.*

The skilled developers #4, #7, #8, #13, and #15 may decide to fix security vulnerability by themselves. While developers #4, #8, #13, and #15 said that they prefer to create an internal fork of a vulnerable dependency and maintain it until an "official" vulnerability fix is released, the developers #7 and #13 reported that developers of their companies actually fix discovered security issues and contribute to third-party projects by opening pull requests in their FOSS repositories:

```
If this vulnerability seriously impacts our work and if this is
an open source product, then we just fix it. For example, if it
is just in Github, we just fix it, creating Pull Request. And
we ask contributors or maintainers to merge this Pull Request
into the master branch. And we are pushing them to release a
new version faster. (#13)
```

Discussion.Several recent papers [7, 16, 29] reported that, depending on the expertise, developers might decide to contribute to the dependency projects to fix some functionality issues. The interviewed developers reported that they distinguish functionality and security fixes, and think that security fixes require higher expertise. We also observe that skilled developers also contribute to the dependency projects by fixing their security issues.

Observation 21: *As the last resort, developers may substitute vulnerable dependency of their project with another library that provides similar functionality.*

If the fix of a software library is too complicated and the library is not well supported, then developers may decide just to stop using it and switch to another library (for example, #3 and #23).

```
Simply, we used another library, which more or less did the
same thing. [...] And that, of course, caused us to rewrite
some piece of software. At least we solved this memory leak
problem in [Library Name]. (#23)
```

Discussion.Several studies suggested that developers might decide to update or downgrade a vulnerable dependency to fix bugs [29] or even contribute to the dependency project [7, 29]. In this respect, we contribute to this body of knowledge by showing that enterprise developers sometimes decide to substitute a vulnerable library with another one that provides similar functionality.

6 IMPLICATIONS

Implication 1: *Considering security while selecting new dependencies might be expensive for individual and SME developers.*

While looking for libraries to include in their projects, developers have to seek and combine information from various sources, like discussions present in developer forums or code metrics extracted from software repositories. This process requires time and expertise, and therefore, is preferably performed by experienced developers or software architects (O3). In large enterprises developers sometimes have a pull of preapproved FOSS and homegrown libraries (O1). The developers of such companies could use these libraries without further investigations as they are guaranteed to be reliable. However, smaller software development companies or individual developers (e.g., freelancers) do not have such a reliable source. While hiring an experienced software architect might be quite expensive for them.

Research ideas: To help SME and individual developers consider security while selecting new dependencies for their projects, the complex information could be combined, e.g., in the form of badges or *meta-metrics* accessible and understandable by developers (O13). Such meta-metrics are expected to facilitate the following tasks:

- demonstrate that library is well-supported and its issues are resolved quickly (O2 and O9);
- suggest that the library is not affected by known security vulnerabilities (O13);
- demonstrate that the library is mature, so it does not bring many undiscovered bugs and security vulnerabilities (O13);
- show licenses for the library itself and its transitive dependencies (O6).

Implication 2: *Both LE and SME developers are more likely to adopt a security fix not bundled with functionality improvements.*

Since security fixes (at least for well-supported libraries) typically do not introduce breaking changes (O9) and they should not be bundled together with functionality improvements: if they are mixed together, developers would have to spend efforts to cope with breaking changes introduced by functionality improvements. Under the constraints of limited resources (O10), developers will most likely ignore such an update and stay with the vulnerability. Instead, if a security fix is well-indicated, well-documented, and it does not require significant development effort, then it has more chances to be adopted.

Research ideas: To help library creators always keep functionality, updates, and security fixes separate, researchers could design an automatic approach capable of distinguishing functionality and security changes. Then developers might decide to release two independent library versions. For library users, researchers could develop an automatic classifier capable of identifying whether a specific library version includes changes related to functionality or security. So, developers could immediately adopt security fixes (as they do not introduce breaking changes) and schedule adoption of functionality updates.

Implication 3: *LE developers tend to adopt automated dependency analysis tools, while SME and individual developers are not encouraged to use them.*

LE developers have policies to consider the security of their dependencies, and therefore they are forced to use the dependency analysis tools (O11). In contrast, SME and individual developers lack procedures for considering security in their projects. Moreover, they are more concerned about developing new functionality, and therefore, they often prefer to ignore "annoying" alerts of dependency analysis tools (O14) and fix security issues in dependencies of their projects only if these issues are severe and widely-known.

Research ideas: To facilitate the adoption of dependency analysis tools by SME and individual developers, tool creators could design their tools to satisfy the following developers' requirements:

- report only vulnerable dependencies that actually affect the analyzed project (O14, O16, and O18);
- identification of the part of the analyzed project affected by the vulnerability (O18);
- suggest both new and early safe versions of the dependency, so developers could select the best mitigation strategy: to adopt a new version or roll-back to an earlier one (O16);
- suggest if a fixed version introduces breaking changes (O16).

Implication 4: *LE developers are more proactive in fixing vulnerabilities within dependencies of their projects, while SME and individual developers tend to behave passively.*

LE developers sometimes contribute to the projects they depend on by fixing vulnerabilities and creating pull requests (O20). However, SME and individual developers might not have enough time, skills, and development resources to support dependency projects. Therefore, they tend to rely on community support of their dependencies and would prefer to either stay with vulnerability (O18) or temporarily disable some functionality of their projects[11] (O19).

Research ideas: If there is no fixed version available for a vulnerable dependency, the developers perform manual analysis to devise the countermeasures for the discovered issue. Since this action is critical, on top of the requirements presented in Implication 3, there is a need to have support from the dependency analysis tools on the following aspects (especially for LE developers):

[11]Some LE developers also prefer to temporarily disable the feature within their projects, when such an option is allowed by their company policy.

Table 6: Summary of Results

RQ	Analysis summary	Implications
RQ1	When selecting a new dependency, developers pay attention to security only if it is required *and* enforced by the policy of their company. Otherwise, they mainly rely on popularity and community support of libraries (e.g., number of stars, forks, project contributors).	High level metrics, that allow developers to understand that the library is well-supported, mature, and not affected by security vulnerabilities, could facilitate library selection.
RQ2	As generally developers lack resources to cope with possible breaking changes, they prefer to avoid updating dependencies for any reason. Security vulnerabilities motivate developers for updating only if they are severe, widely known, and adoption of the fixed dependency version does not require significant efforts.	To be adopted, library versions that fix vulnerabilities should (i) be well-indicated, (ii) not introduce breaking changes, and (iii) not contain functionality improvements (as they are likely to break dependent projects).
RQ3	Developers perform sensitive dependency management tasks (e.g., updates) manually. Current dependency analysis tools (if used) only facilitate identification of vulnerabilities in the project dependencies. Developers complain that dependency tools produce many false-positive and low-priority alerts.	Dependency analysis tools should (i) generate alerts only relevant to the fragments of the libraries used by dependent projects; (ii) show the affected components of the dependent projects; (iii) suggest if there exists a fixed version and if its adoption introduces breaking changes.
RQ4	The interviewed developers suggested the following actions when a vulnerability is discovered in a dependency, but no newer version fixes it: (i) assess whether this vulnerability impacts them since maybe they may not use that particular functionality; (ii) leave the vulnerability and wait for the fix or a community workaround, (iii) adapt own project, i.e., disable vulnerable functionality or rollback to a previously safe version of the library; (iv) maintain own fork (possibly fixing and making a pull request).	Dependency tools should (i) primarily determine *which part of the dependent project is actually affected* by the vulnerability in a dependency; (ii) facilitate access to the dependency source code, so developers could assess and possibly fix the vulnerability by themselves; (iii) suggest an alternative library that provides similar functionality.

- accessing the dependency source code, so the developers could directly check it and possibly fix the issue (O20);
- finding an alternative library with similar functionalities and estimating the cost of switching to this library (O21).

7 THREATS TO VALIDITY

We recruited developers for our study without using any material rewards, only based on their interest in the topic. In our study, we aimed to receive information from industrial specialists who have good solid positions. Hence, we could not think of any better reward for them than a possibility to improve the development practice by sharing their experience and to tell us their opinions on their problems. Moreover, very often, the developers were motivated by the fact that we had already had a prototype of a tool that we could use to produce some dependency analysis reports for their projects. We believe that this strategy allowed us to receive the especially valuable feedback from the field specialists, who have the appropriate level of knowledge of the topic.

We applied the snowballing approach to increase the number of developers we could reach. This may potentially attract developers from the same development communities who share common views. To mitigate this bias, we selected developers to come from different companies and different countries. The finally interviewed developers have various backgrounds and company positions. Hence, we believe that this threat is minimal.

Our observations are based on facts as perceived by the interviewees. They might not necessarily reflect the reality, hence, more qualitative and quantitative studies are needed to validate the presented implications. Unfortunately, field observational studies are hard to get. For example, de Souza and Redmiles [10] report two case studies for a total of 23 interviews. In spite of de Souza being embedded in the company for several weeks, only 'some of the team members agreed to be shadowed for a few days'. Similarly [42] did a survey of 274 developers but, to observe developers, had to recruit 44 of them and assigned them laboratory designed tasks.

Currently, we mostly asked developers about dependency management practices within their companies, which may hide some issues related to the development of FOSS projects. However, nowadays, developers often have to consume, contribute to, or, at least, follow the trends in FOSS community: several interviewees, although being industrial employees, also told us about their contributions to FOSS projects. Hence, we believe that the analysis and the implications presented in this study provide valuable insights for developers working in both FOSS and enterprise contexts.

We present our interpretations of the developers statements. To minimize confirmation bias, the two researchers individually extracted their observations and implications from the interviews, while the third researcher performed an additional validation of the analysis results. Additionally, we performed a validation of the results with the developers, by sharing the one-page summary of the findings with the interviewees. Hence, we believe our results correspond to the actual reported dependency management practices.

8 CONCLUSIONS AND FUTURE WORKS

This paper reports the results of a qualitative study of developers' perception of software dependencies and the relative importance of security and functionality issues. We run 25 semi-structured interviews, each around 30', with developers from both large and small-medium enterprises located in nine different countries.

All interviews were transcribed and coded, along with the principles of applied thematic analysis. We summarise the implications of our qualitative findings as follows:

- Optimal selection of (FOSS) libraries could be facilitated with high-level metrics that allow developers to understand that a library is well-supported, mature, and not affected by security vulnerabilities.
- Dependency updates break dependent projects, so *if ain't broken, don't touch it* rules the world. To be adopted, security fixes should be well-indicated, not introduce breaking changes, and not require significant efforts.
- To maximize utility, dependency analysis tools should generate alerts only relevant to the fragments of the libraries used by dependent projects and suggest possible mitigation strategies along with estimation whether they introduce breaking changes.

- Given the strong forces against updates, general security alerts are likely to end as unheeded ‘cries for wolf’. Actionable tools should determine which part of the dependent project is *actually* affected by the vulnerability in a dependency and suggest alternative libraries that provide similar functionality along with the estimation of the cost of switching to that library.

Several nuances are still unaddressed by our study, starting from broadening our studies to more countries to correlating results with different types of industries (e.g., financial companies, critical infrastructures, or social media - as we cover all of them but with too few samples each). The most challenging future work for us and the community at large is to develop the dependencies and security analysis tools required by our developers.

ACKNOWLEDGMENTS

We thank the interviewed developers for their time and wiliness to share their perceptions on the dependency management process. We are grateful to Dr. Achim D. Brucker, Dr. Leysan Nurgalieva, and Prof. Paolo Tonella for their valuable comments on the early versions of this paper and to the anonymous reviewers for their insightful and actionable suggestions.

This research has been partly funded by the EU under the H2020 Programs H2020-EU.2.1.1-CyberSec4Europe (Grant No. 830929) and the NeCS: European Network for Cyber Security (Grant No. 675320).

REFERENCES

[1] B. Adams. 2018. Developers of popular software projects are overloaded by the requests from academic researchers. (2018). Suggested during a personal communication with the authors at ESEM’2018.

[2] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. 2016. Tracing known security vulnerabilities in software repositories–A Semantic Web enabled modeling approach. *Sci. Comp. Program.* 121 (2016), 153–175.

[3] Hala Assal and Sonia Chiasson. 2019. ’Think secure from the beginning’ A Survey with Software Developers. In *Proc. of CHI’19*. 1–13.

[4] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. 2012. Cohesive and isolated development with branches. In *Proc. of ICFASE’12*. Springer, 316–331.

[5] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. 2010. Codebook: discovering and exploiting relationships in software repositories. In *Proc. of ICSE’10*, Vol. 1. IEEE, 125–134.

[6] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proc. of ASEW’15*. IEEE, 86–89.

[7] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. of FSE’16*. ACM, 109–120.

[8] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *Proc. of SANER’15*. IEEE, 516–519.

[9] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *Proc. of ICSE’15 (ICSE ’15)*. IEEE Press, Piscataway, NJ, USA, 109–118. http://dl.acm.org/citation.cfm?id=2819009.2819027

[10] Cleidson de Souza and David Redmiles. 2008. An empirical study of software developers’ management of dependencies and changes. In *Proc. of ICSE’08*. IEEE, 241–250.

[11] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on Android. In *Proc. of CCS’17*. ACM, 2187–2200.

[12] Leo A Goodman. 1961. Snowball sampling. *AOMS* (1961), 148–170.

[13] Robert Wayne Gregory, Mark Keil, Jan Muntermann, and Magnus Mähring. 2015. Paradoxes and the nature of ambidexterity in IT transformation programs. *ISR* 26, 1 (2015), 57–80.

[14] Greg Guest, Kathleen M MacQueen, and Emily E Namey. 2011. *Applied thematic analysis*. Sage.

[15] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. 2018. On adopting linters to deal with performance concerns in android apps. In *Proc. of ASE’18*, Vol. 11. ACM Press.

[16] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. 2013. Categorizing developer information needs in software ecosystems. In *Proc. of WEA’13*. ACM, 1–5.

[17] Mohanad Halaweh. 2012. Using grounded theory as a method for system requirements analysis. *JISTEM* 9, 1 (2012), 23–38.

[18] Regina Hebig and Jesper Derehag. 2017. The changing balance of technology and process: A case study on a combined setting of model-driven development and classical C coding. *Journal of Software: Evolution and Process* 29, 11 (2017), e1863.

[19] JI Hejderup. 2015. In dependencies we trust: How vulnerable are dependencies in software modules? (2015).

[20] J. Huang, N. Borges, S. Bugiel, and M. Backes. 2019. Up-To-Crash: Evaluating Third-Party Library Updatability on Android. In *Proc. of EuroS&P’19*. 15–30.

[21] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don’t software developers use static analysis tools to find bugs?. In *Proc. of ICSE’13*. IEEE Press, 672–681.

[22] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proc. of MSR’17*. IEEE, 102–112.

[23] Andrew J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proc. of ICSE’07*. IEEE Press, 344–353.

[24] Paul R Kroeger. 2005. *Analyzing grammar: An introduction*. Cambridge University Press.

[25] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2017. Do developers update their library dependencies? *Emp. Soft. Eng. Journ.* (11 May 2017). https://doi.org/10.1007/s10664-017-9521-5

[26] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proc. of NDSS’17*.

[27] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *Proc. of ESEM’13*. IEEE, 383–392.

[28] SS Jeremy Long. 2015. Owasp dependency check.

[29] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs? A case study on the GitHub scientific Python ecosystem. In *Proc. of ICSE'17*. IEEE, 381–392.
[30] Mark Mason. 2010. Sample size and saturation in PhD studies using qualitative interviews. In *Forum qualitative Sozialforschung/Forum: qualitative social research*, Vol. 11.
[31] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proc. of ASE'17*. IEEE Press, 84–94.
[32] Amantia Pano, Daniel Graziotin, and Pekka Abrahamsson. 2018. Factors and actors leading to the adoption of a JavaScript framework. *Empirical Software Engineering* (2018), 1–32.
[33] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proc. of ESEM'18*.
[34] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. 2012. Information needs for integration decisions in the release process of large-scale parallel development. In *Proc. of CSCW'12*. ACM, 1371–1380.
[35] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *Proc. of ICSME'15*. IEEE, 411–420.
[36] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software. In *Proc. of ICSME'18*.
[37] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage.
[38] Khaironi Y Sharif, Michael English, Nour Ali, Chris Exton, JJ Collins, and Jim Buckley. 2015. An empirically-based characterization and quantification of information seeking through mailing lists during open source developers' software evolution. *Information and Software Technology* 57 (2015), 77–94.
[39] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451.
[40] Anselm Strauss and Juliet Corbin. 1990. *Basics of qualitative research*. Sage.
[41] Hataichanok Unphon and Yvonne Dittrich. 2010. Software architecture awareness in long-term software product evolution. *Journal of Systems and Software* 83, 11 (2010), 2211–2226.
[42] Dirk van der Linden, Mark Levine, and John Towse. 2020. Schrödinger's Security: Opening the Box on App Developers' Security Rationale. In *Proc. of ICSE'20*. IEEE.
[43] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *Proc. of SANER'18*. IEEE, 38–49.
[44] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proc. of MSR'16*. IEEE, 351–361.
[45] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *Proc. of ICSME'12*. IEEE, 306–315.
[46] Robert K Yin. 2015. *Qualitative research from start to finish*. Guilford Publications.

A APPENDIX – STATE OF THE ART COMPARISON

Table 7 presents a summary of qualitative studies of developers' attitudes and practices. We identify the following sources of information used by the selected studies: interviews, surveys, mailing lists, observations of developers' work process. For each study we report the number of participants and whether the study provided the insights for the code groups identified in Section 2.

B APPENDIX – FAILED ATTEMPT OF THE INTERVIEWEE SELECTION

Interviewee selection – failed attempt. First, to invite developers for the interviews, we decided to reach developers of the most popular open-source Java projects. For this purpose, we created a search on Github by the keyword "Java" and selected the top 20 most starred projects (Table 8). Then we used our tool for the dependency study (See §4.1 for details) to generate dependency analysis reports for those projects. We sent these reports to the main contributors (or owners) of the selected projects and asked them to provide their feedback on the reports as well as to dedicate some time for an interview. Unfortunately, this activity did not provide us with the sufficient number of interviewees, because there was only one response.

In agreement with B.Adams [1], the most likely reason for the fact, that developers of the most popular Github projects ignored us, is that they may be overloaded by the various research studies. I.e., the developer selection approach we followed is very tempting for researchers. Hence, developers of popular research projects may receive many emails with different requests for participation in various scientific studies. So, they treat such kind of requests as spam and ignore it. In our case the request for the study also contained an attachment. And in the light of constantly increasing threat of ransomware, such kind of emails looked very suspicious. So, we had to select a different strategy for hiring interviewees.

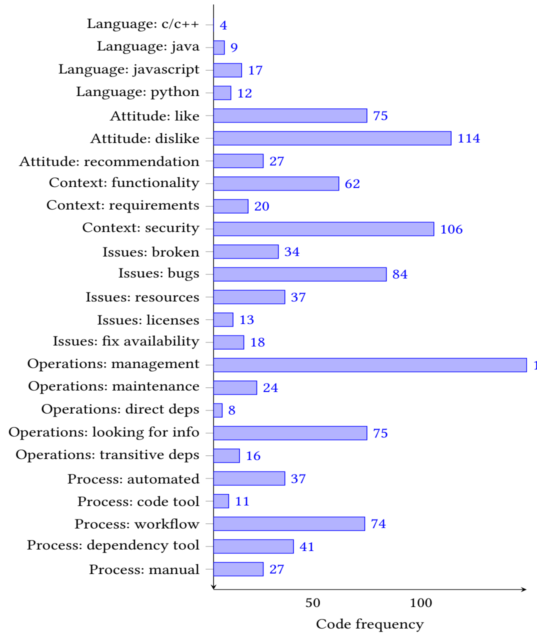

Figure 2: Code frequency by code groups

C APPENDIX – CODES DISTRIBUTION

Figure 2 shows the frequency distribution (number of occurrences) of the codes attributed to the fragment of interviews. Developers are worried about the possible issues (including security bugs) that dependencies may introduce into their projects: *dislike* (114 occurrences), *security* (106 occurrences), and *bugs* (84 occurrences) are within the topmost mentioned codes. At the same time, the relatively low number of occurrences of codes such as *direct deps* (8 occurrences) and *transitive deps* (16 occurrences) in an interview about dependencies suggests that developers may not consider all details of the dependency management process to be really problematic (*like* had 75 occurrences). After all, this is the whole advantage of using dependencies as black boxes:

```
If there's something we really know to be broken, we fix it.
Otherwise, it's kind of left to itself. (#1)
```

Table 7: Summary of qualitative studies about developers' attitudes and practice

The table presents a summary of qualitative studies of developers' attitudes and practices by interviews (I), surveys (S), mailing lists (M), observations of developers' work process (O). For each study we report the number of participants and whether the study provided the insights for the code groups used in this study .

	Dependency studies								Tool/Technique validation studies						Information needs and decision making											
	[11]	[16]	[10]	[29]	[6]	[7]	[9]	[25]	[31]	[21]	[15]	[43]	[18]	[4]	[32]	[38]	[41]	[39]	[27]	[34]	[23]	[5]	[45]	[3]	[42]	Ours
Type	S	S	O+I	S	I	I	I	M	S	I	I	S+I	I	I	I	M	I	O	I	I	O	I	O+I	S	S	I
#Partic.	203	14	6,8+15	116	7	28	5	16	62	20	14	42+11	6	6	18	ND	15	25	15	7	17	15	6+6	123	274	25
Deps	✓	✓	✓	✓	✓	✓	✓	✓	✓						✓		✓									✓
Lang	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓								✓			✓
Attitude	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Context Function.	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Context Security	✓						✓	✓															✓	✓	✓	✓
Issues				✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓								✓	✓	✓	✓
Operation	✓	✓	✓	✓	✓	✓			✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓
Process	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 8: Top 20 most starred projects from Github

url	version	project name
https://github.com/square/retrofit	parent-2.4.0	retrofit
https://github.com/square/okhttp	parent-3.11.0	okhttp
https://github.com/google/guava	released-all-futures	Google Guava
https://github.com/apache/incubator-dubbo	dubbo-2.6.4	Apache Dubbo
https://github.com/zxing/zxing	BS-4.7.8	ZXing
https://github.com/kohsuke/jenkins	1.386	Jenkins
https://github.com/raver119/deeplearning4j	latest_release	Deeplearning4j
https://github.com/eclipse/vert.x	3.5.4	Vert.x
https://github.com/prestodb/presto	0.212	Presto
https://github.com/perwendel/spark	2.7.2	Spark
https://github.com/brettwooldridge/HikariCP	HikariCP-3.2.0	HikariCP
https://github.com/junit-team/junit4	JUnit 4.12	Junit4
https://github.com/xetorthio/jedis	jedis-2.9.0	Jedis
https://github.com/code4craft/webmagic	WebMagic-0.7.3	WebMagic
https://github.com/google/auto	auto-value-1.6.3rc1	Google Auto
https://github.com/dropwizard/dropwizard	v2.0.0-rc0	Dropwizard
https://github.com/emeroad/pinpoint	1.6.2	Pinpoint
https://github.com/redisson/redisson	redisson-3.8.2	Redisson
https://github.com/codecentric/spring-boot-admin	2.0.3	Spring Boot Admin
https://github.com/swagger-api/swagger-core	v2.0.5	Swagger Core library

The preliminary analysis also suggests that developers prefer to use dependencies as they are (i.e., adopt new ones or update them) rather than to go deeper into details and change the source code: all interviewed developers discussed *management* (149 occurrences), while only 15 out of 25 developers touched the *maintenance* topic (24 occurrences).

Then we analyze the codes that are mentioned together. For this purpose, we have extracted the co-occurrence table of the interview codes [24]: each column and row of the table corresponds to an interview code, while each cell contains the number of code co-occurrences. To identify the cells with a significantly high number of co-occurrences, we have calculated the mean and standard deviation for the code co-occurrences ($\mu = 8.27$, $\sigma = 11.62$) in the table and underlined the values in cells, where the number exceeds μ by at least the value of σ (i.e. 19.89). To reduce the noise, we will not report the columns where cell values do not exceed μ.

D APPENDIX – PER LANGUAGE ANALYSIS

Figure 3 shows the distribution of codes by languages.

E APPENDIX – COMPLETE CO-OCCURRENCE TABLE

Figure 4 shows the complete co-occurence table for the Section 5.

C

Code	maintenance & C	management & C	direct_deps & C	looking_for_info & C	trans_deps & C
automated	0	2	0	2	0
code tool	0	0	0	0	0
workflow	0	8	0	7	0
dependency tool	0	3	0	4	0
manual	0	1	0	0	0

Code	automated & C	broken & C	bugs & C	maintenance & C	management & C	direct_deps & C	functionality & C	workflow & C	resources & C	security & C	looking_for_info & C	trans_deps & C
dislike	1	1	4	3	9	0	6	2	2	6	4	0
like	0	0	1	0	0	0	0	0	0	1	0	0

JAVA

	maintenance & Java	management & Java	direct_deps & Java	looking_for_info & Java	trans_deps & Java
automated	1	4	0	1	1
code tool	0	3	0	1	0
workflow	0	14	1	1	1
dependency tool	1	4	0	1	1
manual	2	4	0	2	0

	automated & Java	broken & Java	bugs & Java	maintenance & Java	management & Java	direct_deps & Java	functionality & Java	workflow & Java	resources & Java	security & Java	looking_for_info & Java	trans_deps & Java
dislike	3	12	5	2	29	1	6	7	6	8	3	5
like	2	1	9	2	18	0	6	4	0	16	4	1

JAVASCRIPT

	maintenance & JS	management & JS	direct_deps & JS	looking_for_info & JS	trans_deps & JS
automated	1	9	0	3	2
code tool	0	2	0	1	0
workflow	0	16	0	3	0
dependency tool	2	15	0	4	1
manual	1	6	0	2	0

	automated & JS	broken & JS	bugs & JS	maintenance & JS	management & JS	direct_deps & JS	functionality & JS	workflow & JS	resources & JS	security & JS	looking_for_info & JS	trans_deps & JS
dislike	6	3	4	1	25	1	2	3	2	10	5	5
like	0	3	10	2	18	0	0	4	2	17	1	1

Python

	maintenance & Python	management & Python	direct_deps & Python	looking_for_info & Python	trans_deps & Python
automated	0	9	0	2	0
code tool	0	1	0	0	0
workflow	0	12	1	2	1
dependency tool	0	10	0	1	0
manual	0	6	1	3	1

	automated & Python	broken &	bugs &	maintenance & Python	management & Python	direct_deps & Python	functionality & Python	workflow & Python	resources & Python	security & Python	looking_for_info & Python	trans_deps & Python
dislike	4	6	11	0	33	0	8	6	11	12	4	2
like	1	4	16	0	22	0	1	0	0	16	5	0

Dependency operations for languages **Developer attitutes for languages**

Figure 3: Distribution of codes by languages. Available online at https://gofile.io/?c=u54rXh

Code	automated	broken	bugs	c/c++	code tool	dependency tool	direct deps	dislike	fix availability	functionality	java	javascript	licenses	like	looking for info	maintenance	management	manual	python	recommendation	requirements	resources	security	transitive deps	workflow
automated	0	1	9	6	7	18	0	14	0	2	8	12	1	4	7	1	18	3	14	7	2	2	18	2	13
broken	1	0	6	1	0	0	0	21	2	7	16	8	1	6	0	4	28	3	13	1	0	6	4	2	4
bugs	9	6	0	10	2	10	2	29	15	18	16	20	1	31	19	9	45	8	33	4	4	12	57	3	17
c/c++	6	1	10	0	3	6	0	17	4	10	0	1	8	2	17	4	20	1	0	5	2	2	14	2	12
code tool	7	0	2	3	0	3	0	1	0	1	5	2	0	0	2	0	5	0	2	2	0	0	9	0	7
dependency tool	18	0	10	6	3	0	0	11	0	2	6	17	2	7	9	3	26	2	14	6	1	4	18	1	9
direct deps	0	0	2	0	0	0	0	2	0	0	4	2	0	0	1	0	7	1	3	0	0	1	3	4	2
dislike	14	21	29	17	1	11	2	0	7	23	33	31	7	3	15	9	86	9	44	5	3	23	36	12	16
fix availability	0	2	15	4	0	0	0	7	0	5	7	2	0	9	2	9	3	2	5	0	0	4	11	0	2
functionality	2	7	18	10	1	2	0	23	5	0	25	9	5	8	16	4	33	2	16	8	2	2	8	3	10
java	8	16	16	0	5	6	4	33	7	25	0	4	3	28	26	6	47	6	0	9	7	9	26	7	21
javascript	12	8	20	1	2	17	2	31	2	9	4	0	1	27	22	4	46	7	10	8	4	7	29	5	26
licenses	1	1	1	8	0	2	0	7	0	5	3	1	0	1	3	0	6	0	2	1	0	1	1	1	2
like	4	6	31	2	0	7	0	3	9	8	28	27	1	0	9	8	44	2	34	1	2	3	44	1	6
looking for info	7	0	19	17	2	9	1	15	2	16	26	22	3	9	0	4	50	7	17	5	2	0	22	1	13
maintenance	1	4	9	4	0	3	0	9	9	4	6	4	0	8	4	0	7	7	13	2	0	7	6	1	1
management	18	28	45	20	5	26	7	86	3	33	47	46	6	44	50	7	0	13	49	18	14	28	56	14	45
manual	3	3	8	1	0	2	1	9	2	2	6	7	0	2	7	7	13	0	17	0	0	3	8	1	8
python	14	13	33	0	2	14	3	44	5	16	0	10	2	34	17	13	49	17	0	6	5	15	33	3	18
recommendation	7	1	4	5	2	6	0	5	0	8	9	8	1	1	5	2	18	0	6	0	1	2	7	4	6
requirements	2	0	4	2	0	1	0	3	0	2	7	4	0	2	2	0	14	0	5	1	0	0	9	0	8
resources	2	6	12	2	0	4	1	23	4	2	9	7	1	3	0	7	28	3	15	2	0	0	17	0	3
security	18	4	57	14	9	18	3	36	11	8	26	29	1	44	22	6	56	8	33	7	9	17	0	5	33
transitive deps	2	2	3	2	0	1	4	12	0	3	7	5	1	1	1	1	14	1	3	4	0	0	5	0	2
workflow	13	4	17	12	7	9	2	16	2	10	21	26	2	6	13	1	45	8	18	6	8	3	33	2	0

Figure 4: Full Co-occurence table. Available online at https://gofile.io/?c=iBuWJW

F APPENDIX – INTERVIEW TRANSCRIPT EXAMPLE

a. **How do you deal with software dependencies in your projects?**
Usually, when I deal with software dependencies, I rely on some tools, for example, Maven, Gradle for Java. Or pip for Python. Some dependencies, that you introduce, which can be, let's say, not compliance with other libraries about some reason maybe. Maybe one dependency has a dependency on another library, but different versions, which can be tricky. I think so. Let's say, I have also an issue, while external dependency for.. I think, it was Json, no xml parser in Java. And this library created memory leaks in context at the time. And that was very bag experience with external libraries. Because I needed to take some memory snapshot to understand what was the leak. And I understood, that the leak was caused by an external library. So not by the code, that we were writing.

b. **And how did you cope with that bug? What did you do?**
Simply we used another library, which more or less did the same thing. Now I am thinking, that it wasn't an xml parser, it was something to, an utility to expose REST services in Java. And we used another tool. We basically changed library. And that, of course, caused us to rewrite some piece of software. At least we solved this memory leak problem in Tomcat.

c. **Ok, I see. So you basically substituted this library?**
Exactly. A solution maybe to make an issue to a library and wait for a fix, but at that time we decided to change the library. Also because we changed the library and we wrote better some piece of software. We took the moment to do it.

d. **Yes, sure. Fair enough. There was an alternative. That's good. I see. I wanted to understand better on what you're telling me. Can you tell me a bit about your background? I nderstand, that you're a Java developer. How much experience do you have?**
I was working basically five years in .Net, and then three years in Java plus university projects if you can count them. They were also in Java. And then also one year and a half in Python. There was a JavaScript framework for web development. Which in the case was NodeJS. But, let's say, in that case we used npm, so node package manager to manage the libraries in JavaScript.

e. **And currently you are working in a company, right?**
Yes, I recently changed again. Yesterday I started at the new company. SO here the recent project, that I am involved in, again Java. I came back to Java.

f. **And in the previous job?**
In the previous job I was working with Python and I was working with Django. So, the back- end. And at least vJS at some time.

g. **I see. And what was the scope of the company? I mean, what kind of projects were you working on?**
This is a big corporate, wanted to implement a certain solution. They produce plastic for automatic surface. And it has several clients all over the world. China, South America, Europe, Nothern America. And they wanted to build a system to make the Industry 4.0. Basically, so it is still a big project. The development of a web application to be used by all the employees of the company, which allows to read data from sensors installed on the machines through several protocols, for example, PROTOCOL1. To read SOME data. And then to also read data from other sources. For example, some ERP system. And then a lot of features, that are still under development to digitalize the production sector and standardize the way they use the system all over the world. It's very big activity to summarize it in several words. I hope, I was enough clear.

h. **Yeah, yeah. I understand something. In broad perspective. Ok, and how old was the project, that you were working on?**
This project, I mean, in the last company was.. I mean, we started it from scratch. Then I had other experiences before, working on some, let's say, established software. And so in that case we had a lot of dependencies. And introduced those dependencies, I think, in our pipeline or development environment.

i. **Ok, I see. in both projects, that you are talking about. They were Python projects, right?**
Python project was in the last company, where I was working for one and a half years. Before I was working in another company, and there I was developing in Java. There I was working on both old piece of software without any kind of, unfortunately, a dependency management. At the beginning. Then we introduced Maven to fix the jar we were facing, let's say. It works on my machine, then I had a chance to use Gradle. But for those projects I used Gradle as a very beginning level. So the depependency were controlled by those tools.

j. **Ok, and so when you implemented this switch to Maven, when you introduced Maven to this project.. How did you select the dependencies, that you want to include?**
Well it was complicated. Let's say, we had lib folder with a lot of dependencies inside. The guy, who implemented the software didn't.. They had no idea on how to organise them better. And then basically we

started from scratch to compile the software and added dependencies one by one. It was a long process. And then we faced also some problems of these versions of required libraries. And then we also had some problems at runtime. You know, in pom files there are explicit dependencies to other jar files. So sometimes it also depends on the quality of the dependency. Dependency pom file. Sometimes they say compile exception, because the dependency tree was not satisfied. And it was a painful process. But it was necessary to introduce a new development to the field. And it was a mess with it. And then without this technology you cannot think about any improvements. You know, like continuous deployment or use some automatic tools online to build and deploy applications.

k. **Yeah, of course. I see. And so. Ok, when you.. Did you also face sometime the situation, when you have to select the new dependency for your project? Like to introduce some new functionality?**
Yes, I did. This especially happens with the project I started by myself. Of course, I needed some extra features. And I used dependencies for that. And it actually very convenient to take external dependencies.

l. **Ok, but how do you actually select them? So, what do you consider during that selection?**
Well, I usually, check if they are reliable by looking at Github sites and numbers of.. I am talking only about open source dependencies. I check if they are reliable, I look at the Github stars, number of commits, contributors. I see if the project is active. And so I understand, that this is a dependency I can introduce, because a lot of people is using it. And it is still maintainable. It is stable. Then in other cases you can buy some external library. But it was really bad situation. And I was basically just asking for feedback of other people, who were using the same library. You can read reviews online. If there is a customer support also. Of course, when you have paid solution, you have also another kind of support.

m. **I see. In case you had to select this paid library.. Why did you have to go for it? Why did you decide to select the private, commercial library and not the open source?**
Especially, when you work for a Microsoft environment for .Net applications, there are no a lot of free alternatives for enterprise applications. For example, windows form applications for, let's say rapid development. You want to speed up the development, so you basically buy these libraries, because there was no open source solutions in that case. For example, to develop user interfaces in windows form, I used the vectors. And the library, which was build on top of windows form, that also they have libraries for also web development and other software stuff. And there, of course, you pay quite a lot. But then you have a lot of features already out of the box, functionalities, which is actually 99% what customer wants. So in that case, let's say. Since you pay, these libraries are very reliable. Usually there is no problem of integration or bugs, or malicious software.

n. **I see. But still, there may be some new bugs discovered. Or some new vulnerabilities discovered?**
Vulnerabilities - no. But bugs - yes. Sometimes you find some bugs also in this case. And there is a support, that you submit these issues. And if you pay, they answer you after it. Then it depends if you are... Because we are working for a customer and he had, he bought the golden support. And with golden support they also release patches for you. And they introduce the patch in the next version, so for other people. So this is the other type of contact.

o. **Ok, I mean, that's interesting. This patch, I assume, that you need to just basically apply for your dependency, without update to a new version. Right?**
Yes.

p. **And doesn't break the build of the project?**
Usually it's a minor release, so it's not breaking anything. Then, of course, when there are major releases. The risk you break something, which also can be high.

q. **I see. Did you actually face the updates of the library? Did you have to updates the libraries in your projects?**
Yes. Let's say. It depends on the policy of the company. Some says, do not touch anything unless it stops working. But then you, maybe if you need to update some library and you move some version 1.0 to 3.5, because, let's say, you decide or your CTO in three years do not update anything. But then it becomes a mess. Because skipping a lot of versions can really break a lot of stuff. In another company they instead have this policy to update the main libraries very often. And that of course any time when the library was updated, you should do a lot of tests.

r. **Ok, but do you see any correlation with the policy of the company and programming language that they use?**
Well, it is. Usually I saw Microsoft environment, let's say, in corporate environments of big companies, big corporates, that can afford paying licenses. And another companies that use Java and other open source technologies are much smaller. But about the policies about updating or not updating dependencies - no. I cannot say anything, that there is a correlation.

s. **In this case do companies prefer to update libraries or they prefer just to keep them?**

It depends on the company. How many developers you have, how money you have on the project. Because, of course, this updating process is also time consuming. Apparently time consuming, because it take resources from developing new features, solving new bugs. But then, let's say, in long term, I think, it saves a lot of time, because ok, one day you will need to update all libraries, I think. Because, you know, some bugs, some vulnerabilities.. I don't know. It can be a lot of stuff. It really depends on how long the company exists, how experience the managers are.

t. **I see. This is really interesting what actually drives the companies to update software dependencies. For example, in your experience, how often did you update dependencies? And when did you decide to update them?**
Let's say, in.. When I was working for .Net applications. At least we updated the main dependencies once per year. Because these libraries received major updates once per year. So we basically updated these. To always be align with basic features. But it was a more structured company and we had people to do it. And also, let's say, there was no issue not to do it. Then, of course, if you work for a smaller company. Or maybe if you are alone managing the project, then you basically update when you need to. Or several factors can drive this. For example, you need to.. There is a bug or there are new features, that the new version offers. Or you need to change the version of Java, because the version you are using needs updating. Then you understand, that your all dependencies need to be updated also. Let's say, I think in a well-structured software company, they should plan the updates frequently. If you want to maintain your products. If you want to include new features. But then, of course, it depends on different cases.

u. **And what about the security side? So, did you ever faced the security of the dependencies?**
About security I do not have a lot of examples. I know, that, for example, for Java and also other environments, they. At some point they stop to release patches, security patches. It depends also how much you.. Is your business to make application secure.

v. **So in your experience, do the companies looked somehow on the security sides of their dependencies? Did they check their dependencies on the presence of security vulnerabilities?**
No, in this case, I can say. Let's say, in this case I saw the dependencies are used around, where, let's say, mostly trustworthy, because they are very used dependencies. Because they are also, dependencies are coming from enterprises. So you kind of trust them. It's not just a random dll or random jar you find. So, let's say, that we are using, we search for reliable sources. Also, you sometimes want to look at the code to understand if it actually introduces some security vulnerabilities.

w. **Ok, I see, it's fair enough. If the publisher development company is trustable, if it is big enough. Like Microsoft, then you kind of trust them And they do not ship bad code.**
Yes yes yes. Also if you use some dependencies from open source, from Github with a lot of contributors, a lot of stars, you can trust. I don't know if anyone is going to check anything in such libraries. A lot of people use them and they can just do whatever. But there is someone in the world, some nerd ones, that take look at the code, at every line of code and do this work for me.

x. **Yes, sure. That's the idea behind the dependencies. So kind of outsource some part of your work to somebody else.**
Aha, yeah.

y. **I have just last question basically. You already mentioned, that there was a time, when you had to switch to another library, because there was a problem, there was a bug. So basically there was no fix. Can you comment about this situation? So, if you, also from the perspective of different companies where you worked. So, if you face this situation, when there is no fix, what would be your reaction if there is no new version with the fix of the bug or security vulnerability?**
In this case if it was an open source library, I don't know, we could complain to the maintainer of the library. If there is a license somewhere, if there is a line saying: I'm not responsible for any bugs. That policy of the company was, that when they understood, that the memory leak was caused by this library in our, let's say, in our software configuration. We checked where the library was used and we understood if it was very painful to change it or not. And we understood, that it wasn't that painful and we took also the chance to rewrite some old class in a better way. Then we didn't experience, the memory leak, let's say, any more.