UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
IECS International Doctoral School

Towards Understanding and Securing the OSS Supply Chain

Ly Vu Duc

Advisor:

Prof. Fabio Massacci

University of Trento, Italy and Vrje University, The Netherlands

External Reviewers:

Prof. Achim D. Brucker

University of Exter, The United Kingdom

Prof. Mariano Ceccato

University of Verona, Italy

Eindhoven, The Netherlands
February 28, 2022

Acknowledgements

> *"We're born alone, we live alone, we die alone. Only through our love and friendship can we create the illusion for a moment that we're not alone."*
>
> – Orson Welles

Above all, my wholehearted thanks are owed to the support of the European Union through the H2020 NECS and AssureMOSS projects. With the fund of these projects, I was able to perform various research, training, and social activities during my Ph.D.

My special gratitude goes to my advisor, Prof. Fabio Massacci. Under your advice, I have grown from a newbie to a researcher, and more importantly, a better person. For your patience, I owe you immeasurably.

I am indebted to Henrik Plate and Dr. Antonino Sabetta, who gave me valuable advice during my internship. I had gained pretty much industrial experience while working with you at SAP Security Research.

I want to give my thanks to the reviewers of this thesis: Prof. Achim D. Brucker and Prof. Mariano Ceccato, for your valuable comments and feedback. This thesis could not be done without you.

I want to thank Prof. Bruno Crispo for supporting me when I was an early-stage researcher in the NECS project from 2017 to 2020 and being on my qualifying exam as well as thesis defense committee.

I want to thank my colleague Dr. Ivan Pashchenko for your constant support and mentor. A special mention to my former colleague Dr. Chan-Nam Ngo for your help in both life and research during my Ph.D. journey. I also thank my labmate Giorgio Di Tizio and the other members of the Security Group at the University of Trento.

I want to thank Pierantonia Sterlini, Irma Carolina Fung, and Tasiana Egorova for supporting my activities in Trento. Also, thanks to the IECS Doctoral School officers for supporting me in my Ph.D. program.

Last but not least, my special gratitude goes to my family for your love and support. To my father and grandfather, I know that you have always been with me whenever you are. Thanks to my brother Tuong ('Andy') Vu for the two beautiful years. Last but not least, special thanks go to my financée Van ('Ami') Le for always being with me in my Ph.D. journey.

Abstract

Free and Open-Source Software (FOSS) has become an integral part of the software supply chain in the past decade. Various entities (automated tools and humans) are involved at different stages of the software supply chain. Some actions that occur in the chain may result in vulnerabilities or malicious code injected in a published artifact distributed in a package repository. At the end of the software supply chain, developers or end-users may consume the resulting artifacts altered in transit, including benign and malicious injection.

This dissertation starts from the first link in the software supply chain, 'developers'. Since many developers do not update their vulnerable software libraries, thus exposing the user of their code to security risks. To understand how they choose, manage and update the libraries, packages, and other Open-Source Software (OSS) that become the building blocks of companies' completed products consumed by end-users, twenty-five semi-structured interviews were conducted with developers of both large and small-medium enterprises in nine countries. All interviews were transcribed, coded, and analyzed according to applied thematic analysis.

Although there are many observations about developers' attitudes on selecting dependencies for their projects, additional quantitative work is needed to validate whether behavior matches or whether there is a gap. Therefore, we provide an extensive empirical analysis of twelve quality and popularity factors that should explain the corresponding popularity (adoption) of PyPI packages was conducted using our tool called PY2SRC*.*

At the end of the software supply chain, software libraries (or packages) are usually downloaded directly from the package registries via package dependency management systems under the comfortable assumption that no *discrepancies are introduced in the last mile between the source code and their respective packages. However, such discrepancies might be introduced by manual or automated build tools (e.g., metadata, Python bytecode files) or for evil purposes (malicious code injects). To identify differences between the published Python packages in PyPI and the source code stored on Github, we developed a new approach called* LastPyMile . *Our approach has been shown to be promising to integrate within the current package dependency management systems or company workflow for vetting packages at a minimal cost.*

With the ever-increasing numbers of software bugs and security vulnerabilities, the burden of secure software supply chain management on developers and project owners increases. Although automated program repair approaches promise to reduce the burden of bug-fixing tasks by suggesting likely correct patches for software bugs, little is known about the practical aspects of using APR tools, such as how long one should wait for a tool to generate a bug fix. To provide a realistic evaluation of five state-of-the-art APR tools, 221 bugs from 44 open-source Java projects were run within a reasonable developers' time and effort.

Keywords— Automated Program Repair, Mining Source Code Repositories, Software Supply Chain Attacks, Software Security, Software Supply Chain Security, Software Vulnerabilities, Dependency Management, FOSS Ecosystem, Empirical Study, Qualitative Study, Quantitative Study.

Contents

1 Introduction **1**

1.1 Problems and Research Questions 2

1.2 Contributions 4

1.3 Thesis Structure 7

1.4 Conventions Used in This Thesis 9

1.5 List of Publications 10

Glossary **12**

Acronyms **14**

2 Background **15**

2.1 General concepts of the software supply chain 15

2.2 The Python Package Index (PyPI) ecosystem 17

2.3 Qualitative studies about developers' attitudes and practice 18

2.4 Dependency Management Issues and Mitigations 20

2.4.1 Technologies/tools automation in software development process 22

2.4.2 Information needs and decision making during software development . . . 22

2.5 Quantitative studies on software dependencies 23

2.5.1 Selection of software dependencies/libraries 23

2.5.2 Finding source code repository URLs that correspond to PyPI packages . 24

2.6 Discrepancies between Sources and Packages 25

2.6.1 The last mile from source to package 25

2.6.2 Software supply chain attacks on package repositories 27

2.7 Malware detection in package repositories 28

2.7.1 Detecting malicious PyPI packages 30

2.7.2 Reproducible builds as an ideal solution 32

2.8 Automatic repairing bugs in software dependencies 32

3 A Qualitative Study of Dependency Management and Its Security Implications 34

3.1 Goal and Research Questions 34

3.2 Methodology 36

3.2.1 Recruitment of participants 37

3.2.2 Interview process 39

3.2.3 Interview coding and analysis 40

3.2.4 Final Code Book 42

3.3 Findings 45

3.3.1 RQ1.1: Rationale for Dependency Selection 45

3.3.2 RQ1.2: Motivations for (not) updating dependencies 48

3.3.3 RQ1.3: Automation of Dependency Management 51

3.3.4 RQ1.4: Mitigating unfixed vulnerabilities 55

3.4 Implications and Research Ideas 57

3.5 Threats to Validity 59

3.6 Conclusions 60

4 PY2SRC: Automatic Identification of Source Code Repositories and Factors for Selecting New PyPI Packages 62

4.1 Goal and Research Questions 63

4.2 Methodology 64

4.2.1 Finding GitHub URL of a PyPI package 64

4.2.2 Evaluation of Quality and Popularity Factors 68

4.3 Information sources for finding GitHub URLs of PyPI packages 70

4.4 RQ2.1: Combining information sources 72

4.4.1 Combining information sources. 73

4.4.2 Scaling the evaluation to 325 packages 76

4.5 RQ2.2: Factors that explain package adoption 79

4.6 Threats to Validity 84

4.7 Conclusions 85

5 LASTPYMILE: Identifying the Discrepancy between Sources and Packages 86

5.1 Goal and Research Questions 87

5.2 R3.1 LASTPYMILE efficiency for code injection identification 89

5.3 Data collection 93

5.4 RQ3.2: Differences between source code and package repositories 96

5.5 RQ3.3: LASTPYMILE combined with other package scanners 98

5.6 Threats to Validity . . . 102
5.7 Conclusions . . . 103

6 Please hold on: more time = more patches? Automated program repair as anytime algorithms 105
6.1 Goal and Research Question . . . 105
6.2 Benchmarks for Automated Repair of Java Programs . . . 106
6.3 Automated Program Repair Techniques . . . 107
6.3.1 Generate and Validate APR techniques . . . 108
6.3.2 Sematics-based APR techniques . . . 109
6.3.3 Metaprogramming-based APR techniques . . . 110
6.4 Evaluation of APR Techniques . . . 110
6.5 Preliminary Experiments . . . 111
6.6 Findings . . . 113
6.7 Manual Validation of generated patches . . . 116
6.7.1 Reason of Failed Repair Attempts . . . 116
6.7.2 Success repair attempts . . . 116
6.8 Threats to Validity . . . 119
6.9 Conclusions . . . 119

7 What is Next? 120

8 Appendix 122
8.1 CRediT authorship contribution statement . . . 122
8.2 Ethical Issues . . . 122
8.2.1 Main Ethical Issues . . . 123
8.2.2 Main Ethical Mitigations . . . 126
8.2.3 Data Collection and Protection . . . 126
8.3 Appendix – Failed attempt of the interviewee selection . . . 127
8.4 Appendix – Codes Distribution . . . 128
8.5 Interview transcript example . . . 130
8.6 Appendix – Complete co-occurrence table . . . 137
8.7 Appendix – Per language analysis . . . 137
8.8 Appendix – Typosquatting and Combosquatting Attacks in PyPI . . . 137

Bibliography 141

List of Tables

1.1 Mapping of the Contributions to the Problems and Research questions 2

2.1 Search queries on Elsevier Scopus database on March 2019 19
2.3 Summary of qualitative studies about developers' attitudes and practice 20
2.4 Discrepancies in files of legitimate and malicious typosquatting packages 27
2.5 Existing tools for analyzing Python packages 31

3.1 Descriptive statistics of the number of interview participants in the selected papers 37
3.2 Interviewees in our study 38
3.4 Codes used in the study. The final code book consists of 27 codes grouped into 7 code groups. Figure 8.1 in Appendix 8.4 shows the frequency of occurrences of the resulted codes. 43
3.5 Developers' attitudes: likes vs dislikes 49
3.6 Dependency operations vs Process 52
3.7 Summary of Results 61

4.1 Information sources for identifying GitHub URLs of PyPI packages 65
4.3 Reliable attribution metrics 67
4.5 The factors developers rely upon when selecting new dependencies 69
4.7 Information sources reporting a URL for PyPI packages 70
4.8 Information sources agreement on found URLs 72
4.9 Agreement between the reliability scores of URLs returned by information sources and manually identified URLs 74
4.10 Performance of OFS and methods that combine information sources 75
4.11 PY2SRC sources and tools comparison in terms of Precision and Recall among the 325 packages 76
4.12 The correlation between the automated computed reliable attribution metrics 77
4.13 Descriptive statistics for the values of the factors 80
4.14 Quality factors correlations 81
4.15 Linear regression summary for number of *Dependent repositories* 82
4.16 Linear regression summary for number of *PyPI downloads* 82

5.1 Number of source code repositories found by locations 90

5.2 Running time comparison between LastPyMile and git log approaches 92
5.3 Number of unique phantom files and lines versus total 92
5.4 Descriptive statistics about phantom lines in the artifacts 93
5.5 Descriptive statistics of GitHub repositories for the selected packages 93
5.6 Number of processed packages and artifacts 95
5.7 Differences between package artifacts and their source code repositories 97
5.8 Top different phantom Python files in our sample. 97
5.9 Top ten API and Sensitive calls in modified Python files 98
5.10 LastPyMile on Malware Checks and Bandit alerts 100

6.1 Factors affecting Success/Unsucccess Patches 111
6.2 Repair Benchmarks used in this study 112
6.3 Manual Validation of the Generated Plausible Patches 116

8.1 Main Ethical Issues 123
8.2 Top 20 most starred projects from Github 128
8.3 Malicious PyPI packages in our sample. 140

List of Figures

1.1 The software supply chain, and our contributions, research questions 6

2.1 Current Development, Build, Publication and Security Review pipeline of PyPI packages 16

2.2 Roles and responsibilities in the PyPI ecosystem.[1] 17

2.3 GitHub tags and PyPI releases of SeleniumLibrary 26

2.4 Detecting suspicious squatting packages (Vu et al. [201]).[2] 29

3.1 Research design of our study 41

4.1 Methodology process flow 64

4.2 Distribution of the number of identified Github URLs of the packages 71

4.3 Manual and Automatic reliability score over the 45 selected packages 75

4.4 Distribution of Reliability Score for *Final URLs* 78

5.1 LastPyMile in the context of the overall security review pipeline 89

5.2 Number of files differing between source and package 94

5.3 Number of lines differing between source and package 95

5.4 Percentage of different kinds of changes in artifacts and files 96

6.1 Experiments of different APR tools on different benchmarks constrained by time budgets 111

6.2 Total generated patches by all APR tools 114

6.3 Generated patches by each individual APR tool 115

8.1 Code frequency by code groups 129

8.2 Full Co-occurence table. Available online on Zenodo at [137] 138

8.3 Developer operations for languages. Available online on Zenodo at [137] 139

8.4 Developer attitutes for languages. Available online on Zenodo at [137] 139

“To my family, my mentors, and friends. I couldn’t have done this without you. Thank you for all of your support along my journey.”

– Ly Vu Duc

1

Introduction

> *"In science, if you know what you are doing, you should not be doing it. In engineering, if you do not know what you are doing, you should not be doing it."*
>
> – Richard Hamming

The expression *software supply chain* refers to "anything that goes into or affects your code from development, through your CI/CD pipeline, until it gets deployed into production" [87]. In the past decade, Free and Open-Source Software (FOSS) has become an integral part of the software supply chain: as much as 99% of codebases contain open-source code [180], and 85% [174] to 97% [195] of enterprise codebases comes from open source.

Unfortunately, software supply chain compromises are common and impactful. An attacker can compromise any single step in the process to maliciously modify the software and harm any of this software's users [183]. According to the Symantec Internet Threat Security Report (ISTR), Software Supply Chain compromise is the fastest growing threat to internet users — which rose 438% from 2017 to 2019. High and low profile companies are affected including companies like Apple, Microsoft [27]. Protecting against attacks on the software supply chain presents a complicated challenge because, as mentioned above, the ecosystems in which software are made are incredibly varied, and a compromise of a simple node in the pipeline often produces a complete subversion of the delivered product.

To harden the software supply chain security, this dissertation investigates each step of the software supply chain, for example:

- How and why developers select a software dependency (or package)?
- How do developers manage and maintain the dependencies of their projects? How do developers react to vulnerabilities in their dependencies?

Table 1.1: Mapping of the Contributions to the Problems and Research questions

Contribution	**Problem**				**Research Question**									
	P1	P2	P3	P4	RQ1.1	RQ1.2	RQ1.3	RQ1.4	RQ2.1	RQ2.2	RQ3.1	RQ3.2	RQ3.3	RQ4.1
C1	✓	✓		✓	✓	✓	✓	✓						
C2		✓	✓						✓	✓				
C3			✓								✓	✓	✓	
C4				✓										✓

- Which factors developers rely on when selecting a package? Are those factors always available?
- Are there any differences between packages (consumed by downstream projects) and sources? If so, how much?
- To what extent automatic tools are able to support developers fixing a bug under time constraints?
- How do the automated program techniques perform in practice?

Developers are key factors of the supply chain, thus the thesis first qualitatively looks at the choices and the interplay of *functional and security concerns* on the developers' overall decision-making strategies for *selecting, managing, and updating software dependencies* in their projects. Then a quantitative validaton of the factors mentioned by developers is performed on the top most downloaded Python packages on the PyPI ecosystem. Discrepancies in distributed package artifacts compared to their respective source code repositories, which pose both operational risks and security risks in the software supply chain, the next chapter proposes a methodology to identifying these discrepances. Finally, we investigate practical use cases of automated program repair to generate fixes for software bugs in the final products of the supply chain.

The next sections specify the problems associated with different elements of the software supply chain, and thesis's contributions to addressing the problems. Table 1.1 summarizes the association of contributions and problems in this thesis.

1.1 Problems and Research Questions

This section describes the problems followed by our research questions formulated to be answered to address the problems. Table 1.1 shows the mapping of the problems/research questions and our contributions described in the next section.

A handful of studies (Table 2.3) report that developers do not update dependencies in their projects [29,30,75]. The developer's perception of software dependencies and the relative importance of security and functionality issues have been studied using the rigid format of surveys [46] and the anecdotal examples that complement quantitative studies on dependencies [91]. Although functionality and security appear to be essential factors that affect developers' decisions [14], those studies mainly focus on functionality aspects, and therefore, provide limited insights on the impact of security concerns on developers' reasoning. Several studies [46, 82] also show this tension between functionality and security, but the studies are about ecosystems that do not feature a central place for storing and managing app dependencies. Developers with a central dependency management system, like Maven, npm, or PyPI, might have a very different approach towards the dependencies of their applications.

Problem 1: The existing studies have not investigated the interplay between security and functionality or the use of a central dependency management system. Hence, state-of-the-art studies do not capture different trade-offs between objectives.

RQ1.1: *How do developers select dependencies to include into their projects, and where (if at all) does security play a role?*

RQ1.2: *Why do developers decide to update software dependencies, and how do security concerns affect their decisions?*

RQ1.3: *Which methods, techniques, or automated analysis tools (e.g., Github Security Alerts) do developers apply while managing (vulnerable) software dependencies?*

RQ1.4: *Which mitigations do developers apply for vulnerable dependencies with no fixed version available?*

Developers rely on various factors displayed on Github repository (e.g., number of *stars*, presence of a repository *badges*) [94, 133] to form impressions about a library. Developers are reported to rely on factors available on the source code repositories. Since the number of 'factors' that may play a role in the selection process of a library is substantial, we identify the following problem in 'factors' suggested in the qualitative studies (e.g., interviews or surveys):

Problem 2: Current qualitative studies lack validation that source code repositories are always easy to find or factors supporting developer's choices of a software dependency are available.

RQ2.1: *How can we combine the information on PyPI pages to accurately identify GitHub URLs corresponding to PyPI packages and validate them?*

RQ2.2: *Which of the suggested factors do explain the adoption of a library?*

Developers download pre-built artifacts from package repositories (such as npm for JavaScript, PyPI for Python, or RubyGems for Ruby) under *the comfortable assumption* *that no discrepancies are introduced in the last mile between the source code and their respective packages.* However, such differences might be introduced by manual or automated build tools (e.g., metadata, Python bytecode files) [68] or for evil purposes. In particular, the differences between distributed artifacts and source code repositories could pose both operational risks (e.g., making dependent projects unable to compile) and security risks (e.g., deploying malicious code during package installation) for downstream components in the software supply chain. We identify the corresponding problem as follows:

Problem 3: Current package malware scanners which focus on one file are feasible but not enough, and reviewing an entire package is unfeasible due to the high number of false positives.

RQ3.1: *Can we effectively and efficiently identify differences?*

RQ3.2: *How big are the 'normal' differences between source code and package repositories?*

RQ3.3: *Can* LastPyMile *be combined with package scanners while keeping the number of alerts manageable by a human?*

Automated program repair (APR) techniques can relieve programmers from the burden of manually fixing the ever-increasing number of programming mistakes or software vulnerabilities during the software development and maintenance phase. Developers would like to have a way to automatically fix bugs based on the qualitative study in Chapter 3. Unfortunately, the practical aspect of APR tools on developers' development has received little attention. We have identified the following problem:

Problem 4: Current evaluations of automatic program repair (APR) techniques focus on tools' effectiveness, while little is known about the practical aspects of using APR tools, such as how long one should wait for a tool to generate a bug fix.

RQ4.1: *By doubling the time budget of an APR tool, do we get twice more plausible patches?*

1.2 Contributions

The thesis contributes to understanding and addressing the problems in the software supply chain both qualitatively and quantitatively. Figure 1.1 depicts the connections between each component in the software supply chain and the contributions and research questions in this thesis.

C1: A sound qualitative study of the motivation of developers between the rigid format of surveys (e.g., Derr et al. [46]) and the anecdotal examples that complement quantitative studies on dependencies (e.g., Kula et al. [91]). We conducted 25 semi-structured interviews with the professional developers of both large and small-medium enterprises located in nine countries. All interviews were transcribed, coded, and analyzed according to applied thematic analysis.

The quantitative studies have been focused on the packages consumed by the end users in the software supply chains. We focus on Python packages in the PyPI package repository because Python is the most popular language based on the 2021 TIOBE index [182] and the common language used by the developers in our qualitative study 3.

C2: **A quantitative validation** of the most commonly cited factors and their influence on the choices of a software dependency. We have developed a tool called PY2SRC to automatically identify GitHub source code repositories corresponding to packages in the Python ecosystem PyPI and extract the popularity and quality factors of the source code from GitHub that should explain the corresponding popularity (adoption) of packages in PyPI.

C3: **A methodology** called LastPyMile identifies the differences between build artifacts of software packages and the respective source code repository. We show how it can be used to extend current package scanning practices for malware injection (which only covers less than 1% of the code of deployed packages). We are working to submit LastPyMile as a new check to PyPI [204].

A preliminary assessment of automatic program repair tools on supporting fixing bugs of the software used in the software supply chain. We focus on the time budgets needed by the tools because it is the essential factor affecting development activity and the integration of the tools into the software development pipeline.

C4: **An evaluation** of Automated Program Repair (APR) tools as any time algorithms (e.g., the more time they have, the more fixes they generate, so it makes sense to trade off longer time for better quality). Our preliminary experiment on five APR tools and 221 Java bugs shows that the amount of plausible patches, given exponentially greater time, only increases linearly or not at all.

Section 5 of this thesis was performed in close collaboration with SAP Security Research to devise a methodology to detect code injections in software supply chain attacks. However, this thesis represents the research carried out by the author and does not necessarily represent the official position or research interests of SAP. The concrete contributions of the author of this thesis to the published works are declared in Appendix 8.1.

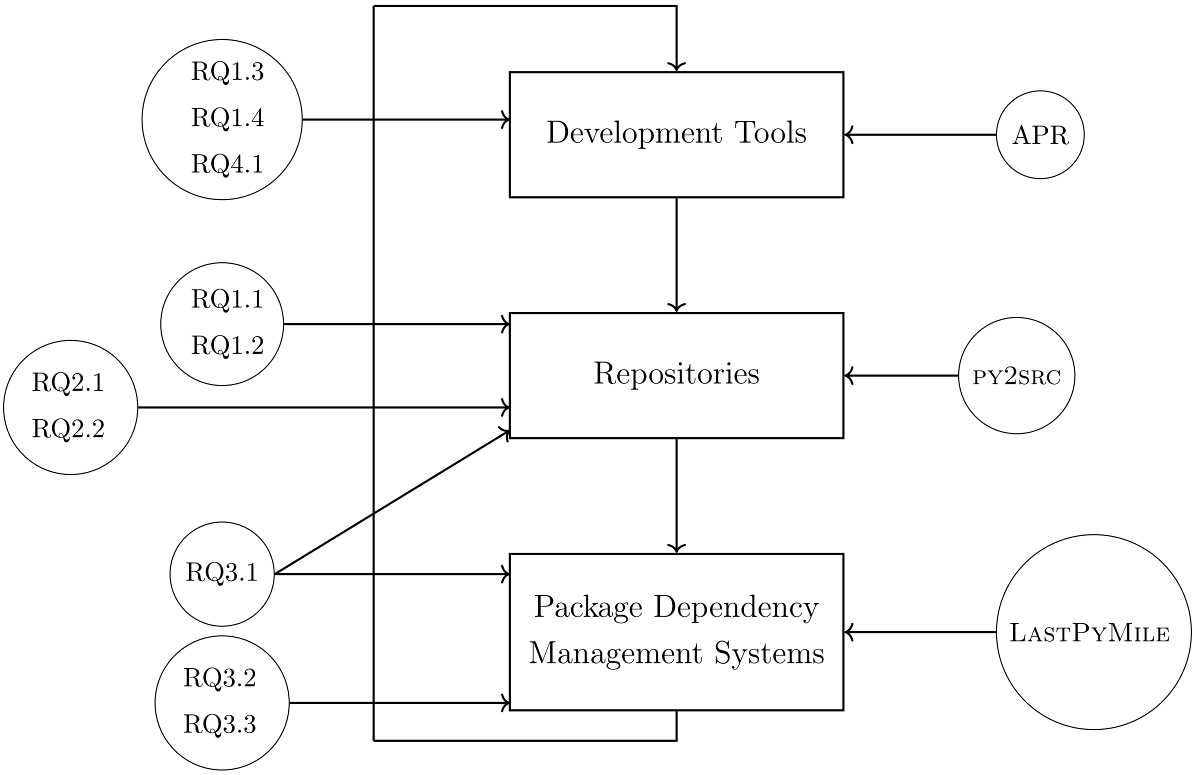


Figure 1.1: The software supply chain, and our contributions, research questions

1.3 Thesis Structure

This thesis is organized as follows:

Chapter 2 presents general concepts of the software supply chain, a systematic literature review that aggregates, summarizes relevant primary studies and techniques as well as existing services concerning each stage in the software supply chain. The content of this chapter was partially published in the following papers:

> [136] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. "A qualitative study of dependency management and its security implications." *In Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*, 2020. A poster [135] was also presented at *the 42nd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2020.

> [201] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, Antonino Sabetta. "Typosquatting and Combosquatting Attacks on the Python Ecosystem" *In Proceedings of the 5th IEEE European Symposium on Security and Privacy Workshops (WACCO)*, 2020.

Chapter 3 reports the results of 25 semi-structured interviews with professional software developers on their perception of software dependencies. All 25 interviews as well as the full analysis were reported in the following paper:

> [136] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. "A qualitative study of dependency management and its security implications." *In Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020. A poster [135] was also presented at *the 42nd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2020.

Chapter 4 depicts the results of our empirical validation for each observation from the previous chapter of this thesis. The automatic approach to identify Github URLs that correspond to PyPI packages (Subsection 4.2.1 and Subsection 4.3) and the manual validation of the Github URLs (Subsection 4.4) were reported in the following paper:

> [198] Duc-Ly Vu. "PY2SRC: Towards the Automatic (and Reliable) Identification of Sources for PyPI Package" *In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE SRC)*, 2021. This paper has also been awarded *an ACM Silver Medal* in the graduate category at the same conference.

The complete analysis of the qualitative identified quality and popularity factors of the PyPI packages for the actual selection of a new package is planned to submitted to submitted a software engineering journal:

> Simone Pirroca, Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko. "py2src: Automatic Identification of Source Code Repositories and Factors for Selecting New PyPI Packages" *To be submitted to a software engineering journal.*

Chapter 5 describes a methodology called, LastPyMile , for identifying the discrepancy between packages and sources (whether a particular code fragment in a package originates from its source code repository). The approach for detecting the discrepancy and complete analysis of the discrepancy were presented in the following paper:

> [202] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, Antonino Sabetta. "LastPyMile: identifying the discrepancy between sources and packages" *In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021. A poster [200] was also presented at *the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

Chapter 6 reports a preliminary evaluation of APR tools focusing on how much time a developer should wait for a fix. The main findings were provided in the following paper:

> [199] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci. "Please hold on: more time= more patches? Automated program repair as anytime algorithms" *In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering Workshops (Automated Program Repair)*, 2021.

Finally, Chapter 7 concludes this dissertation with important and newly opened research questions that this thesis has shed light on. In addition, we sketch out some ways to answer these questions.

1.4 Conventions Used in This Thesis

There are a number of text conventions used throughout this thesis.

Italic

Indicates service names (e.g., *libraries.io*), code names, filenames, options, and file extensions,

SMALL CAPS

Used for tool names (e.g., BANDIT), methodology names, function/method names

`Constant width`

Package name, Library name, Repository name, Service name, Developers quotes, commands and program listings, releases/versions

1.5 List of Publications

International Journals and Magazines

- Ivan Pashchenko, Duc-Ly Vu, Fabio Massacci, Henrik Plate, and Antonino Sabetta. "(In)secure Last Mile of the Free Open Source Software Delivery" To be submitted to IEEE Security & Privacy Magazine.

International Conferences and Workshops

- Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. "Typosquatting and Combosquatting Attacks on the Python Ecosystem." In Proceedings of the 5th IEEE European Symposium on Security and Privacy Workshops (WACCO), 2020.

- Duc-Ly Vu, Ivan Pashchenko, and Fabio Massacci. "Please hold on: more time = more patches? Automated program repair as anytime algorithms." In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering Workshops, 2021.

- Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. "LastPyMile: identifying the discrepancy between sources and packages." In Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2021.

- Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. "A qualitative study of dependency management and its security implications." In Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security, 2020.

Posters and Research Compition

- Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. "Towards Using Source Code Repositories to Identify Software Supply Chain Attacks." In Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security, 2020.

- Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci."Preliminary Findings on FOSS Dependencies and Security: A Qualitative Study on Developers' Attitudes and Experience." In Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2020.

- Duc-Ly Vu, "py2src: Automatic Identification of Source Code Repositories and Factors for Selecting New PyPI Packages." In the 35th IEEE/ACM International Conference on Automated Software Engineering (SRC), 2021 - **ACM Silver Medal**.

Publications that are not included in the thesis

- Duc-Ly Vu, Trong-Kha Nguyen, Tam V Nguyen, Tu N Nguyen, Fabio Massacci, and Phung H Phu. "HIT4Mal: Hybrid image transformation for malware classification." Transactions on Emerging Telecommunications Technologies, 2020.

- Duy-Phuc Pham, Duc-Ly Vu, and Fabio Massacci "Mac-A-Mal: macOS malware analysis framework resistant to anti evasion techniques." Journal of Computer Virology and Hacking Techniques, 2019.

- Duc-Ly Vu, Trong-Kha Nguyen, Tam V Nguyen, Tu N Nguyen Fabio Massacci, and Phung H Phu. "A convolutional transformation network for malware classification" In Proceedings of the 6th NAFOSTED conference on information and computer science (NICS), 2019.

Glossary

Artifact also known as *published artifact*, a software entity that contains all necessary items (e.g., files) to run the software and can be installed or directly used by project consumers. Typically, they are produced by the build process [93]. In Python, built distributions (e.g., `Wheels`) are generated from the source distributions (e.g., tarballs) [147].

Correct patches A program edit is marked as a correct patch, if it semantically fix the bug. The correctness of a pactch is usually assessed by human.

Dependency 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 [57]. Thus, we can correctly communicate the meaning of their thoughts delivered in the form of quotations later in the study.

Dependency maintenance access and modification of the source code of project dependencies. Thus, 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., submitting pull requests with new features or bug fixes.

Dependency management 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 the own code of their projects.

Library a separately distributed software component, which other software projects might reuse functionality.

Package ‘exists to be installed (or deployed)’ [148], and is a collection of pre-built and versioned artifacts for one or more target environments that is made available to consumers as an entity. In the remaining thesis, we refer to a package as a package in the PyPI ecosystem if not otherwise stated.

Package Developers/Package Owners developers who are involved in the development of a library or package for others to use.

Package repository a place to distribute pre-built packages to consumers.

Package Users/End Users developers who choose packages to include as dependencies.

Phantom a software entity (e.g., files, lines of code) present in an artifact but does not match the one submitted to the source code repository. We use *phantom lines* to refer to lines of code and *phantom files* to refer to entire files.

Plausible patches A program edit is marked as a plausible patch, if it passes all the available test cases.

Releases (Versions) A specific version of a package that can be distributed in a package reposiotry.

Repository a cloud provider with a versioning system to store and access several versions of a software project.

Source code human-readable instructions that others could check to understand the functionality of a software project.

Source code repository a place to store and maintain the project source code. From now on, we use the term for the projects hosted publicly on Github. For Github services (e.g., Github stars, forks), we use the list of terms available at [62].

Acronyms

APR Automated Program Repair.

AST Abstract Syntax Tree.

FOSS Free Software Open Source.

LE Large Enterprise.

LoC Line of Code.

NPM Node Package Manager..

OFS OSS Find Source.

OSS Open Source Software.

PyPI Python Package Index.

Regex Regular Expression.

SME Small and Eedium-sized Enterprise.

SRC Source.

UG User Group.

2

Background

> *"The most important days in your life are the day you were born and the day you find out why."*
>
> – Mark Twain

To make this chapter self-contained, we briefly introduce some concepts about the open-source software supply chain and about the PyPI supply chain in particular.

2.1 General concepts of the software supply chain

The software supply chain involves storing, retrieving, and analyzing software. The open-source software supply chain can be decomposed into four main components: Developers, Repositories, Package dependency management systems, and End Users [59]. Figure 1.1 shows simplified steps of the software supply chain in the context of this thesis.

In Figure 1.1, developers build the source code using development tools such as IDEs or debugging tools. For a team, developers collaborate with their peers mainly on a shared source code repository (e.g., GitHub [63], GitLab [64], Bitbucket [28]). In Figure 2.1, in *development* stage, developers can write all the code of a project, or they can fork any public repository to change locally and evolve the project independently. Note that we do not consider the 'fork' of a Github project in this thesis, and when applicable, we discuss these limitations in the Threats to Validity section at the end of each chapter.

During the *development* phase, developers may run automated testing tools to test the functionality of the project and the absence of security vulnerabilities. Moreover, if a project is open-source, other developers could access the code, check it, and suggest improvements. Indeed, the source code repository is often an essential source of infor-

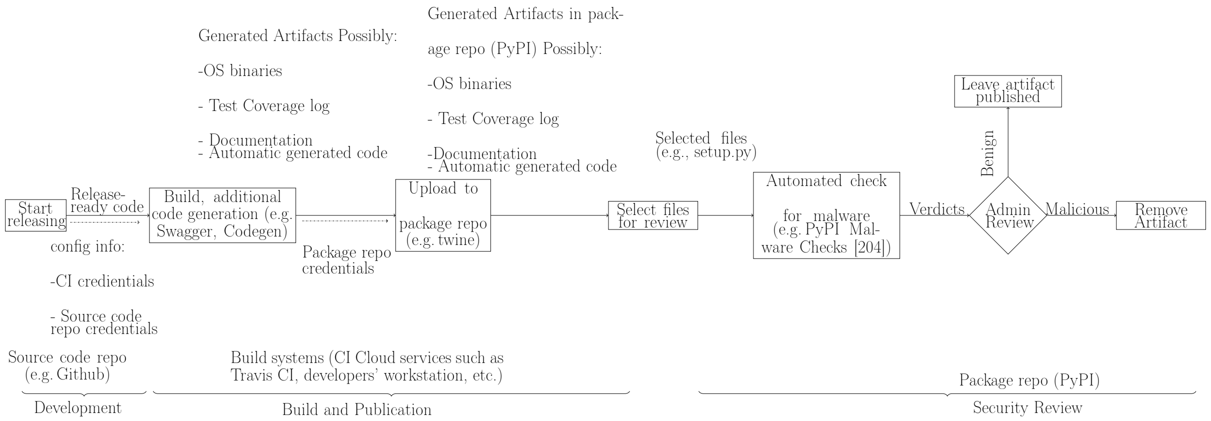

Generated Artifacts Possibly:
-OS binaries
- Test Coverage log
- Documentation
- Automatic generated code
Generated Artifacts in package repo (PyPI) Possibly:
-OS binaries
- Test Coverage log
-Documentation
- Automatic generated code
Leave artifact published
Benign
Selected files (e.g., setup.py)
Start releasing
Release-ready code
Build, additional code generation (e.g. Swagger, Codegen)
Upload to package repo (e.g. twine)
Select files for review
Automated check for malware (e.g. PyPI Malware Checks [204])
Verdicts
Admin Review
Malicious
Remove Artifact
config info:
-CI credientials
- Source code repo credentials
Package repo credentials
Source code repo (e.g. Github)
Development
Build systems (CI Cloud services such as Travis CI, developers' workstation, etc.)
Build and Publication
Package repo (PyPI)
Security Review

Figure 2.1: Current Development, Build, Publication and Security Review pipeline of PyPI packages

mation for the developers to decide on the quality of a delivery artifact (discussed in Chapter 4) [136]. For example, Github repositories provide various metrics of popularity (e.g., the number of *watchers*, *forks* and *stars*), activity of a project (e.g., number of *releases* and *commits*). We discuss those metrics in more detail in Chapter 4.

In Figure 2.1, when developers decide to make the software version available for other people (i.e., make a release), they move the code to the *build* stage. At this stage, services such as `Travis CI` [184], `Jenkins` [84], `AppVeyor` [12], or `GitHub Actions` [5] use the information stored in the project configuration files to build it. These tools fetch the source code of the package and execute the build scripts that collect all the necessary dependencies, add package metadata, generate code (e.g., SWAGGER CODEGEN [179]), and create artifacts that are ready to be distributed, like source archives, Linux, or Windows binaries, test coverage logs, and documentation. At this stage, some can download the source directly from the source code repository and recompile it in their own environments.

In Figure 2.1, developers may want to publish a built artifact to a package repository (e.g., PyPI [150], npm [125], Maven Central [124]) either manually or automatically by using the build tool from the packaging stage. Much of the complexity of using packages is delegated to a utility program called a *package manager* or *package dependency management system* (Figure 1.1). End users (of the package) typically use the software version stored and published via package repositories. They typically provide the name of a package to a package manager tool (e.g., PIP [150], or NPM [125]) to install the package from the package manager. The package manager tool merely looks for the package in the package repository by its name, identifies and resolves its dependencies, downloads all the required components, and installs them on the end user's computer. Various package

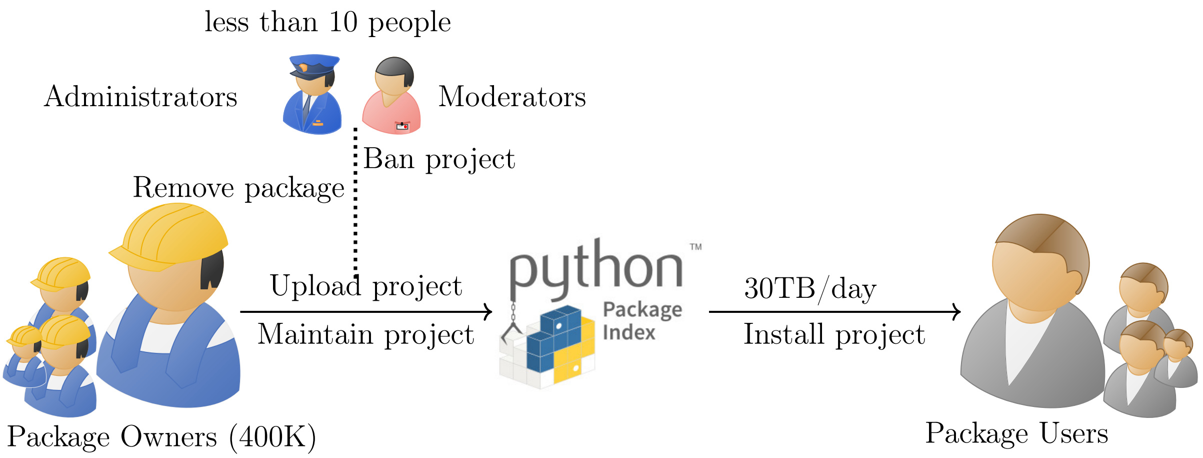


Figure 2.2: Roles and responsibilities in the PyPI ecosystem.[1]

managers differ in their mechanisms to handle the web of dependencies and their security. We discuss the security mechanisms of the popular package dependency management system for Python packages in Chapter 5.

After delivering software artifacts to end-users, during the *maintainance* phases, if there are software bugs or vulnerabilities that appear in the project, developers or maintainers are responsibly fixing them. Given many dependencies in a software project, there are repetitive bugs, and here automated program repair (APR) tools could help resolve the vulnerabilities. APR techniques allow developers to fix bugs proactively, preventing their software projects from the vulnerability chains. We provide a preliminary assessment of APR techniques in Chapter 6.

2.2 The Python Package Index (PyPI) ecosystem

PyPI is a popular repository of Python applications or packages: as of February 2022, it hosts more than 358 200 projects, and the total number of downloads exceeded 126 billion times in the year 2021. PyPI is maintained by a group of developers called Python Packaging Authority (PyPA for short). Figure 2.2 provides an overview of different roles envisioned by PyPI: End Users, Package Owners, PyPI Moderators (PyPA), and PyPI Administrators (PyPA).

Package Users provide the name of a package to a package manager tool, like PIP [149] to install the package from PyPI. Although PIP does everything automatically for installing a package, it neither requires user authentication nor performs any validation of the package. Instead, PIP merely looks for the package in PyPI by its name, identifies

[1] Data collected from `pypistats.com` on Feb 15, 2020

and resolves its dependencies, downloads all the required components, and installs them on the Package User's workstation.

Package Owners can distribute code on PyPI using the tool called SETUPTOOLS [151] that packs the original source code and generates a local distribution that is either in a source or built format [147]. Package owners register a package name on PyPI and publish the distribution artifacts of the package. If Package owners have provided both distribution types, PIP prefers to install the built distribution first. Publishing a package on PyPI is restricted to Package Owers, who can later modify (e.g., update a new version) existing packages that they have access to.

PyPI Administrators and Moderators have exclusive rights to ban or revoke packages of Package Owners. For example, if a particular package is reported as malware, the Administrators will delete the package from PyPI and block the malicious developer. The Administrators can delete the corrupted package and support the Package Owners in recovering their access if their package owner credentials are compromised or damaged.

2.3 Qualitative studies about developers' attitudes and practice

To better understand the state-of-the-art knowledge on the topic, we looked for it on the Elsevier Scopus database. We have made two queries as shown in Table 2.1. The papers published between 2006 and 2019 in Elsevier Scopus database [53] study developers' attitudes and report findings on at least one of the code groups identified in Subsection 3.2.4 and that mention surveys, interviews, case or qualitative studies, etc.

After an initial selection of 159 papers, two researchers individually marked papers as relevant or not relevant while a third researcher solved ties. We excluded the studies that:

- have different interview targets rather than developers (e.g., business or different subjects, salesperson) or,
- do not mention software dependencies, at least briefly. This exclusion resulted in 20 papers as shown in Table 2.3.

After a preliminary selection of 159 articles, we narrowed it down to 25 (including suggestions from anonymous reviewers). To further analyze them, we cluster the papers into the following groups based on the primary research focus:

- *Dependency studies* aimed at the qualitative analysis of the impact of dependencies on the developers' decision-making;
- *Tool/Technique validation studies* have the primary goal to validate a certain tool

Table 2.1: Search queries on Elsevier Scopus database on March 2019
Two researchers compose the search queries and the third research evaluate the queries to decide the final queries. Irrelevant papers are excluded by manually examining their abstracts and research questions (if any).

Search Query	**#Papers**	**Selected**
(TITLE-ABS-KEY (*developers*) OR TITLE-ABS-KEY (*programmers*)) AND (TITLE-ABS-KEY (*qualitative* AND *stud**) OR TITLE-ABS-KEY (*interview**)) AND (PUBYEAR > *2005*) AND (LIMIT-TO (SUBJAREA , *COMP*)) AND (TITLE-ABS-KEY (*change*) OR TITLE-ABS-KEY (*evol**) OR TITLE-ABS-KEY (*migrat**) OR TITLE-ABS-KEY (*adopt**)) AND (TITLE-ABS-KEY (*manage**) OR TITLE-ABS-KEY (*maint**) OR TITLE-ABS-KEY (*update**) OR TITLE-ABS-KEY (fix*) OR TITLE-ABS-KEY (*repair**) OR TITLE-ABS-KEY (*debug**) OR TITLE-ABS-KEY (*find**) OR TITLE-ABS-KEY (*select**)) AND (TITLE-ABS-KEY (*need**) OR TITLE-ABS-KEY (*importan**))	158	27
(TITLE-ABS-KEY (*developers*) OR TITLE-ABS-KEY (*programmers*)) AND (TITLE-ABS-KEY (*qualitative* AND *stud**) OR TITLE-ABS-KEY (*interview**)) AND (PUBYEAR > *2005*) AND (LIMIT-TO (SUBJAREA , *COMP*)) AND (TITLE-ABS-KEY (*dependenc**))	38	6
Total	**196**	**33**

Table 2.3: 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											
	[46]	[75]	[42]	[109]	[29]	[30]	[39]	[91]	[119]	[85]	[74]	[193]	[78]	[19]	[132]	[166]	[186]	[167]	[96]	[139]	[89]	[20]	[216]	[14]	[191]	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	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

or a technique;

- *Information needs and decision-making* the reasoning and motivations behind various decisions that developers make during the software engineering process.

Table 2.3 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. Then, for each study, we report the number of participants and whether the study provided the insights for the code groups identified (see Subsection 3.2.4 for more details).

2.4 Dependency Management Issues and Mitigations

Many empirical studies [9,33,39,79,88,91,95,107,133,140,141,213] investigate the topic of security vulnerabilities introduced by software dependencies. Cox et al. [39] 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 [91,133], JavaScript [79,88,213], Ruby [88], Rust [88], etc., provide the evidence that developers often do not update software dependencies.

Derr et al. [46] surveyed Android developers to identify their library usage and requirements for more effective library updates. When updating their app libraries, developers consider bug fixing the most critical reason while security plays a minor role. Developers are wary of updating their dependencies if they work as intended. A follow-up quantitative study [82] 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. However, limited insights are provided on the developers' motivations for updating either 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. [75] reported the impact of changes to be one of the leading developers' concerns when updating their dependencies. Later, Bogart et al. [29] 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. Their later study [30] shows that breaking changes are the main factor that prevents developers from updating their project dependencies. Also, the authors observed that developers sometimes do not update their dependencies in their projects even though their company's policy recommends this. However, the studies considered only the effect of functionality issues introduced by dependencies and did not consider the impact of security concerns.

Kula et al. [91] 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 the dependencies of their projects. 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 quantitative study in Maven by Pashchenko et al. [133] suggested that the results presented in [91] might be affected by false positives as the authors considered vulnerable dependencies used only for testing purposes.

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 (Subsection 3.3.1), their further management (Subsection 3.3.2), and how developers mitigate bugs and vulnerabilities in case there is no fixed version of a dependency available (Subsection 3.3.4).

2.4.1 Technologies/tools for automating processes of 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. [193] 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. [85] 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. However, the findings might not apply to the developers' perceptions of using dependency analysis tools that do not actually analyze code.

Mirhosseini and Parnin [119] 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 do not know whether security may change the developers' reactions to automated notifications. The study focused on JavaScript developers who used the tool called GREENKEEPER.IO [70] as a dependency management tool, so its findings might not apply to other dependency management tools.

Summary: The qualitative studies of technologies/tools for automating the software engineering process report exciting observations regarding the developers' experience, but the studies that involve dependency analysis tools focus primarily on functionality aspects and, therefore, provide limited insights on how developers can use them to discover and mitigate security issues introduced by software dependencies (more discussions can be found in Subsection 3.3.3 and Subsection 3.3.4).

2.4.2 Information needs and decision making during software development

Several studies [20, 75, 89, 91, 96, 139, 166, 167] describe the information needs and decision-making strategies of industrial software practitioners. For example, Unphon and Dittrich [186] observed that the architect or key/lead developer plays a central role in designing and revising software architecture. Pano et al. [132] 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 enterprise developers' information needs and behavioral patterns but do not report security concerns on decision-making preferences.

Assal and Chiasson [14] 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. Although the paper provides a good insight into human aspects of developers' behavior towards their own code, it does not tackle third-party dependencies (i.e., other people's code).

Linden [191] 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 selecting external SDKs. However, the study provides limited insights into developers' reasoning while working with dependencies. Moreover, the findings are reported based on observing and surveying only Android developers and, therefore, might not apply to other development environments, especially those with a central dependency management system, like npm or PyPI.

Summary: The studies on information needs provide valuable 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 (see our findings in Subsection 3.3.1 and Subsection 3.3.2).

2.5 Quantitative studies on software dependencies

2.5.1 Selection of software dependencies/libraries

The library (dependency) selection is often assigned to skilled developers or software architects [136, 186], or the combination of developers, customers, teams, and team leaders [132] in the decision-making process. For brevity, in this thesis, we refer to the people who perform an actual selection of dependencies for software projects as developers.

Various qualitative studies report different factors that developers and/or software architects said to consider while selecting software libraries for their projects, such as technical (e.g., usability), human (e.g., popularity) [136, 191], or legal (e.g., license) factors [94]. For example, *licensing* is a critical concern as it may introduce legal issues [136] for using third-party code [191], and developers often have a limited understanding [191]. Another important developers' concern when selecting an external library is its security [191]: the effort allocated to the implementation stage is much higher than in the code analysis, testing, and review stages [14].

To select a dependency, developers can extract various factors from the source code repository of the dependency to evaluate its quality or attractiveness [32,47,190,219]. For instance, developers might consider factors like the commit frequency or the time to fix a vulnerability [136]. The source code repository can also provide a source for checking the reproducibility of a package [68] or malicious code injections into the packages (Chapter 5). Source code repositories (e.g., GitHub repositories) may as well store information about the security issues and fixes of the project [158] which could support developers in choosing the well-maintained project and has the security issues fixed fast [136].

When selecting a library, developers might also use the information provided by other services, such as the service `libraries.io` provides aggregated statistics about packages (e.g., top packages providing similar functionality along with their number of downloads).

On the other hand, although existing research [94,136] has successfully attempted to provide various factors that influence developers' decisions for selecting libraries, they did not offer large-scale, quantitative perspectives on the factors.

Summary: For dependency selection, developers often rely on various factors extracted from the source code repositories of the libraries. However, the considered factors for the dependency selection are often identified qualitatively, while quantitative validation of the identified factors is limited (Subsection 4.5).

2.5.2 Finding source code repository URLs that correspond to PyPI packages

While developers usually download and install a package from a package repository (e.g., Maven Central), they often examine the source code repository of that package when deciding whether include it in their project. Github, a massive worldwide software archive, provides various services to support developers to support developers in selecting libraries and maintaining the libraries, such as the activity of a package (number of *releases* and *commits*) or overall user engagement (Github *stars*, *forks*, *contributors*).

Various tools, such as FOSSTARS-RATING-CORE [163], CRITICALITY_SCORE [131], and OPENSSF SECURITY METRICS [58] profile a GitHub repository of a package to automatically provide various factors for open-source projects. However, these tools require the source code repository URL of a package that package repositories (e.g., npm or PyPI) do not require developers to declare. On the other hand, the declared GitHub URL of a particular package might be outdated, leading to a not existing page or even a different GitHub page [34] (see Example 2.3).

In practice, developers often report that they look for the source code repositories of the prospective dependencies of their projects manually [72,94,136]. Researchers typically take a similar approach [68] by assuming the links to GitHub repositories exist without revealing their Github URL discovery methodology. However, even though the manual finding of GitHub URLs is the most precise approach, one would need significant resources to scale it to all packages in the package repository. Moreover, library developers might decide to update the source code repository of their package (e.g., to change the organization of a GitHub repository). As a result, extensive human effort would be needed to maintain the collected list of GitHub URLs corresponding to PyPI packages up-to-date.[2]

The current source code finding approaches typically rely on the metadata information present in the PyPI page of a package to automatically extract GitHub URLs of the packages [190,200,201]. However, such an approach might produce an insufficient number of GitHub URLs for many packages. For example, 1104 out of the top 4000 popular PyPI packages in our sample do not have a GitHub URL in their metadata (Table 4.7).

Microsoft OSS Find Source [118] (We use OFS as its abbreviation in Chapter 4) is the only open-source tool that we are aware of, capable of automatically finding Github URLs for PyPI packages. However, by studying the tool's implementation, we found that the tool also relies on the information present in the package metadata. For example, on the dataset of the top 4000 most downloaded PyPI packages, this method returned 3369 URLs (84%) (see Subsection 4.3 for details). Moreover, OFS does not provide any supplementary indicators (as provided by our approach py2src in Chapter 4) to evaluate the reliability of the reported URLs.

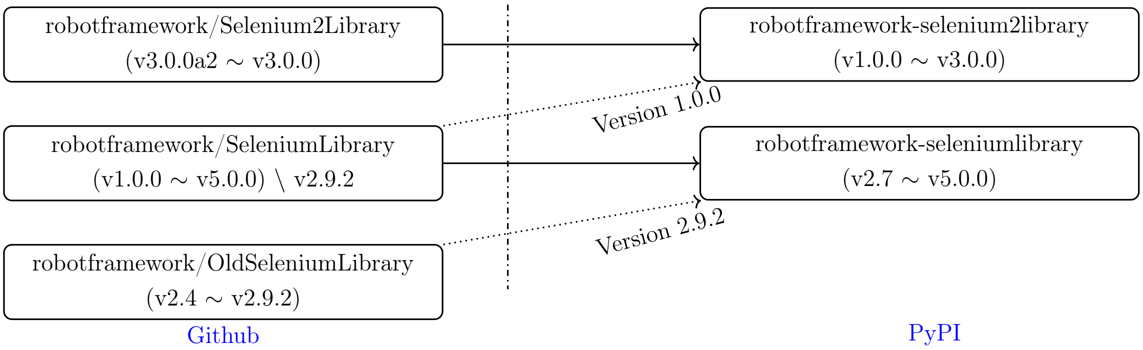
Summary: Most tools require a source code repository URL as a manual input to produce quality and popularity metrics for a project. OFS (the only existing tool to automatically extract the Github URL of a package) does not provide any reliability information of its findings (Subsection 4.4).

2.6 Discrepancies between Sources and Packages

2.6.1 The last mile from source to package

The most recent attention regarding securing the software supply chain has focused on the first element (the developers) or the last part (the end users). Most of the steps (*development* and *build and distribution* in Figure 2.1) in the PyPI software supply chain align with the chain described in Subsection 2.1. Hence, we only discuss the last stage that is specific to the PyPI ecosystem.

[2] To the best of our knowledge. Currently, there is no such dataset available.


robotframework/Selenium2Library
(v3.0.0a2 ∼ v3.0.0)
robotframework/SeleniumLibrary
(v1.0.0 ∼ v5.0.0) \ v2.9.2
robotframework/OldSeleniumLibrary
(v2.4 ∼ v2.9.2)
Github
robotframework-selenium2library
(v1.0.0 ∼ v3.0.0)
Version 1.0.0
robotframework-seleniumlibrary
(v2.7 ∼ v5.0.0)
Version 2.9.2
PyPI

When developers move stuff around repositories with different names, the automatic traceability between package and source code repositories becomes complicated as links in packages (solid lines) can point to the latest but possibly wrong source repository. Thus, a human must read the docs to find the correct Github repository.

Figure 2.3: GitHub tags and PyPI releases of SeleniumLibrary

To recap, Figure 2.1 shows a typical package process of releasing an artifact of a Python package in PyPI that consists of three main stages: *development*, *build and distribution*, and *security review*.Uploaded artifacts need to go through the *security review* stage of a package repository. For example, in the PyPI ecosystem, PyPI administrators run multiple checks (see Subsection 2.7.1) on uploaded artifacts. The checks will generate a verdict if an uploaded artifact contains suspicious behavior. The administrators are then reviewing the verdicts to decide to keep the artifact.

The match between the source code version of a project and the packages that correspond to that version is taken for granted [136]. However, *several tools (and humans) are involved at different stages of the pipeline (Subsection 2.1), and some actions may result in a published artifact containing code that is not present in the source code repository.*

During the *packaging* stage, building tools add metadata files and augment existing code files (e.g., *setup.py*) with information [3], such as license, timestamp [146], release version. Developers also use code generation tools such as Swagger Codegen that automatically generate code files (e.g., server stubs and client SDKs for APIs). Developers may also change the code of a published artifact directly to backport a bug or a vulnerability fix [201]. As a result, developers' actions might create difficulties to connect the distributed artifacts and their source code repositories, as demonstrated in Figure 2.3.

The discrepancies in the code hosted on Github and the code hosted in a package repository may be due to malicious action. For example, Table 2.4 shows the modified files in both legitimate PyPI packages: `requests` and `Flask` and malicious packages `request` and `urlib3`. Both kinds of packages differ from the source code repositories.

Table 2.4: Discrepancies in files of legitimate and malicious typosquatting packages
Distrib shows the number of lines of code that are not present in the source code repository for the requests and Flask legitimate packages as well as the difference between the malicious packages request and urlib3 from the original legitimate source code repositories of the benign packages (the targets) reported at [169]

Filepath	**Number of Lines of code**	
	Source	+Distribution
requests-1.2.2/requests/models.py	687	+5
Flask-0.5.2/flask/templating.py	88	+2
urlib3-1.21.1/setup.py	174	+23
request-1.0.117/hmatch.py	27	+81

Summary: Differences between the source and package repositories may be due to 'normal' activities (refer to Subsection 5.4 for more details).

2.6.2 Software supply chain attacks on package repositories

Software supply chain attacks occur when malicious or vulnerability is injected into different stages of the software development chain [80, 102] (described in Subsection 2.1). Weaknesses exist at all steps leading to several incidents such as package hijacking, package typosquatting, and backdoor planted in source code or updates.

Zimmermann et al. [221] study security risks for users of npm, including potential vulnerable and malicious code in third-party dependencies. The authors showed that npm suffers from single points of failure in which individual packages could impact large parts of the entire ecosystem. Attackers could compromise a minimal number of maintainers' accounts to inject malicious code into most packages.

Ohm et al. studied several attacks on different package ecosystems [129] (e.g., npm, PyPI) and found hijacking and typosquatting attacks to be the most common. Compromising the package owner's credentials would allow attackers to inject malicious payloads into the existing artifacts so that users will download and install them. Some examples of attacks are the injection of backdoors into the PyPI `ssh-decorate` package [1], Ruby `rest-client` package, or npm `even-stream` package [69]. On the other hand, attackers could commit malicious code directly into a source code repository of a package [65, 142].

Package name *-squatting attacks are more prevalent than package owner hijacking attackers [129]. In typo- and combosquatting attacks [129, 201] adversaries inject a malicious payload into the code of a popular package. Then they release this new package with a name nearly identical to the name of the original package to trick package users who mistype the package name and install the malicious one. This attack becomes especially attractive considering the limited automatic controls integrated into the package publishing process, and the certain unbalance concerning the number of package users and PyPI Administrators/Moderators (40K to 1 as depicted in Figure 2.2) [207].[3]

EXAMPLE 1. A typosquatting PyPI package `urlib3` *[169] impersonated the popular package* `urllib3` *[187]. urlib3 contains malicious code to exfiltrate user information to a remote server. The malicious package* `urlib3` *has a single release* `urlib3-1.21.1.tar` *and the same Github URL as the popular package* `urllib3` *[189].*

Summary: Malicious packages are primarily spread via package names squatting and account hijacking attacks. Attackers can inject malicious code or restore vulnerable code for later exploitation when a package is installed.

2.7 Malware detection in package repositories

Several attempts were made to identify the typo- and combosquatting packages present in the package managers [22, 176, 185, 201]. These works([181, 201]) study the potential impact of typosquatting attacks on PyPI packages based on the Levenshtein distance and number of downloads of the targeted package. For instance, using the Algorithm 2.4 we can detect fifteen out of the 28 squatting attacks in Appendix 8.8 are identified by setting the distance threshold of one. Increasing the threshold to two allows us to capture six additional attacks (21 out of 28). Our study [201] shows a large number of typosquatting candidates in PyPI that PyPI Administrators should investigate. However, relying only on package names may cause many false alerts, and a more in-depth analysis of distributed artifacts is required for verifying those alerts [201].

Garrett et al. [54, 60] use an anomaly detection-based approach on features extracted from packages' metadata and source code to detect suspicious package updates in npm. The method could reduce the review effort by 89%. However, the approach does not highlight the code injections with the existing features as it analyzes only the published artifacts. Therefore, a new approach is needed to highlight the code injections and can be adapted to provide explanations to developers.

Static code-based approaches ([26], [130], [48], [138]) can provide more accuracy in detecting malicious packages. Several approaches scan the *setup.py* file of a distributed artifact to identify suspicious code. Bertu [26] statically parses the Python installation

[3] Data collected from PyPI [150] on Feb 2020.

[4] To simplify the algorithm, unspecified paths all lead to the legitimate packages which are not required inspection.

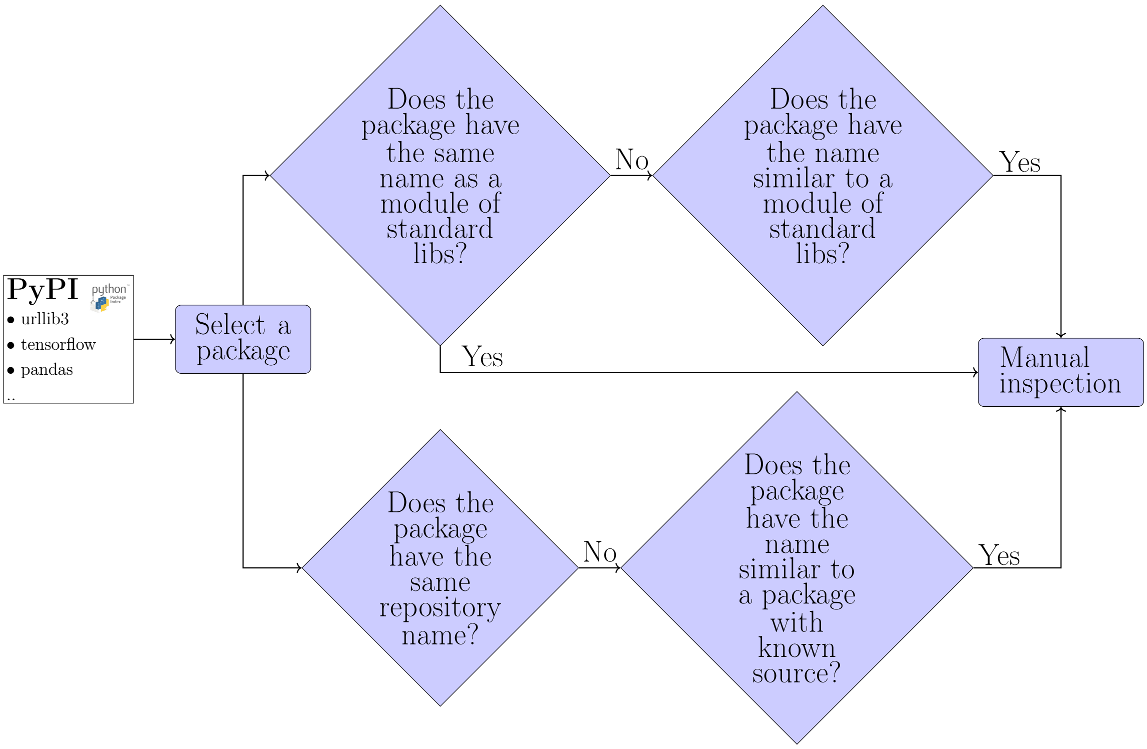

PyPI
python Package Index
• urllib3
• tensorflow
• pandas
..
Select a package
Does the package have the same name as a module of standard libs?
No
Does the package have the name similar to a module of standard libs?
Yes
Yes
Manual inspection
Does the package have the same repository name?
No
Does the package have the name similar to a package with known source?
Yes

Figure 2.4: Detecting suspicious squatting packages (Vu et al. [201]).[4]

script *setup.py* as an AST tree and looks for suspicious patterns (e.g., network connections). Malware Checks uses a set of Yara rules on the lines of code on the *setup.py* file. Similarly, ApplicationInspector and OSSGadget [118] check the distributed artifacts using a set of regular expressions to identify potential backdoors within a package. Although these approaches are fast and straightforward, they can generate many false alerts when scanning *setup.py* performing legitimate behavior in the installation process (e.g., downloading a dependency from a remote server). A viable solution is to enable these tools to scan only the lines that are not present in the source code repository and which were potentially introduced by either developers or attackers.

Several approaches use dynamic analysis to expose the malicious behaviors of the package. Unlike static analysis techniques, dynamic analysis tools rely on running the package's code to observe their behavior. For example, the tool Buildwatch [130] dynamically executes package code using the Cuckoo malware sandbox [56] and captures all system calls, such as kernel services requested by processes. Duan et al. [48] propose a hybrid approach, called MalOSS, which extracts various features of distributed artifacts using metadata, static and dynamic analyses. These methods, however, are resource-heavy which may be challenging to integrate into the development pipeline. We need a lightweight detector to help these approaches reduce the number of files and lines that need to be scanned to detect malicious code injections, making the existing techniques efficiently adapt to individual developers' development pipelines.

Summary: Although current approaches on auditing packages caught some malicious examples, their focus is on detecting malicious patterns in published artifacts, which may cause many false alerts. Also, scanning the whole code of an artifact would not be effective in code injection attacks where only a small subset of code is malicious. Instead, our approach in Chapter 5 focuses on detecting code injections in distributed artifacts, thus complementing the current techniques by reducing the number of code lines to be analyzed to detect software supply chain attacks (Subsection 5.2).

2.7.1 Detecting malicious PyPI packages

To demonstrate a security vetting process for code published in a package repository, we choose to analyze the security tools for Python packages because Python is one of the most commonly used languages and is being used by our developers in Chapter 3. In addition, Duan et al. [48, 54, 128] provides a comprehensive review of the security mechanisms for other interpreted languages including JavaScript and RubyGems.

In this thesis, we discuss the PyPI built-in protection called Malware Checks against malicious packages uploaded to the package repository. To avoid being overwhelmed by false positives, the current PyPI security review called Malware Checks [204] scans only the installation script file called *setup.py*. However, unfortunately, several known attacks [108, 169] injected malicious code into different files as shown Listing 2.4 and described in the below example. Hence, the review of only *setup.py* files is not enough.

EXAMPLE 2. The typosquatting package ***`jeIlyfish`*** *[108] mimicked the popular package* ***`jellyfish`*** *(the first L is an I) to steal SSH and GPG keys [36]. There are two injected files: setup.py, and _jellyfish.py in the typosquatting package. The malicious code is stored in the _jellyfish.py as shown Listing 2.4 and then being implicitly called by the package installer via the packages option in the setup.py file shown in Listing 2.3.*

There are other tools in Table 2.5 that support identifying malicious Python packages. Several scanning tools [26, 48] parse files into abstract syntax trees (AST) and perform rule-based searches on their nodes. Microsoft's ApplicationInspector [117] and OSSGadget [118] simply detect use suspicious lines of through regular expression (Regex) match. However, the tool authors mention that their tools generate many false positives if run on the entire package code [118].

Given that a substantial amount of code in a package is legitimate, this high number of false positives is to be expected. For instance, consider the code snippets from Listing 2.1

Table 2.5: Existing tools for analyzing Python packages
Regex *bases on the raw lines of code while* AST *analysis requires transforming the code into a tree. The hybrid analysis consists of metadata, AST, and dynamic execution of an artifact*

Package scanner	**Detection Granularity**	**Technique used**
Malware Checks [204]	*setup.py* file	Static (Regex)
MalOSS [48]	Package	Static and Dynamic
Application Inspector [117]	Artifact	Static (Regex)
OSSGadget [118]	Package & Artifact	Static (Regex)
Ohm et al. [128]	Artifact	Static (AST)
Bertu [26]	*setup.py* file	Static (AST)

and Listing 2.2. Both code snippets use the *b64decode* function from the `base64` library. Listing 2.1 is a malicious fragment that collects the user information and sends it to a remote server via a network socket, while the code in Listing 2.2 simply decodes a (benign) certificate. A package checking tool that consider the *b64decode* function as suspicious since it is often used in malicious packages will produce a true positive for Listing 2.1 and a false positive for Listing 2.2. Unfortunately, the *b64decode* function can be widely used for legitimate purposes, and the tool will generate many false positive alerts as it has no way to distinguish between benign and malicious usage without further analysis.

```
# Establishing a socket connection to a remote server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
rip = ’M’ + ’TIxL’ + ’jQyL’ + ’jIx’ + ’N’ + ’y4’ + ’ONA’ + ’==’
# Sending the encoded data via the established socket
s.connect((base64.b64decode(rip), 017620))
```

Listing 2.1: Malicious code exfiltrating information via a networking socket

```
# Decoding a bundle of certs in PEM format
der_certs = [
base64.b64decode(match.group(1))
for match in _PEM_CERTS_RE.finditer(pem_bundle)
]
```

Listing 2.2: Legitimate *b64decode* call in the `urllib3` package

```
# process all pure Python modules found in 'jeIlyfish'
packages=['jeIlyfish']
```

Listing 2.3: Invoking malicious code in the installation file *jeIlyfish-0.7.1/setup.py*

```
        ZAUTHSS = PAYLOAD
        # Decoding and executing the obfuscated payload
        ZAUTHSS = base64.b64decode(ZAUTHSS)
        ZAUTHSS = zlib.decompress(ZAUTHSS)
        exec(ZAUTHSS)
```

Listing 2.4: Malicious code injected into the file *jeIlyfish-0.7.1/jeIlyfish/_jellyfish.py*

Summary: If scanning one file in a package is feasible but not enough and reviewing an entire package is unfeasible due to the high number of false positives, a different solution is needed (skip to Subsection 5.5 for the solution).

2.7.2 Reproducible builds as an ideal solution

Reproducible builds [43] is a set of development practices that create an independently verifiable path from source to published artifacts. They could be the ideal solution to verify that no vulnerabilities or backdoors have been introduced during the build process. Given the source files (e.g., source code, build scripts), the reproducibility builds identical binary under pre-defined build environments. This technique prevents attacks against the built environment to produce packages with backdoors [156].

In practice, to achieve the reproducibility of the build process, we must eliminate varying elements in release pipelines. For example, builds should not include any CPU, timestamp, or locale information in distributed artifacts [68]. Hence, reproducible builds require a significant overhaul in the language-based package managers such as PyPI or npm [3, 68] because current release pipelines augment packages with more information, such as metadata, debug data, or automatically generated code files (See Subsection 2.6.1).

Some free software distributions, such as Debian, have procedures to identify the original source code and a different file that includes all changes made specifically for Debian, including all files related to packaging [92]. However, after trying for seven years, Debian states that "it is a stretch to say that Debian is reproducible" [44].

Summary: Reproducible builds are challenging to achieve given the diversity of packaging tools and current implementations of the release pipeline (e.g., embedding timestamp or compiler traces in built artifacts).

2.8 Automatic repairing bugs in software dependencies

Developers are reported to be 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 [109,136]. Automatic bug fixing or suggestions techniques [61, 120] have been developed to relieve developers' burden in coping with software bugs and, specifically, vulnerabilities. While automatic fixing techniques provide a complete patch that is intended to work, fixing suggestions provide partial solutions that may need to be revised by developers before its adoption into the codebase.

Given a buggy project and a set of test suites, automated program repair tools first try to localize potential faults and apply mutations to the source code until the program passes all unit test cases. Unfortunately, the generated patches, which are called *plausible patches* are not necessarily correct due to the limitation of the bug oracle [106,111]. Recent research focuses on the correctness of plausible patches by manually comprehending the patches or comparing them with the human patches.

The mutations vary in different APR techniques ranging from small changes like modification, addition, or deleting a single code line to complex patterns such as add an 'if' and then 'throw exception' mined from source code repositories [115]. In addition, there are specific bug fixing patterns shared between software bugs and vulnerabilities. By understanding what APR techniques perform on normal functional bugs, we could adapt them for software vulnerabilities. For example, some vulnerabilities could be fixed by adding an 'if' statement or changing the variable type, leading to infinite loops.

One of the requirements of an APR tool to be adopted by developers in their development pipeline is that the tool should be efficient and provide patches or suggestions in a reasonable time.

Summary: Current evaluations of automatic program repair (APR) techniques focus on tools' effectiveness, while little is known about the practical aspects of using APR tools, such as time budgets to provide a patch for developers.

3

A Qualitative Study of Dependency Management and Its Security Implications

> *"One particularly challenging aspect of dependency management is security."*
>
> – Nadia Eghbal

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. In this chapter, we 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 practical support to mitigate vulnerabilities (for example, bundling security fixes with functional changes might hinder adoption due to a 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.

3.1 Goal and Research Questions

Vulnerable dependencies are a known problem in the software ecosystems [91, 133], because free and open-source software (FOSS) libraries are highly interconnected, and developers often do not update their project dependencies, even if they are affected by known security vulnerabilities [46, 91].

A handful of studies report that developers do not update dependencies in their projects since they are not aware of dependency issues [29] or do not want to introduce breaking changes in their projects [30, 75]. Although functionality and security appear to be essential factors affecting developers' decisions [14], 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. For example, on the Android ecosystem, mobile app developers do not consider security as the top-priority task [46]. A later study by the same group [82] explained the reason behind it as a significant clash with functionality: the 'easy' updates would 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 a central dependency management system, like Maven, npm, or PyPI might have a very different approach to their projects' dependencies.

For example, an initial quantitative study of the Maven ecosystem [91] analyzed more than 4600 Github repositories and provided yet another evidence that developers keep their project dependencies outdated. However, a later study [133] 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 study is to provide a *sound qualitative analysis* of the motivation of developers between the rigid format of surveys (e.g., [46]) and the anecdotal examples that complement quantitative studies on dependencies (e.g., [91]).

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

RQ1.1: *How do developers select dependencies to include into their projects, and where (if at all) does security play a role?*

RQ1.2: *Why do developers decide to update software dependencies, and how do security concerns affect their decisions?*

RQ1.3: *Which methods, techniques, or automated analysis tools (e.g., Github Security Alerts) do developers apply while managing (vulnerable) software dependencies?*

RQ1.4: *Which mitigations do developers apply for vulnerable dependencies with no fixed version available?*

In summary, this chapter makes 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 improve 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 the development of web, embedded, mobile, or desktop applications. Some interviewees 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 in various positions, from regular developers to company CTOs, including a Java Users' Group coordinator and a lead developer of a Linux distribution. They come from 25 companies located in nine different countries.

Each interview (lasting 30 minutes 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. We give an example of our interview transcript in the Appendix 8.5.

This study illustrates the insights with quotations from the interviewees to better grasp developers' reasoning while managing software dependencies and how security concerns affect their decisions. 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. After completing the analysis, we returned the overall findings to the participating developers to check that we had not misinterpreted their thoughts.

3.2 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 may blur the developers' opinions. Instead, semi-structured interviews suited best for our goals [217]. 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 2.3).

Table 3.1 shows the descriptive statistics of the number of participants in the papers

Table 3.1: 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. [91] involved 16 developers in their study, while Sharif et al. [166] studied mailing lists from six FOSS projects but did not report the number of participants.

	Number of Studies	**Number of 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

discussed in Section 2.3. 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. These interviews may potentially introduce bias since developers may share same development strategies and approaches.

3.2.1 Recruitment of participants

As a source for finding software developers, we referred to local development communities. First, we used the public channels for these groups to post our call for interviews and contacted their reference people. Then we applied the snowball sampling approach [66] 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 work on the web, embedded, mobile, or desktop applications.

Table 3.2: Interviewees in our study

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**	**Company type**	**Country**	**Experience (years)**	**Developer 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	Sr. 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++

In total, we interviewed 30 developers[1] and eventually retained 25 for the analysis distributed over 25 different companies located in nine countries.[2] Table 3.2 summarizes the key demographics of the interviewees in our sample.

3.2.2 Interview process

To collect primary data, we had interview sessions lasting approximately 30 minutes. We met in person the interviewees who reside in our city and scheduled remote meetings with others via the video conferencing services 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 the reward we could offer. Instead, we proposed that they share their expert opinions on the topic interesting for them. We followed the Ethical Review Board procedure of the University of Trento for the management of consent and processing of data (See Appendix 8.2). We explained that all interviews would be reported anonymously, and neither personal nor company identifiable data would be made available. Also, 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 [76]. Still, we made sure all interviews included the following parts[3] (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 the selection and inclusion of new dependencies into her software projects;

[1] 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. Furthermore, their companies were unwilling to let us study their internal libraries without a legal agreement in place, while analysis of third-party libraries was not interested by developers.

[2] 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 they do not use software dependencies due to the company policy. Hence, we discarded five interviews from our analysis.

[3] 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.

- *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 automated tools for dependency analysis* - D describes an automatic tool (if used) that facilitates the dependency analysis process in her projects and provides some general details about the integration of this tool;
- *Mitigation of dependency issues* - D describes how she addresses issues in dependencies (e.g., functional bugs or vulnerabilities);
- *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.[4] The recordings were then destroyed for preserving the possibility of identifying the interviewees.

3.2.3 Interview coding and analysis

To analyze the interviews, we have adopted the applied thematic analysis [73]. Figure 3.1 summarizes our approach. It follows the principle of emergence [71], 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 [177].

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 (Table 3.2). Two researchers independently followed the "iterative process" described by

[4] 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.

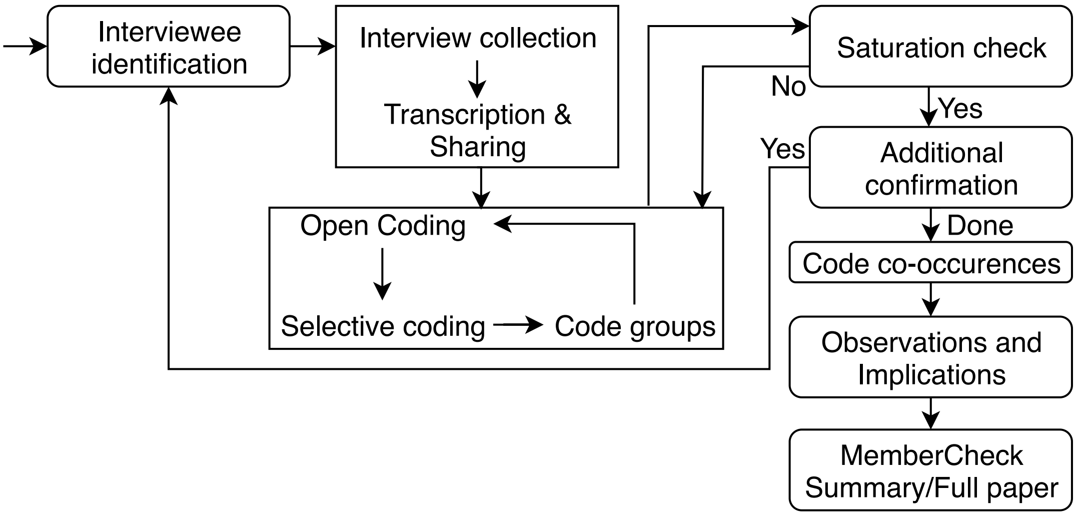


Figure 3.1: Research design of our study

Saldaña [162] to code the transcribed interviews.[5] Then they looked together at the resulting codes and agreed on the shared 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 [114], 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 3.1).

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 consolidation resulted in 151 quotations with 28 codes assigned to them. In the subsequent 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),[6] 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. Finally, we have added the codes corresponding to the interview process parts in the last step of the coding process. Hence, we have ended up with four codes corresponding to developer roles, six codes for interview process parts, and 27 codes

[5] For coding we have used the ATLAS.TI software [15]

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

for developer answers assigned to 829 interview quotes.

To validate our observations and implications, we shared the one-page summary of our analysis, along with the full version of the study, with the interviewees. Then, we asked them to validate if the results correspond to their expectations (last step in Figure 3.1).

3.2.4 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, JavaScript, 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's 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 3.4 summarizes the resulted list of codes in our study while Figure 8.1 in Appendix 8.4 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*.

Table 3.4: Codes used in the study.
The final code book consists of 27 codes grouped into 7 code groups. Figure 8.1 in Appendix 8.4 shows the frequency of occurrences of the resulted codes.

Code group	**Code**	**Description**	**Example (Developer's Quote)**
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)

3.3 Findings

We have checked whether practices established within development communities affect our findings.[7] Considering the per-language code distributions, we observed that Java, JavaScript, and Python 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.

3.3.1 RQ1.1: Rationale for Dependency 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 pool 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 the security history of a library in case the library is planned to be included in the core of their project.

The developers #10, #12, and #13 mentioned that their companies have a pool of preapproved FOSS and homegrown libraries. These libraries and their dependencies are checked for the presence of security issues and functionality bugs. Therefore, they have a higher priority to be used compared 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. [46] reported that Android developers consider security among the least important criteria for selecting new dependencies. At the same time, several recent papers underline the impact of company policies on developers' decisions to consider security. For example, the early dependency studies [30,39] reported that company policies might encourage developers to consider security, but these policies are not always followed in practice. More recent studies (e.g., [14, 191]) observed the more substantial impact of

[7] For detailed analysis, please refer to Appendix 8.7.

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 [46,85,191] 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). In addition, previous papers [30,75] suggested that developers prefer libraries that are popular and well-supported to include in 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 as 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 selecting 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. [132] reported that a combination of developers, customers, team, and team leaders 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 the 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-occurrences of *security* and *selection of new dependencies* codes in the interviews of seven 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), npm, 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., [30, 39, 46]) 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 [75, 191] provided limited insights on the impact of legal concerns on developers' decisions for selecting dependencies. For example, Linden et al. [191] 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 covered organization type (FOSS, SME, LE, UG) have reported that they consider licenses of dependencies before using them.

3.3.2 RQ1.2: Motivations for (not) updating dependencies

To answer RQ1.2, 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 by counting the code co-occurrences (Table 3.5). We do not apply a statistical test like Fisher because such a test is not appropriate to analyze co-occurrences tables as the notion of the independent experimental unit is not met for this qualitative analysis. For example, comments could come from different developers or from the same developers, so the numbers reported in the rows and in the columns are not independently drawn. Therefore, it would violate the assumptions of the test. For this reason, a statistical test is rarely applied to a co-occurrence table in most qualitative

Table 3.5: 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 8.6.

	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**

studies and papers (see [162] for guidelines).

Observation 7: *In general, developers have mixed perceptions about the dependency management process, while few developers have strongly negative and 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. Although several previous studies [29,30,39,91,119] reported developers to have mainly negative attitudes towards dependency management, we observe that several enterprise developers expressed entirely positive attitudes.

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

The most important and discussed issue by the developers in our sample were *bugs* (84 occurrences in 22 interviews). Furthermore, when the developers spoke about bugs, they often 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, it will not be exploited (#3, #4, #9, and #24). Also, the adoption of fixed versions typically does not break the dependent projects (#1, #4, #11, and #14).

Discussion. Developers are reported to be less proactive about dependencies [29] as they felt challenging to manage dependencies or lack support from vendors [91].

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 avoid updating dependencies as they lack resources to cope with the breaking changes (possibly hidden in transitive dependencies) introduced by new dependency versions.*

Many interviewees reported that they generally 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 [30,46,119] 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 test their projects thoroughly). 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 [46].

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. Thus, 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.

```
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. The previous studies report that developers do not update dependencies because they work as intended [46,119], the update contains only minor improvements [14], or there is not enough maintenance resources available [91]. In contrast, 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 critical factor for such a change in the dependency management practice.

3.3.3 RQ1.3: Automation of Dependency Management

To answer RQ1.3, 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 to identify arising issues within dependencies, so developers can assess the findings to decide whether to*

Table 3.6: 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 8.6.

	Dependency operations				
	maintenance	management	direct deps	look for info	trans deps
automated	1	**<u>18</u>**	0	7	2
code tool	0	5	0	2	0
workflow	1	**<u>45</u>**	2	13	2
dependency tool	3	**<u>26</u>**	0	9	1
manual	7	13	1	7	1

adopt a new dependency version. The dependency update itself is performed manually.

On dependency management (see Table 3.6), 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 [29,39,82] reported that developers keep dependencies outdated due to the lack of awareness about security issues. There are some reasons for this: the absence of proper security knowledge, lack of plans for security assessment, and appropriate tools [14]. 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 [119].

Observation 13: *Developers recommend using high-level metrics showing that a library is safe to use, 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 Subsection 3.3.1), 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 [119] reported that developers would like to see some supporting and explanatory arguments for 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 they use social channels, like Twitter or project mailing lists, to receive information about discovered issues and new versions of their project dependencies.

Discussion. Observation O14 suggests that dependency analysis tools share the well-known weakness of static analysis tools (e.g., [85, 193]) 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 [30].

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). Developer #9 suggests that security tools should work offline since they may disclose 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. [85] 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 [173]) 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. However, 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 [85, 193], integrated into an IDE [74], or a code review [193].

3.3.4 RQ1.4: Mitigating unfixed vulnerabilities

To answer RQ1.4, 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 could 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 first assess whether this vulnerability impacts their projects since they may not use the affected functionality. If a vulnerable dependency does not impact their project, developers may 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., [46, 91, 109]) 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 know if the vulnerability impacts their project [75, 91] 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 they could just roll back to a previous unaffected version of a vulnerable dependency.

Suppose developers decide to address a security issue in the dependencies of their projects without having a fix . In that case, 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. [30] 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 [109]. On the other hand, we observe that developers are more proactive in case of vulnerability disclosures: they check the impact of the vulnerability on their projects and provide immediate solutions by disabling the 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 [30,75,109] 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 security fixes require additional 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 [109] or even contribute to the dependency project [30,109]. 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.

3.4 Implications and Research Ideas

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 pool 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 pretty 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 the library's licenses and 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 security fixes and functionality improvements are mixed, developers would have to spend efforts to cope with breaking changes introduced by the latter. 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, welldocumented, and does not require significant development effort, 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 if a specific library version includes changes related to functionality or security. So, developers could immediately adopt security fixes and schedule the 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. Therefore, they often prefer to ignore "annoying" alerts of dependency analysis tools (O14) and fix security issues in the 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[8] (O19).

[8] Some LE developers also prefer to temporarily disable the feature within their projects, when such

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):

- 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).

3.5 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 of improving the development practice by sharing their experience and telling us their opinions on their problems. Moreover, the developers were very often motivated because we had already had a prototype of a tool that we could use to produce some dependency analysis reports for their projects. Therefore, we believe that this strategy allowed us to receive precious 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 from different companies and countries. As a result, the 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 and observational studies are hard to get. For example, de Souza and Redmiles [42] report two case studies for a total of 23 interviews. However, despite 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, Van et al. [191] did a survey of 274 developers but, to observe developers, had to recruit 44 of them and assign them laboratory-designed tasks.

an option is allowed by their company policy.

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 this study's analysis and implications 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 validated 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 actually reported dependency management practices.

Our study does not represent specific industries as they come from a sample that is not representative of those industries. We cannot disclose the identity of the participants, and the interviewees are agreed to be transparent and open under the promise that their interviews would be anonymous. This risk would not be balanced by the disclosure as the sample would still be limited, so one could still argue that the findings are not significant.

3.6 Conclusions

This chapter 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 minutes, 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. Table 3.7 summarizes our findings and their implications.

Table 3.7: Summary of Results

RQ	**Analysis Summary**	**Implications**
RQ1.1	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.
RQ1.2	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).
RQ1.3	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 used fragments of the libraries; (ii) show the affected components of the dependent projects; (iii) suggest if there exists a fixed version and if its adoption introduces breaking changes.
RQ1.4	The interviewees suggested the following actions when a vulnerability is discovered in a dependency but no newer version fixes it: (i) assess if this vulnerability impacts them since 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 library version; (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.

4

PY2SRC: Automatic Identification of Source Code Repositories and Factors for Selecting New PyPI Packages

> *"It won't be covered in the book. The source code has to be useful for something, after all ..."*
>
> – Larry Wall

Selecting which libraries ('dependencies' in the software industry's jargon) to adopt in a project is an essential task in software development. Recent qualitative studies show that popularity, licensing, and quality (from security to timeliness) of the corresponding source code are some of the factors behind this selection. In this chapter, we seek an empirical, quantitative confirmation of the role of popularity and quality factors in the adoption of a dependency or just its download as a software package in the Python Package Index (PyPI) ecosystem. We develop a tool called PY2SRC to automatically identify GitHub source code repositories corresponding to packages in PyPI and extract the popularity and quality factors of the source code from GitHub that should explain the corresponding popularity (adoption) of packages in PyPI. The regression analysis shows that the number of stars, closed issues, and number of found and fixed vulnerabilities correspond to an increase in adoption (number of dependent repositories and package downloads). On the other hand, the frequency of releases contributes to decreasing the number of package downloads. The proposed PY2SRC tool facilitates the selection of new software dependencies for Python projects by providing their source code repository URLs along with a set of associated reliability metrics.

4.1 Goal and Research Questions

Several recent papers discussed qualitative reasons behind the developers' choice of dependencies [91, 136]. In particular, they investigated how and why developers select dependencies to include in their projects and how security concerns affect their decisions.

Developers reported that for dependency selection, they refer to high-level information (e.g., *meta-metrics*) that demonstrates package adoption of a library rather than low-level details of a library source code [Observation 5 in Chapter 3] Still, most of the factors mentioned by developers are connected with the library source code repositories (e.g., GitHub), such as the number of *stars*, *forks*, *project contributors*, *open/closed issues*, speed of issue resolution, general code quality.

Although there are many observations about the general thinking of developers, there is very little evidence that the 'factors' suggested in the qualitative studies (e.g., interviews or surveys) explain the popularity of the libraries on a large scale. Moreover, extracting these factors requires identifying the source code repositories of the libraries: the state-of-the-art automatic tools (Subsection 2.5.2) capable of extracting such factors typically assume the corresponding source code URLs to be always available.

This chapter aims to find an automatically and reliable way to identify Github repositories corresponding to PyPI packages. Furthermore, we empirically study the influence of the factors extracted from the source code repositories on adopting new PyPI packages. In particular, we focus on the following research questions:

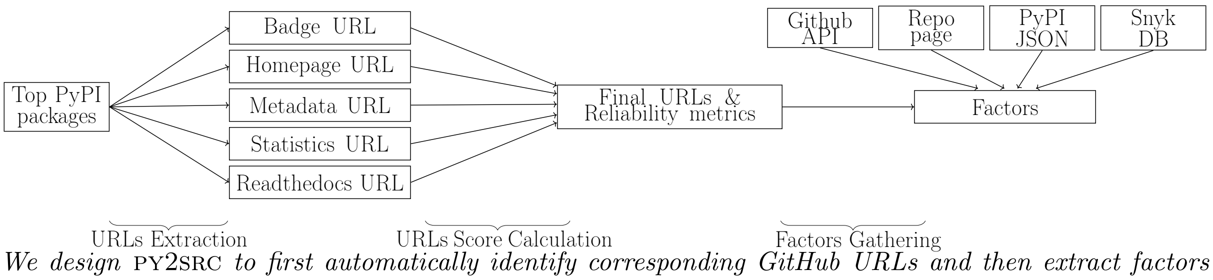
RQ2.1: *How can we combine the information displayed on PyPI pages to accurately identify GitHub URLs corresponding to PyPI packages and validate them?*

RQ2.2: *Which of the suggested relevant factors do explain the adoption of a library?*

This chapter makes the following contributions:

- We developed a tool called PY2SRC to automatically identify the GitHub source code repositories corresponding to packages in PyPI and extract the quality as well as popularity (adoption) factors of these repositories
- We performed an extensive empirical study of the quality factors that should explain the corresponding popularity (adoption) of PyPI packages.

Our regression analysis on the top most downloaded PyPI packages shows that *Stars*, *Closed Issues*, and the number of fixed vulnerabilities increase the adoption of packages in both GitHub and PyPI. In contrast, *Release Frequency* contributes to decreasing the number of package downloads from PyPI.



We design PY2SRC *to first automatically identify corresponding GitHub URLs and then extract factors reported by developers as reasonable proxies for package quality and adoption.*

Figure 4.1: Methodology process flow

4.2 Methodology

Figure 4.1 describes our procedure for finding and verifying the resulting Github URLs. First, we identify the GitHub source code repository URL that corresponds to a PyPI package. Then we use this URL to extract the factors that, reported by the developers' interviews [94, 136], are used to decide package quality and adoption.

4.2.1 Finding GitHub URL of a PyPI package

To find the GitHub URL of a PyPI package, we refer to the information extracted from PyPI. For example, a PyPI page might reference the corresponding source code repository in the package description or the statistics section. Also, each PyPI package has a package metadata in JSON format that might as well contain a GitHub URL reference. Existing tools such as OFS uses package metadata as the primary information source for identifying corresponding GitHub URLs. Table 4.1 lists the information sources that could point to the corresponding GitHub URL of a PyPI package.

EXAMPLE 3. The PyPI page of the `six` *package [24] contains the GitHub repository URL [25] in the description, statistics, homepage, and metadata sections [23]. There are also the Readthedocs page and the Travis badge pointing to the GitHub repository.*

Table 4.1: Information sources for identifying GitHub URLs of PyPI packages

Information Source	**Description**
Badge	Badges allow developers to, for example, easily access the source code of the package (e.g., GitHub badge) or understand the status of the project build (e.g., Travis badge[a]). We extract URLs pointing to GitHub repositories from these badges.
Homepage	A PyPI page might contain a link to the *homepage* of the given package. We check if this URL points to GitHub.
Metadata	Every PyPI package has a metadata information that can be accessed via `https://pypi.org/pypi/{package_name}/json`.
Page Conservative URL	We look for GitHub URLs mentioned at the entire PyPI page and return a URL only if there is just that single URL mentioned among the whole page.
Page Majority URL	We look for GitHub URLs mentioned on the PyPI page and return the URL with the highest occurrences.
Readthedocs	*Readthedocs.io* documentation web pages often include GitHub URLs. Hence, we parse the *Readthedocs.io* page of a package to extract the corresponding GitHub URLs.
Statistics	A PyPI page might include a Statistics section that displays some GitHub factors (e.g., stars, forks, contributors). We use this section to extract the corresponding GitHub URL.

[a] GitHub URL can be obtained from the Travis Badge by substituting '`https://travis-ci.org`' with '`https://github.com`'.

We combine the URLs reported by the information sources into the *Mode URL* as follows:

- We extract all URLs from the information sources;
- If there is no URL pointing to a GitHub repository,[1] we return an empty MODE URL;
- If all information sources point to the same URL, we set MODE URL to be equal to this URL;
- If URLs point to different GitHub repositories, we set MODE URL to the URL that has a bigger number of information sources supporting it;
- If several URLs are supported by an equal number of information sources, we set MODE URL to the first observed URL, according to the order of Table 4.1.

For each identified GitHub URL (including the *Mode* URL), we extract additional reliability metrics (shown in Table 4.3) that support the correspondence of this URL to the PyPI package. A metric might have a value of 1 or -1 depending on whether the metric condition is satisfied. If the corresponding information for a metric is not available (e.g., there is no PyPI badge), we assign value zero to the metric. Hence, the score for a URL may range from -4 to 4.

EXAMPLE 4. Consider the GitHub repository [25] for the six package reported by PY2SRC*: the repository has the equal names (+1) and descriptions (+1) to those in PyPI, Python is listed in the Languages section (+1), and there is a badge pointing to the PyPI page of the six package (+1). As a result, the reliability score is 4.*

We have further performed the manual validation of the proposed approach for finding corresponding GitHub URLs on the subset of 325 packages for which the identified URLs have the lowest reliability as follows:

- 181 packages where OFS and MODE report different GitHub URLs. In this way, we investigate the conflicts and look for an effective way to resolve them;
- 74 packages whose URLs reported by OFS have low reliability scores. In doing so, we show how our approach complements OFS in finding correct GitHub URLs;
- 70 randomly selected packages (out of 282) whose URLs are supported by only one or two information sources (out of seven). These packages help us explore new information sources to find correct GitHub URLs.

For these packages, three researchers independently identified the corresponding GitHub URLs without considering the suggestions from information sources and reliability met-

[1] We perform a URL validation to check if the URL is from GitHub and actually working. In case there is a redirect, we follow it and report the last URL.

Table 4.3: Reliable attribution metrics

A metric has value 1 if the condition is satisfied, -1 if the condition is not satisfied, and 0 if the metric cannot be computed (e.g., a badge is absent). All six metrics are used to assign the manual score for the 45 sample packages, while only the first four ones are also computed automatically once the GitHub URL is identified.

Metric	**Description**	**Condition for calculating Automatic score**
Name similarity[a]	We use the Levenshtein distance [101] to calculate the difference between the names of a package in PyPI and its GitHub repository. The distance ranges from `0` (the strings are equal) to the maximum length of the compared strings. However, sometimes the Levenshtein distance could be non-zero because one name is a substring of the other: the name of the source code repository of the `future` package is `python-future`. Hence, we also check whether names are substrings.	Distance[b] < 2 or isSubstring
Description similarity	The Levenshtein distance between the project descriptions (truncated at 1500 characters) in PyPI and GitHub pages. The lower the distance, the more credible the URL is.	$\frac{Distance}{Length} < 0.5$
Python language	Describes if Python is included in the `Languages` section of the GitHub page.	Python in Languages sec.
PyPI badge	Presence of a badge on the GitHub page that points to the PyPI page of the package under analysis.	Badge is present and points to the analyzed package
Authors similarity	PyPI package maintainers are present in the list of the GitHub repository contributors.	N/A (manually assessment)
Releases similarity	Alignment of GitHub repository tags and PyPI package releases.	N/A (manually assessment)

[a] We combine Levenshtein distance and isSubstring to estimate the name similarity since we want to have a single metric that corresponds to a human intuition that the package names are similar. We prefer to keep a single metric as splitting might create an imbalance in the reliability score towards the name of a package in PyPI and its corresponding GitHub repository with respect to the other attribution metrics.

[b] The threshold of two characters cannot be applied for packages whose name has only one character. However, there is only one such package (‘q’) in the set of the most downloaded packages. Moreover, the corresponding GitHub URL name for this package is equal to the package name.

rics. In particular, each package was analyzed manually by two researchers to find the corresponding GitHub URLs. If the validation resulted in a different URL, the third researcher resolved the conflict. So for each package, we had a complete agreement on the identified URL (if any) by the three researchers.

We first randomly sampled 45 packages from the set of 325 packages and manually estimated the correctness of the URLs using the checklist, which includes the four reliability metrics from Table 4.3.

Then we extended the manual validation to all 325 packages (10% of the top downloaded packages) to construct the final dataset of GitHub URLs corresponding to PyPI packages for factors evaluation.

4.2.2 Evaluation of Quality and Popularity Factors

We referred to the previous qualitative studies [94,136] for the list of factors developers consider while selecting new software dependencies for their projectsv(Table 4.5).

We use the following two variables as proxies for package popularity: the number of dependent repositories and package downloads. The first variable represents packages' popularity among developers who actually include these packages as their project dependencies. At the same time, the number of package downloads represents packages' popularity among package users (not necessarily developers) who download these packages from PyPI (also as dependencies for other packages).

We collected data for the factors from GitHub on March 2021 and used GitHub APIs[2] as the primary extraction method for such factors. Additionally, we crawled the GitHub repository page corresponding to a package to extract the factors that are not accessible using the GitHub APIs (e.g., the number of commits and releases). Table 4.5 provides the descriptions of each factor used in our study.[3]

Regarding the data for vulnerabilities affecting PyPI packages, we referred to the Snyk Vulnerability DB [172] as it is one of the most complete, open, and updated vulnerability databases [171]. As of March 2021, the Snyk database contained 1276 vulnerabilities, 707 of which were related to 220 packages in our sample.

Finally, we performed a regression analysis using the first ten quality factors as dependent variables and the last two popularity factors as independent variables (Table 4.5).

[2] It returns a JSON containing repository metadata, accessing from `https://api.github.com/repos/{org_name}/{repo_name}`.

[3] Some of the definitions are taken from the GitHub glossary [62].

Table 4.5: The factors developers rely upon when selecting new dependencies
The first ten factors (the rows shaded in grey) represent the factors suggested by previous qualitative studies [94, 136], while the last three factors serve as proxies for package popularity.

	Factor	**Description**	**Source**	**Extraction Method**
Quality	Stars	The number of GitHub users displaying an appreciation for the repository	GitHub API	Extract the `stargazers` tag in a Github repository
	Contributors	The number of GitHub users who do not have collaborator access to a repository but contributed to a project and had a pull request merged into the repository	GitHub page	Extract the `Contributors` from from a Github repository
	Open issues	The number of not addressed issues in the repository	GitHub page	Extract the `Open issues` from a Github repository
	Closed issues	The number of resolved issues linked to the repository	GitHub page	Extract the *Closed issues* from a Github repository
	Open/Closed issues	The fraction of open and closed issues	GitHub page	Number of *Open issues* divided by number of *Closed issues*
	Commit frequency	The average number of commits per month	GitHub page	Total the number of months divided by total number of commits since the repository creation date[a]
	Release frequency	The average number of days between two releases	GitHub page	Number of days divided by number of releases/tags since the repository creation date
	Time to close issue	The average number of days between opening and closing of an issue	GitHub API	Average of the number of days between opening and closing each closed issue
	Vulnerabilities	The number of vulnerabilities discovered in a package	Snyk DB	Extract the vulnerabilities related to PyPI packages from [192]
	License	Set to 1 if package's license is permissive [116], -1 if it is copyleft, and 0 if it is unknown	GitHub page	Extract permissive licenses
Popularity	Dependent repos	The number of GitHub repositories that have the current repository as a dependency	GitHub page	Extract from the *Used by* section in a Github repository
	PyPI downloads	The number of users that downloaded the PyPI package	PyPI JSON	Extract from [192]

[a] `created_at` value of the JSON metadata returned by GitHub API.

Table 4.7: Information sources reporting a URL for PyPI packages
The last columns refer to OFS *and the two proposed approaches to combine information sources (the first columns). The Final URL is equal to the Mode URL, if the Mode URL is present, and to the* OFS *URL otherwise.*

	Badge	**Homepage**	**Metadata**	**Page Conserv.**	**Page Major.**	**Readthedocs**	**Statistics**	**OFS**	**Mode**	**Final**
# Packages	1491	2127	2896	2210	3069	822	3074	3369	3473	3493
% Packages	37%	53%	72%	55%	77%	21%	77%	84%	87%	87%
No URL	2509	1873	1104	1790	931	3178	926	631	527	507

4.3 Information sources for finding GitHub URLs of PyPI packages

To check the proposed approach for finding the GitHub URLs in the selected information sources (Table 4.1), we use the list of 4000 top-downloaded packages[4], used by the previous works to study different aspects of PyPI packages in both academia [31] and industry [45, 123, 194]. The presence of a package within the list of the top downloaded packages is a good indicator of its popularity [211].

Table 4.7 shows the number of GitHub URLs reported by the information sources. The *Statistics* section of a PyPI page contains GitHub URLs for 77% of packages. In contrast, the *Readthedocs* page allows us to identify GitHub URLs for 21% of the packages. A PyPI package might have several different URLs reported by the information sources:

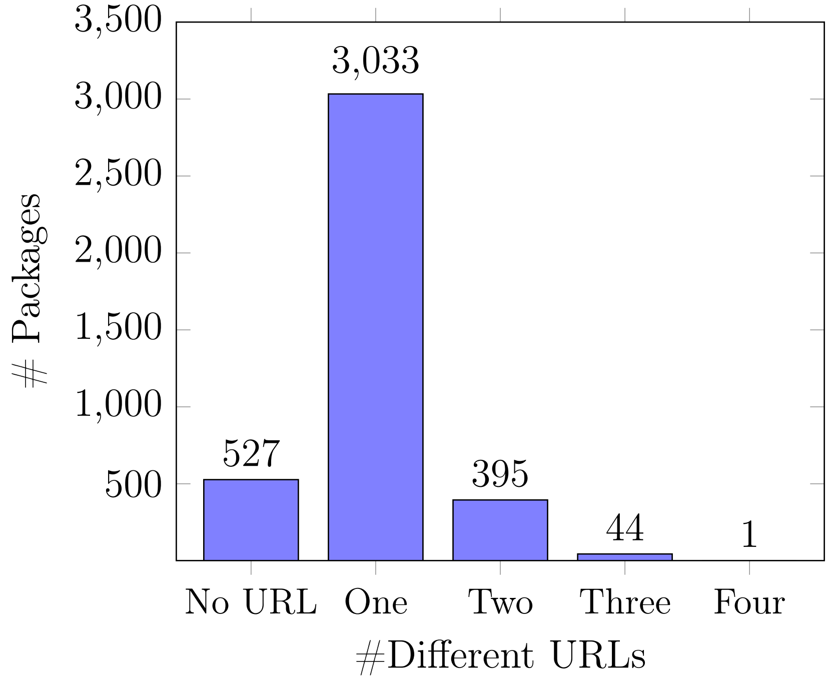
EXAMPLE 5. The Information sources suggest four different GitHub URLs for the `zappa` *package: the most present is returned by Badge, Metadata, and Statistics, while Homepage, Page Majority, and Readthedocs report the other three URLs.*

When considering the aggregated URLs of the information sources, the *Mode URL* allows us to find GitHub URLs for 87% of the selected packages. At the same time, OFS can identify GitHub URLs for 84% of packages in our sample. Furthermore, the Final URL (i.e., combined Mode and OFS URLs) enables us to find GitHub URLs for 20 additional packages.

Figure 4.2 shows the distribution of the number of URLs identified by the number of information sources. In particular, the information sources return one URL for 3033 packages (76%), two different URLs for 395 packages (10%), three different URLs for 44 packages (1%), and only the ZAPPA package has four different URLs.

Table 4.8 shows the number of the same URLs reported by individual information sources (i.e., agreement). We observe that the *Statistics* information source has the highest number of URLs that agree with other sources. For example, 3061 (out of 3074) *Statistics*

[4] Number of downloads in the last 365 days as of September, 2020 [150]



For three out of four packages, the information sources return one GitHub URL, while for one out of eight packages no URL is returned.

Figure 4.2: Distribution of the number of identified Github URLs of the packages

URLs are equal to the URLs found by OFS, while URLs for only 13 packages are different. However, OFS identified URLs for 295 packages, where the Statistics did not report a URL. Similarly, we observe that the Metadata and OFS report different URLs for six packages. However, the Metadata information source does not report URLs for 479 packages where OSS FIND SOURCE returns a URL. Also, the URLs reported by *Statistics* and *Metadata* information sources differ for 16 packages.

Discussion. An individual information source reports fewer URLs compared to the state-of-the-art OFS tool. However, the MODE combination of the sources allows us to increase the number of identified GitHub URLs by 3%. Hence, the proposed information sources are promising to identify GitHub URLs corresponding to PyPI packages automatically.

The information sources may contain different GitHub URLs because package maintainers might host another repository to provide more information for the original package. Unfortunately, the maintainers forgot to update the new repository in their migration.

EXAMPLE 6. The `robotframework-selenium2library` PyPI page points to the Github repository `https://github.com/robotframework/SeleniumLibrary` that contains the code for the releases before `v1.8.0` and after `v3.0.0a1`. The other releases are stored in a different repository `https://github.com/robotframework/Selenium2Library`.

Table 4.8: Information sources agreement on found URLs

For each row `i` *and column* `j`*, the corresponding cell contains the number of found URLs by the information source* `i` *that equal to the URLs found by the information source* `j`*.*

	OFS	Badge	Homepage	Metadata	Page Conserv.	Page Major.	Readthedocs	Statistics
OFS	**3369**	1371	2018	2890	2188	2891	560	3061
Badge	-	**1491**	959	1169	954	1191	383	1235
Homepage	-	-	**2127**	1828	1902	1778	393	1905
Metadata	-	-	-	**2896**	1888	2674	464	2880
Page Conserv.	-	-	-	-	**2210**	1908	385	2015
Page Major.	-	-	-	-	-	**3069**	488	2764
Readthedocs	-	-	-	-	-	-	**822**	495
Statistics	-	-	-	-	-	-	-	**3074**

Still, most information sources (e.g., 76%) report one URL, and we observe the high agreement between the sources. Therefore, the reported URLs are likely to be the actual URLs corresponding to the source code repository URLs of PyPI packages. We check this further in our following sections using the set of manually identified GitHub URLs that correspond to the 325 PyPI packages and the reliable attribution metrics (Table 4.3).

Summary: The proposed information sources in Table 4.1 enable us to identify GitHub URLs for more packages than the existing tool tool OFS (Table 4.7) for an automatic finding of GitHub URLs that correspond to PyPI packages by 3%.

4.4 RQ2.1: Combining information sources

To evaluate the URLs reported by the information sources, we manually identified GitHub URLs for 325 packages (see Subsection 4.2). We manually inspected the PyPI pages of the selected packages to identify the corresponding GitHub URL. If the web page inspection did not allow us to find the corresponding GitHub URL, we used Google search to look for the corresponding source code repository.

Developers of some packages can use a different platform than GitHub (e.g., GitLab [64], BitBucket [28], or SourceForge [175]) to store their source code while maintaining a mirror of their repository on GitHub. In such cases, we check if a GitHub mirror corresponds to the original repository in terms of the aligned releases and commits. If the GitHub mirror exists, we consider it to be a valid GitHub URL of a package.

EXAMPLE 7. The `peakutils` *package [122], has the original source code reposiotry in BitBucket* `https://bitbucket.org/lucashnegri/peakutils`*, and the aligned GitHub mirror* `https://github.com/lucashn/peakutils`*.*

4.4.1 Combining information sources.

To find the most reliable way to combine the URLs of the information sources and verify the accuracy of the automatic extraction of reliable attribution metrics, we randomly selected 45 packages from the 325 packages for which we identified GitHub URLs manually:

- 15 packages where OFS and MODE reports the same URL;
- 15 packages where the manually identified URL equals to the MODE, but not to the OFS URL;
- 15 packages where the manually selected URL equals to OFS URL, but not to the MODE.

A GitHub repository can host multiple packages in its subdirectories, which are considered *mono repositories* [143]. In such a case, we consider only the parent repository as the right one representing all its sub repositories. Therefore, even though we can find a specific URL that corresponds to a package that is a part of a mono repository, we do not consider such URLs for the analysis of reliable attribution metrics.

EXAMPLE 8. 19 different PyPI packages (e.g., `grpc-google-iam-v1`) belong to the `googleapis` repository on Github [11].

In our sample, we observed that four of the selected packages correspond to the `googleapis` and `twitter-archive mono` repositories [13]. Therefore, we do not consider them for further analysis.

For each of the 41 URLs, we manually calculate the reliable attribution metrics (shown in Table 4.3): each check gives a URL a score of 1, -1, or 0 if the URL is respectively correct, incorrect, or empty. The reliability score of a URL is the sum of all the metrics values. Hence, a URL has a reliability score in the range from -6 to 6.

EXAMPLE 9. The GitHub repository name [16] of `awscli` [17] differs from the package name by one character (+1). The GitHub and PyPI descriptions are the same (+1). The repository has a Python percentage of 100% (+1). There is a match between the list of GitHub tags and PyPI releases (+1). PyPI maintainers can be found in the list of GitHub contributors (+1). There is no badge pointing to the PyPI page on the GitHub page (0). The total reliability score we assign to the URL is 5.

We then check the agreement between the URLs reported by the information sources (including OFS and MODE) and manually identified URLs. Finally, we evaluate the agreement score by counting the number of matching reliable metrics for the manually

Table 4.9: Agreement between the reliability scores of URLs returned by information sources and manually identified URLs
We used the Pearson correlation coefficient to rate the agreement between the returned URLs.

	Badge	**Homepage**	**Metadata**	**Page Conserv.**	**Page Major.**	**Readthedocs**	**Statistics**	**OFS**	**Mode**
Agreement	97%	92%	83%	85%	62%	92%	81 %	80%	83%

selected URL of a given package (+1 value) and assigning the positive value to the URL if it equals the manually identified URL and negative otherwise.

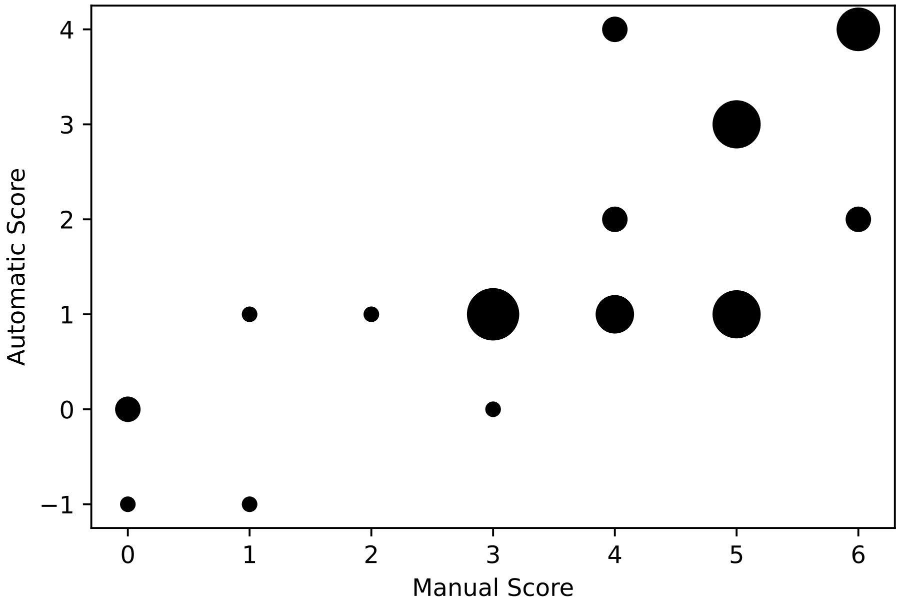
If the URL returned by an information source is empty, we assign it a zero score (i.e., it does not help us decide whether the URL is reliable). Then we compare the correlation of reliability and agreement scores for each URL.

As shown in Table 4.9, we use the Pearson's correlation cofficient [21] to measure the agreement between the information sources in Table 4.1 plus OFS, and Mode URL. While most of the information sources demonstrate a high degree of agreement (75%), the Page Majority information source has a lower agreement level (62%). Moreover, our manual analysis of the URLs reported by Page Majority suggests that it is prone to report a high number of wrong URLs, which might lead to less precise URLs returned by the Mode URL. On the other hand, we realized that Page Conservative information source often does not return any URL (e.g., many false negatives). In contrast, when the URLs are actually returned, they can also be found by other information sources. For this reason, we decided to exclude Page Majority and Page Conservative URLs from further analysis.

Figure 4.3 shows the bubble plot to compare the manually and automatically computed URL reliability scores: each circle in the plot has a size proportional to the number of URLs with the corresponding tuple of manual-automatic scores. The plot clearly shows the positive correlation between the manual and automatically computed reliability scores over the manually selected URLs. Moreover, the high agreement level of 73% confirms the visual relationships between these manually and automatically computed reliability scores. Hence, automatically calculated URL reliability scores are likely to correspond to the scores assigned by a human.

Further, we compare the performance of OFS and Mode URL in terms of precision and recall. Table 4.10 shows that both methods produce equal recall, but Mode URL has 4% higher precision than OFS (i.e., Mode URL returned more URLs equal to the manually identified ones).

Our manual analysis of the agreement between the URLs reported by both methods suggests the following:



Big bubbles represent that the point is plotted multiple times. Agreement level = 73%.

Figure 4.3: Manual and Automatic reliability score over the 45 selected packages

Table 4.10: Performance of OFS and methods that combine information sources
ModeThenOSG demonstrates the best performance, hence, we use it as the best combination of the sources. In the rest of the chapter, we refer to ModeThenOSG as Final URL.

Resulting URL	**Precision**	**Recall**
OFS	86%	68%
Mode	96%	68%
OFSThenMode	90%	100%
ModeThenOFS (Final)	95%	100%

Table 4.11: PY2SRC sources and tools comparison in terms of Precision and Recall among the 325 packages
The last columns refer to OFS *and the two proposed approaches to combine the remaining information sources (the first columns).*

	Badge	**Homepage**	**Metadata**	**Readthedocs**	**Statistics**	**OFS**	**Mode**	**Final**
Precision	93%	92%	97%	54%	95%	80%	86%	83%
Recall	23%	32%	19%	11%	27%	48%	78%	90%

- When both methods returned the same URL, this URL corresponds to the manually identified one;
- When both methods return different URLs, three out of four URLs identified by MODE URL correspond to the manually identified URL while only one out of three for OFS;
- When OFS returns no URL, MODE URL identifies the URL corresponding to the manually identified one;
- When MODE returns no URL, OFS identifies the URL corresponding to the manually identified one.

The above observations suggest that the combination of OFS and MODE seems promising. Indeed, we observe that by combining the two approaches, we can gain 100% recall and increased precision (the last two rows of Table 4.10). Taking first the OFS URL and, if it is empty, returning the Mode URL (i.e., so-called OSGTHENMODE) allowed us to have 90% precision. In comparison, the inverted approach MODETHENOSG leads to 95% precision. Hence, we adopt the latter combination MODETHENOSG as the most reliable way of combining the information sources. Further, we refer to the URLs obtained by the MODETHENOSG combination as the *Final* URLs.

4.4.2 Scaling the evaluation to 325 packages

We scale the evaluation to 325 packages, for which we identified corresponding GitHub URLs manually. First, we calculate precision and recall for the selected information sources, OFS, MODE URL, and *Final URL* using the manually identified URLs as correct URLs. Table 4.11 shows that *Metadata* has the highest precision of 97%, while *Readthedocs* is the least precise (54%) information source. However, both the information sources *Metadata* and *Readthedocs* have a relatively low recall (19% and 11%), indicating that they report many empty URLs. Among individual information sources, *Homepage* is the best information source in terms of recall.

We observe that OFS has lower precision but higher recall than individual information

Table 4.12: The correlation between the automated computed reliable attribution metrics
Each cell in the table denotes a pair of values: Effect (r) and P-value (p)

	Name similarity		Description similarity		Python language		PyPI badge	
	r	p	r	p	r	p	r	p
Name similarity			+0.25	0.000	+0.36	0.000	+0.22	0.000
Description similarity					+0.17	0.002	+0.25	0.000
Python language							+0.15	0.008

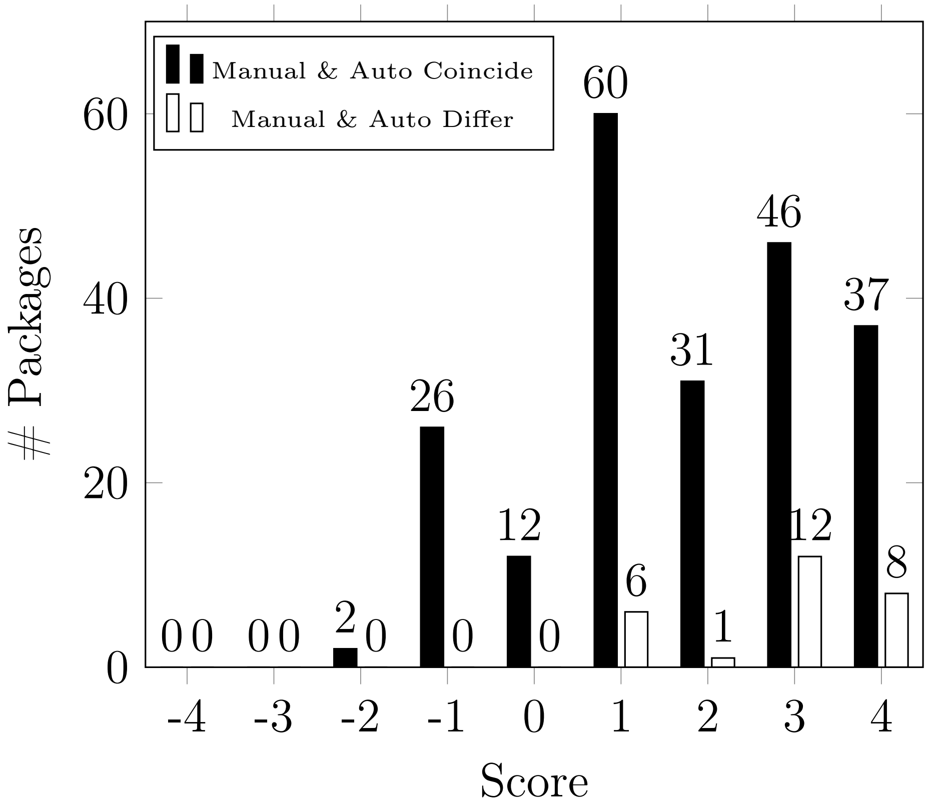
sources: it reports GitHub URLs for more packages but returns more incorrect URLs. Mode URL demonstrates better precision (+6%) and significant improvement in recall (+30%) compared to OFS. The *Final URL* allows us to improve the recall to 90% (+12%) while having slightly worse precision than the Mode URL (-3%).

Next, we interpret the influences of the automatic, reliable attribution metrics on the *Final URL*. Table 4.12 shows correlation analysis of the automatically computed reliable attribution metrics on the set of 325 packages. A low positive correlation (Pearson coefficient ranges from 0.15 to 0.36) between the metrics indicates that they independently contribute to the final reliability score of a URL. Furthermore, all of the p-values in Table 4.12 are less than 0.05 suggesting our results are statistically significant.

Then we compare reliability scores for URLs where the Final URL equals the Manual URL and where these URLs are different. Similar to the previous analysis, we excluded 29 packages whose source code is stored in the `https://github.com/googleapis/googleapis` and `https://github.com/twitter-archive/commons` mono repositories since they are treated differently in our study. Also, we have excluded 39 packages whose *Final URL* was empty, regardless of whether the manually selected URL was empty or not.

Figure 4.4 shows the distribution of the automatically calculated reliability scores for 257 packages. We observe that Final URLs corresponding to the manually identified ones tend to have a positive reliability score. In contrast, the URLs that differ from the manually identified ones tend to have a negative score. The Wilcoxon test ($p = 0.018$) confirmed that the mean values of these two data classes are not due to random error, but that is statistically significant. Our rigorous analysis of the URLs with negative reliability scores, where Final URLs are equal to Manual URLs, suggests that such packages often contain libraries written in languages different from Python (e.g., C++).

EXAMPLE 10. Even though GitHub repository [164] of `libsass` *[165] equals the package name (+1), the descriptions on Github and PyPI are entirely different (-1), there is no badge pointing to the PyPI package (0), and other programming languages are used in the repository, such as C++ and Shell (-1). As a result, the final score is -1.*



The histogram reports the automatic reliability scoring for 241 out of 325 packages that we manually analyzed. The white histogram shows the scoring distribution of the URLs in which the URL which was found by the program differs from the correct URL, which we manually identified. As the histogram shows, the scoring is a good indicator of the correctness of the outcome. The Wilcoxon test for the difference is significant at p=0.018.

Figure 4.4: Distribution of Reliability Score for *Final URLs*

Discussion. The proposed approach for finding GitHub URLs of a package by extracting and combining the information sources outperforms the state-of-the-art tool OFS. Moreover, combining the proposed method with OFS appropriately (e.g., ModeThenOFS) allows developers to extract Github URLs for many packages with high precision.

Our approach is the first to provide a set of reliable attribution metrics (Table 4.3) that allow developers to validate whether the automatically reported GitHub URLs (either by py2src, OFS, or similar package analysis tools) actually correspond to the expectations. Even though we cannot apply reliable attribution metrics for automatic judgment of the packages whose source code is stored in mono-repositories, the reported URLs are still helpful: it is much easier to find a subdirectory corresponding to a given package in a mono repository rather than to search for the source code location from scratch.

Summary: The proposed approach allows developers to automatically identify GitHub URLs for PyPI packages with 83% precision and 90% recall. Furthermore, the automatically identified URLs are easily verifiable due to the proposed automatically calculated reliability score based on metrics like the similarity between GitHub repository and PyPI package names and descriptions, the presence of a validating badge, and presence of Python code within the top programming languages in the identified GitHub repository.

4.5 RQ2.2: Factors that explain package adoption

To find the effect of quality factors on the adoption of PyPI packages (Table 4.13), we apply the Final URL approach to identify GitHub URLs for the 4000 top most downloaded packages. For the popularity factors, we used to consider the number of dependent packages and repositories from the service `libraries.io`. However, our manual examination for several packages reveals a significant mismatch between the data on provided by the service `libraries.io` [103] and the original GitHub repositories.[5] Moreover, once developers visit the GitHub repository, they are likely to extract all the possible information from one place rather than start checking additional sources. Therefore, we use the number of dependent repositories as reported by GitHub.

Starting from the 4000 packages, we exclude the following packages from our analysis:

- 498 packages whose URLs are empty, and therefore, no factors can be extracted;
- 257 packages as they point to the same GitHub repositories (mono-repositories), and therefore, cause duplicated values of factors.

The above filter results in 3245 packages, for which we automatically extract the

[5] For example, as of 14 May 2021 the number dependent repositories reported on the GitHub repository of `requests` library is around 910K, while `libraries.io` reports only around 152K dependent packages

Table 4.13: Descriptive statistics for the values of the factors
License factor is a dummy variable that has value 1, when the license is permissive (e.g., Apache-2.0 or MIT) and -1 when it is not permissive (e.g., GPL).

Factor	**Count**	**Mean**	**Std. dev.**	**Min**	**Q25%**	**Median**	**Q75%**	**Max**
Stars	925	2369	5085	2	207	640	2227	54 252
Contributors	925	74	121	2	17	34	75	1065
Open issues	925	85	190	0	10	27	76	1900
Closed issues	925	379	845	1	42	116	325	7880
Open/Closed issues	925	0.41	0.61	0	0.12	0.25	0.49	8
Commit frequency	925	20	45	0	3	6	18	490
Release frequency (days)	925	154	231	0	43	83	157	2420
Time to close issue (days)	925	92	94	0	28	63	121	666
Vulnerabilities	925	0.17	0.69	0	0	0	0	8
License	925	0.83	0.54	-1	1	1	1	1
Dependent repos	925	14 330	53 061	5	405	1360	5565	685 323
PyPI downloads	925	14.25M	52.27M	0.27M	0.51M	1.32M	5.17M	655.77M

quality and popularity factors. In some cases, PY2SRC returns an empty value for some factors (e.g., *contributors*, *dependent repos*). We verified the factors with empty values to ensure the correctness of the tool by manually checking the corresponding factors on the GitHub page.[6] Our analysis confirms the absence of the values in such cases, and therefore, we filter out 2304 packages that have at least a missing value in the factors.[7] We further remove 16 packages that have outlying values of their factors. Eventually, we end up with 925 packages for the regression analysis.

EXAMPLE 11. Package `urllib3` (the most popular at the time of research) had more than one billion PyPI downloads, 2603 stars, 31 commits per month on average, 52 days on average between two releases, 137 open issues, 757 closed issues, 21 days on average to close an issue, 236 contributors and 504 101 dependent repositories. Eight vulnerabilities affected the package, and it has the MIT license, which is permissive.

Table 4.13 shows the descriptive statistics of the factors for the packages in our study. We consider the quality factors (the first ten) as independent variables and the popularity factors (the last two) as dependent variables for the regression analysis.

We first perform a correlation analysis between the quality factors to check their independence. Table 4.14 shows Pearson correlation coefficients [7] for each pair of quality

[6] We manually look at the factors at given paths *`https://github.com/organization/repository/path_to_factor`*. For example, the path to *contributors* factor is `https://github.com/organization/repository/graphs/contributors`

[7] We could have assigned the zero values for the empty factors, but this might have introduced bias to the regression analysis.

Table 4.14: Quality factors correlations

The bold underlined values indicate the pairs of the strongly correlated factors. We exclude Contributors and Closed issues from further regression analysis.

	Stars	**Contrib.**	**Op. issues**	**Cl. issues**	**Op/Cl. iss.**	**Commit f.**	**Release f.**	**Ttc issue**	**Vulns**	**License**
Stars	**+1.00**	+0.64	+0.44	+0.58	-0.06	+0.36	-0.12	-0.22	+0.23	+0.02
Contrib.	-	**+1.00**	+0.66	**+0.85**	-0.14	+0.61	-0.24	-0.24	+0.38	+0.04
Op. issues	-	-	**+1.00**	**+0.74**	+0.05	+0.51	-0.13	-0.22	+0.14	+0.04
Cl. issues	-	-	-	**+1.00**	-0.14	+0.67	-0.17	-0.26	+0.23	+0.04
Op/Cl. iss.	-	-	-	-	**+1.00**	-0.10	+0.24	+0.13	-0.08	-0.04
Commit f.	-	-	-	-	-	**+1.00**	-0.20	-0.26	+0.17	+0.03
Release f.	-	-	-	-	-	-	**+1.00**	+0.29	-0.08	-0.01
Ttc issue	-	-	-	-	-	-	-	**+1.00**	-0.05	-0.03
Vulns	-	-	-	-	-	-	-	-	**+1.00**	+0.00
License	-	-	-	-	-	-	-	-	-	**+1.00**

factors. We observe that *Contributors* (Pearson coefficient 0.85) and *Open issues* (Pearson coefficient 0.74) have a strong positive correlation with *Closed issues* (e.g., Pearson coefficient greater than 0.7 [7]), so we exclude these factors from further regression analysis. The correlation analysis between the two popularity factors suggests that they are independent. Hence, we use eight quality factors (*Stars*, *Closed issues*, *Open/Closed issues*, *Commit frequency*, *Release frequency*, *Time to close issue*, *Vulnerabilities*, and *License*) as dependent variables and two popularity factors (number of *Dependent repos* and *PyPI downloads*) as dependent variables for the regression analysis.

We transform the factors into logarithmic scales to allow factors with extensive ranges for the regression analysis.[8] Table 4.15 and Table 4.16 shows the regression analysis summary for the factor *Dependent repositories*, and *PyPI downloads*, respectively.

Discussion. The number of *stars* has a statistically significant positive impact on the number of *PyPI downloads* and *Dependent repositories*. Hence, we confirm the qualitative finding that developers tend to adopt packages whose software repositories have a high number of *stars* [136].

Considering security in the regression analysis, we observe that the number of *Vulnerabilities* has a significant positive impact on both popularity factors. This observation might seem counter-intuitive: developers tend to adopt vulnerable packages. However, the vulnerabilities are likely to be discovered in popular packages simply because such packages gain much attention [8, 212]. Therefore, the choice of software dependencies might be affected by vulnerabilities in their projects but should be considered together with the other quality factors, e.g., the number of closed issues.

The number of *Closed issues* positively correlates with both popularity factors: we observe a significant correlation with the number of *PyPI downloads* and no significant

[8] To process factors that have zero values, we increase them by a small precision of 0.001

Table 4.15: Linear regression summary for number of *Dependent repositories*

Factor	**Number of Dependent repositories**		
	Coefficient	Confidence Interval	P-value
Constant	1.918	[+1.696, +0.140]	**0.000**
Stars	0.448	[+0.387, +0.509]	**0.000**
Closed issues	0.112	[+0.032, +0.192]	0.161
Open/Closed isssues.	-0.036	[-0.073, +0.001]	0.325
Commit frequency	-0.013	[-0.045, +0.019]	0.694
Release frequency	0.089	[+0.025, +0.153]	0.167
Time to close issues.	-0.040	[-0.090, +0.010]	0.426
Vulnerabilities	0.098	[+0.076, +0.120]	**0.000**
License	-0.014	[-0.036, +0.008]	0.545

Table 4.16: Linear regression summary for number of *PyPI downloads*

Factor	**Number of PyPI downloads**		
	Coefficient	Confidence Interval	P-value
Constant	6.478	[+6.285, +6.671]	**0.000**
Stars	0.067	[+0.013, +0.119]	0.210
Closed issues	0.1676	[+0.099, +0.237]	**0.016**
Open/Closed issues	0.018	[-0.014, +0.050]	0.575
Commit frequency	-0.0276	[-0.056, +0.000]	0.316
Release frequency	-0.144	[-0.200, -0.088]	**0.010**
Time to close issues	-0.041	[-0.085, +0.003]	0.351
Vulnerabilities	0.087	[+0.067, +0.107]	**0.000**
License	0.002	[+0.002, +0.040]	0.282

impact on the number of *Dependent repositories.* Thus, the intuition suggests that the projects with many closed issues are expected to be well-supported, i.e., developers might expect to receive support for their problems in the future.

Such intuition is also supported by the coefficients for the *Time to close an issue* factor. There is a negative correlation with both the number of *Dependent repositories* and *PyPI downloads*: the longer time of closing issues means that requests posted by software developers receive more extended responses. Hence, software developers have to wait longer to receive support for their problems.

When considering the fraction of open and closed issues, we observe a negative non-significant correlation with both popularity factors: while developers might not like many opened issues that do not receive any response, the high number of open issues indicates existing interests for a particular package. These observations support the developers' considerations reported by the qualitative studies [94, 136].

Commit frequency negatively impacts the number of both PyPI downloads and dependent repositories. This likely happens since the high commit frequency corresponds to the immaturity of a particular package. If adopted as a dependency, such a package might introduce bugs and security vulnerabilities. Also, many new versions of a package (possibly containing breaking changes) will likely be released often. Moreover, developers would have to upgrade their dependencies frequently. Several empirical studies report that developers tend to avoid such dependencies for their projects [91, 94, 134, 136]. However, the negative impact is not significant: high commit frequency might also mean good support for a library.

While such intuition is also supported by the negative statistically significant impact of the *release frequency* factor on the number of PyPI downloads, we observe a positive coefficient of the *release frequency* factor for the number of dependent repositories. We are likely to observe such an effect since package users prefer to stick with a specific version of a PyPI package. At the same time, developers have higher flexibility to declare a package without actually specifying its version in GitHub.

License is often an essential factor when adopting a software package [94, 136]. Interestingly, we observe a positive effect of license on the number of PyPI downloads and a negative impact on the number of dependent repositories. However, such effects are not significant, which might be affected by the fact that developers have a limited understanding of legal issues related to non-permissive licenses [191].

Summary: We found that the popularity metrics for PyPI packages are not correlated. Therefore, developers and researchers can use those quality factors when adopting or studying the adoption of third-party packages. Furthermore, through the regression analysis, we confirm that number of *stars*, *closed issues*, and *vulnerabilities* in a package increase the popularity of a package while the release frequency decreases it.

4.6 Threats to Validity

Threats to *internal validity* concerning the external factors not considered in our study:

We *ignore the mono repositories in our manual validation.* Mono repositories are usually used for code collaboration between different components in an organization [143]. However, in the PyPI ecosystem, small size and less complex packages hosted by a single repository are much more common [4]. Hence, we believe the exclusion of the mono repositories has minimal influence on the results of our analysis.

We *ignore 'forks' in our manual validation.* We manually differentiate the forks from the original source code repositories by using a set of measurements (e.g., number of stars or releases). For example, a fork may have fewer *stars* compared to the original one, or a *fork* does not contain the latest release that does exist in the package repository.

The vulnerability database used for our study may not cover all vulnerabilities that affect PyPI packages. Therefore, this study uses the manually curated Snyk vulnerability database that covers many publicly known vulnerabilities (e.g., the NVD database) and includes the vulnerabilities discovered manually by the Snyk security researchers [171]. Moreover, the developers have access to the same information about existing vulnerabilities. Hence, we believe that the vulnerability information used in this study is sufficiently large and reflects developers' awareness.

Threats to *external validity* concerning the generalization of results of the study:

Currently, we focus only on PyPI packages. We chose Python packages because it is the second most popular general-purpose language [126]. PyPI is the official registry of Python packages, forming an ecosystem comprising over 300 000 projects (as of April 2021). The analysis of package adoption factors can be enlarged to other language-based ecosystems, like npm or RubyGems. Although PY2SRC relies on the information in PyPI, researchers can also extend it to packages in the other ecosystems.

We *only consider repositories hosted on GitHub.* GitHub is the most popular web-based software repository for PyPI packages [31] and mentioned by our interviewed developers in Chapter 3 (Observation O5). However, it is trivial to extend PY2SRC to other web-based repositories, e.g., GitLab or Bitbucket, by tweaking our existing crawler and parser with specific characteristics of these platforms.

We *only considered the top-downloaded packages* hosted on PyPI out of more than 300 000 packages. Top popular packages usually have a tremendous adoption by other packages across the ecosystem [18, 221]. The package adoption percentage is closer to 100% when considering all PyPI packages. Hence, we believe that the reported results of our empirical study are representative.

Threats to *construct validity* concerning the accuracy of PY2SRC:

Some errors might be introduced during data collection as PY2SRC relies on GitHub APIs (with limited rate limit) and web scraping. However, three researchers carried out manual spot checks on the results of each crucial step to confirm that the tool operates correctly. Hence, we believe that this threat is minimal.

4.7 Conclusions

This chapter presented the automatic approach to identify GitHub URLs that correspond to PyPI packages and empirically validated the qualitatively identified factors [94, 136] that developers refer to while selecting new libraries for their projects.

The proposed information sources have enabled us to obtain more precise URLs compared to the state-of-the-art tool OFS. Furthermore, a set of reliability metrics provided by our approach facilitates verification of the reported URLs.

The regression analysis of the package quality factors showed that, for the actual selection of a new package, a developer could rely on the number of *Stars*, *Closed issues*, *Release frequency*, and the number of discovered and fixed vulnerabilities in the package.

To facilitate reproducibility, the PY2SRC tool and the data used in this paper are available at [168].

5

LastPyMile: Identifying the Discrepancy between Sources and Packages

"You can't trust code that you did not totally create yourself."

– Ken Thompson

Open-source packages have source code available on repositories for inspection (e.g., on GitHub), but developers use pre-built packages directly from the package repositories (such as npm for JavaScript, PyPI for Python, or RubyGems for Ruby).

Such convenient practice assumes that there are no discrepancies between source code and packages. Unfortunately, these differences pose both operational risks (e.g., making dependent projects unable to compile) and security risks (e.g., deploying malicious code during package installation) in the software supply chain.

Our empirical assessment of 2438 popular packages in PyPI with an analysis of around 10M lines of code shows several differences in the wild: modifications cannot be just attributed to malicious injections. Nevertheless, scanning again all and whole 'most likely good but modified' packages is hard to manage for FOSS downstream users.

We propose a methodology, LastPyMile, for identifying the differences between built artifacts of software packages and the respective source code repository. We show how it can be used to extend current package scanning practices for malware injection (which only covers less than 1% of the code of deployed packages).

5.1 Goal and Research Questions

One benefit of FOSS is that source code and additional metadata are publicly available for audit, review, and even modification. Developers rely on this information (e.g., number of GitHub stars, number of downloads from the service called LIBRARIES.IO) to decide whether to add a FOSS project as a software dependency into their projects [94, 136]. Organizations with high-security requirements, e.g., government organizations or vendors of commercial enterprise software, commonly establish vetting processes to ensure the quality and security of third-party software and services [40, 127]. In the case of FOSS, this evaluation is performed mainly by manual reviews and automated scans of the source code repository of each dependency [41].

In theory, once code is checked, developers could download software dependencies as source files in tarballs and build them in-house. However, this process can be time-consuming and requires knowledge of the build systems [81].

In practice, developers download pre-built packages from repositories (such as npm for JavaScript, PyPI for Python, or RubyGems for Ruby) under *the comfortable assumption that no discrepancies are introduced in the last mile between the source code and their respective packages*. However, such differences might be introduced by manual or automated build tools (e.g., metadata, Python bytecode files) [68] or for evil purposes. For example, a backdoor was inserted into the PyPI package `ssh-decorate` to collect the users' SSH credentials and exfiltrate them to a remote C&C server [35].

Reproducible builds in Subsection 2.7.2 could be a solution. However, for it to be practical, modifications need to be the exception rather than the norm. Unfortunately, the opposite is true on the field. Indeed, in the npm ecosystem, packages are not easily reproducible from the source code [68]. The same applies to the PyPI ecosystem (see Subsection 5.4 for more discussions).

In the absence of reproducible builds, a vetting process must be extended to cover the risk of malicious code injection in the last mile. Since applications have many direct and transitive dependencies, and because every new version must be verified, scalability and integration with existing security review pipelines are critical.

These requirements clash with the resources at hand for FOSS repositories: less than ten PyPI administrators oversee 400 000 package owners (See Subsection 2.2). At the time of writing, for every new upload, PyPI's vetting pipeline only checks a script called *setup.py* for malicious code that would execute upon package installation [204]. Although *setup.py* is commonly targeted by attackers, malicious code is also injected in other locations. Other approaches also require a significant effort to reduce false positives [48] and to improve the quality of hand-crafted signatures [128]. While suspicious packages or updates might be flagged, too many false alerts are generated for benign packages [118]. In the year 2020, the administrators had to evaluate 1874 new updates *per day*, with an average of 3500 files generated by more than 76 997 developers [31]. Thus, the cost of even a single false positive in the evaluation must be multiplied by those numbers.

A key observation is that in code injection attacks, only a minimal part of the codebase is modified [200]. Therefore, one could focus on the last mile differences between the source code and the submitted packages. Hence our first question:

RQ3.1: *Can we effectively and efficiently identify differences?*

A basic solution already exists: GIT LOG. For each line in an artifact, we check whether it is (or at least was) in the repository at some point. By iterating over all commits (revisions), we ensure that we collect everything in the source code repository, and we eliminate the need for identifying the pair of Git release/tag and PyPI release to be compared. Unfortunately, that does not scale as GIT LOG needs to loop over all revisions and spawns a heavy git process each time it is invoked. We could also use diffing techniques [2,68], but they require a mapping of each PyPI release onto the corresponding Git tag or release, which does always exist.

Our algorithm LastPyMile is a feasible alternative to this problem. By cleverly combining package scraping and artifact hashing, we can extract these differences in a scalable way. Then, we can analyze how big is the gap in the field:

RQ3.2: *How big are the 'normal' differences between the code in source control (Github) and package (PyPI) repositories?*

We show that such differences are pervasive for more than 2000 popular packages in the PyPI ecosystem. If a package code differs from the published source code, one cannot assume that it has been maliciously modified. Differences are too many (65% of artifacts and 22% of files in our sample) and too diverse for reproducible builds to be a solution. However, only a few modifications happen in Python source files (2.6% of files), so that vetting might be a feasible alternative.

Finally, we can try to determine whether this solution can make a difference on the end goal: improving the vetting and coverage of scanners while keeping the number of false alerts manageable for PyPI maintainers given the imbalance ratio between the PyPI maintainers and the number of packages [201].

RQ3.3: *Can* LastPyMile *be combined with package scanners while keeping the number of alerts manageable by a human?*

To be effective in the field, we should allow developers and development organizations to use the same tools to scan the source code repository of a package as part of their vet-

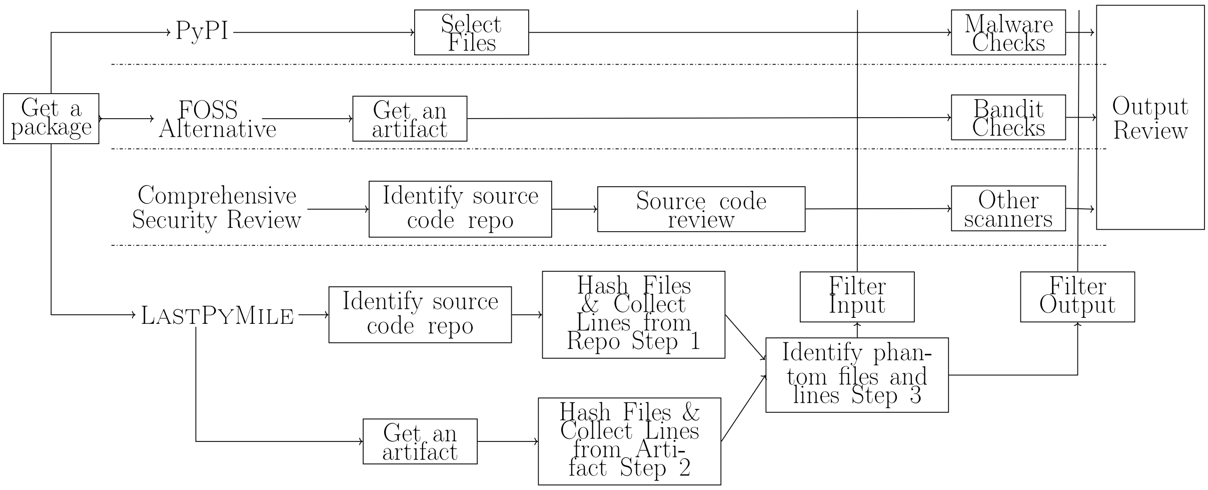

Get a package
PyPI
Select Files
FOSS Alternative
Get an artifact
Comprehensive Security Review
Identify source code repo
Source code review
LastPyMile
Identify source code repo
Hash Files & Collect Lines from Repo Step 1
Get an artifact
Hash Files & Collect Lines from Artifact Step 2
Identify phantom files and lines Step 3
Filter Input
Filter Output
Malware Checks
Bandit Checks
Other scanners
Output Review

Figure 5.1: LastPyMile in the context of the overall security review pipeline

ting process. Without protecting their investments in licenses, workflows, and developer education, an excellent technical solution would be doomed to fail. We show that such an approach is possible with LastPyMile .

5.2 RQ3.1: LastPyMile to improve the efficiency of identifying code injection

The upper part of Figure 5.1 shows the typical process of the security review process of package repositories (e.g., PyPI) for identifying suspicious artifacts that might occur during the release of a software project. First, the code in the published artifact is undergoing code review and scanning by the PyPI Administrators by running security checks [204]. Currently, they are using two checks called SetupPatternCheck and PackageTurnoverCheck (see Subsection 5.5).

Depending on the automated tool used by the maintainers, this scanning could be done on the entire artifact for backdoor injection (e.g., Bandit[144]) or on its files (e.g., Malware Checks [204]). Then the output of the scan is used to decide whether the artifact should be uploaded to the package repository.

The bottom part of Figure 5.1 shows how LastPyMile can augment the traditional security process. As a preliminary, LastPyMile looks for the GitHub URLs of a PyPI package in various places, including package metadata, PyPI, and package homepage. Table 5.1 shows the number of GitHub URLs we found. Most of the packages declare their GitHub repositories in the metadata available on PyPI.

Table 5.1: Number of source code repositories found by locations
Metadata of a package contains multiple fields such as Homepage, Codepage. Package Homepage is the main page that contains additional information about a package (e.g., documentation).

Location	**#GitHub Repos**	**Percentage (%)**
Homepage (Metadata)	2618	77.9
Codepage (Metadata)	68	2
Package Homepage	1418	42.2
PyPI Homepage	1974	58.7
Total GitHub Repos	3662	100

Step 1 Hashing files and lines from source code repository

```
Require: The Github URL of the package: $GithubURL$
Set of file hashes in the repository $H_s$: []
Set of lines of files in the repository $L_s$: []
$Cloned\_Dir = CloneRepositoryToDisk(GithubURL)$
$Commits = GetCommitsFromRepo(Cloned\_Dir)$
for each $c \in Commits$ do
    $F_s \leftarrow CheckoutFilesInCommit(c)$
    for each $f \in F_s$ do
        $H \leftarrow H \cup SHA256(f)$
        $L \leftarrow L \cup ReadFile(f)$
return Set of file hashes, lines: $H_s$, $L_s$
```

In Step 1, LASTPYMILE iterate all commits to compute all file hashes and collect line contents from a source code repository. To ensure that all the files and lines are collected, LASTPYMILE processes commits from all branches and tags in the GitHub repository. LASTPYMILE supports processing the GitHub repository in parallel so that multiple commits can be processed simultaneously. Besides, to avoid processing the same commits in different branches, LASTPYMILE maintains a shared set of already processed commits for synchronizing the processing tasks.

EXAMPLE 12. 18 distributed artifacts `nameko-3.0.0.rcX` contain the source code that is stored in the `v3.0.0-rc` branch.

After collecting all the file hashes and lines from the Github repository, in Step 2, LASTPYMILE processes an artifact to calculate file hashes and collect file lines. Finally, LASTPYMILE compares file hashes and lines of distributed artifacts and those in the source code repository to report the phantom files and lines (Step 3).

Step 2 Hashing files and lines from an artifact

```
Require: $A$, the PyPI package artifact to be evaluated
Set of file hashes in an artifact $H_p$: []
Set of lines of files in an artifact $L_p$: []
Artifact URLs: $A_s = ObtainArtifactURLs(p)$
$Local\_Artifact \leftarrow DownloadArtifactFromPyPI(A)$
$F_s \leftarrow UncompressArtifact(Local\_Artifact)$
for each $f \in F_s$ do
    $H \leftarrow H \cup SHA256(f)$
    $L_s \leftarrow ReadLinesFromFile(f)$
    for each $l \in L_s$ do
        $L_p \leftarrow L \cup l$
return Set of file hashes, lines: $H_p$, $L_p$
```

Step 3 Identifying phantom files and lines in distributed artifacts

```
Require: $H_s, L_s, H_p, L_p$
Set of phantom files: $H_d$: []
Set of phantom lines: $L_d$: []
for each $h \in H_p$ do
    if $h \notin H_s$ then
        $H_d \leftarrow H_d \cup h$
        for each $l \in L_p$ do
            if $l \notin L_s$ then
                $L_d \leftarrow L_d \cup l$
            end if
    end if
return Set of phantom files hashes, lines: $H_d$, $L_d$
```

Table 5.2: Running time comparison between LastPyMile and git log approaches
Both approaches had been run in the same environment. The differences obtained by both the approaches are the same (e.g., number of phantom files and lines).

Package	**git log (seconds)**	**LastPyMile (seconds)**
certifi	1244	48
idna	408	34
six	315	145
s3transfer	1095	44

Table 5.3: Number of unique phantom files and lines versus total
The columns on the left are the files and lines processed by the PyPI Malware Checks *and existing scanning tools, while* LastPyMile *only processes the phantom files and lines on the right. Phantom files are counted by their unique hashes*

	#Total		**#Different**	
	setup.py	All	*setup.py*	All
#Files	4056	90 143	2532	16 170
#Lines	38 750	14 027 895	7236	939 772

LastPyMile takes only 0.04 seconds for scanning `jellyfish` artifact that consists of 530 unique files and 28 104 lines on a laptop with four CPU cores and eight GB RAM. Considering the top four most downloaded packages `six`, `idna`, `python-certifi` and `s3transfer` as shown in Table 5.2, LastPyMile is 16x faster than the default iterative approach that relies on calling git log command for every line of an artifact because LastPyMile preproceses all commits in a repository and require only a single pass over all code, while git log have to iterate over all revisions each time it is invoked.

Table 5.3 compares the number of total files and lines present in the analyzed files with the phantom files and lines reported by LastPyMile. We observe that more than half of *setup.py* files are phantom, while the number of phantom lines of code in the *setup.py* files is six times smaller than the total number of lines in *setup.py* files.

Globally the number of phantom lines of code is 16 times smaller. Table 5.4 shows that a median artifact contains two phantom lines that include at least one API call (e.g., a line of code executes some API calls) and two lines that import some library.

Summary: LastPyMile enables checking the entire codebase of a published artifact 16x faster than the baseline git log approach, as LastPyMile requires only a single pass over all commits.

Table 5.4: Descriptive statistics about phantom lines in the artifacts

	Mean	Min	Q25%	Median	Q75%	Max
#APIs	4	1	1	2	3	946
#Imports	2	1	1	2	3	12

Table 5.5: Descriptive statistics of GitHub repositories for the selected packages
Tags includes Github tags and branches of a Github repository. Unique files and lines are determined by their hashes and contents, respectively.

Number of	Mean	Min	Q25%	Median	Q75%	Max
Tags	29	1	9	19	36	678
Commits	477	2	91	232	548	10 730
Unique files	97	3	14	29	68	17 000
Unique lines	53	1	6	17	43	8732

5.3 Data collection

To select the sample of Python packages for our study, we start with the list of the top 4000 most downloaded packages [192], which is the established approach to study the Python ecosystem, adopted both in academia [31] and industry [123] and [45].

We identify 3662 packages (more than 91% of the selected PyPI packages) that use GitHub to host their source code. Among these packages, 3336 are unique repositories (83%). For simplicity, here we focus only on packages that claim their source code is on GitHub. Table 5.5 shows the characteristics of the collected repositories. Three repositories have only two commits[1], while several repositories had tens of thousands of commits (e.g., the Github project `pip` has 10 730 commits[2]).

As we aimed to have a tool to be runnable "as you wait" [159], we set a timeout period of five minutes for analyzing all artifacts of a given package. As a result, the selected packages resulted in 109 062 artifacts. We had to exclude 15 810 artifacts (14%) belonging to 'surviving' packages with early versions being developed on versioning control systems other than Git and/or with the commit history not being included when moving to GitHub. Therefore, we could not use them in our analysis as there was no source code to compare. The final dataset comprises 93 252 artifacts from 2438 packages, 65% of them are gzip, 29% are *wheel*, 4% are zip, and 2% are eggs.

After checking the differences between the number of different files and code lines be-

[1] For example, `https://github.com/datamade/probableparsing`

[2] At the time of data collection

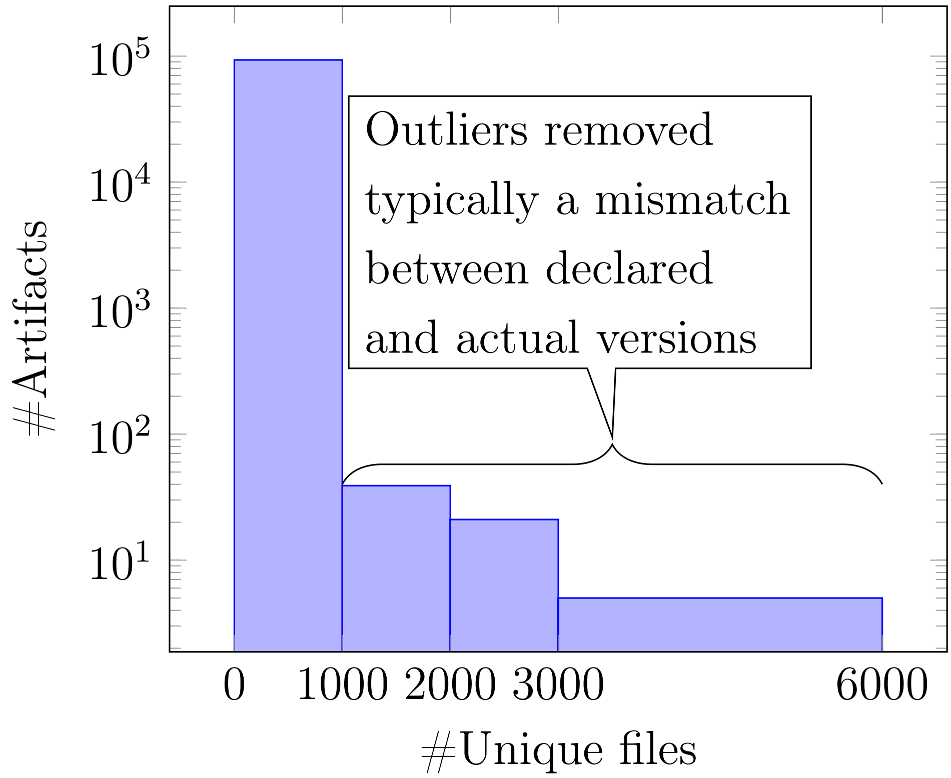


Figure 5.2: Number of files differing between source and package

tween source and package repositories (Figures 5.2 and 5.3, respectively), we observed that *66* artifacts featured a vast number of changes (>1000 different files). We manually analyzed those artifacts and found that the explanation lies in "developers moving stuff around" across repositories, making it nearly impossible to identify source code repositories automatically. The case in Figure 2.3 requires one to actually read the documentation.

Besides the `robotframework-selenium2library` in Example 2.3, we found that the package `sas7bdat` first hosted its source code on GitHub but then was moved to BitBucket. The other reason for not being able to locate the corresponding source code of a package automatically is the usage of submodules [208] by developers. We removed such artifacts from our analysis as their source code could not be found automatically. Hence, the final list of analyzed artifacts comprises 93 252 artifacts. Table 5.6 summarises the number of analyzed packages and corresponding artifacts.

EXAMPLE 13. The `gsutil` *package refers to the Github repository located at the URL* `https://github.com/GoogleCloudPlatform/gsutil` *with two submodules. We could not find any related GitHub tag/release for* `Pyrogram-0.8.0-py3` *and* `Pyrogram-1.0.3`*.*[3]

[3] Our manual analysis of these packages did not reveal malicious injections.

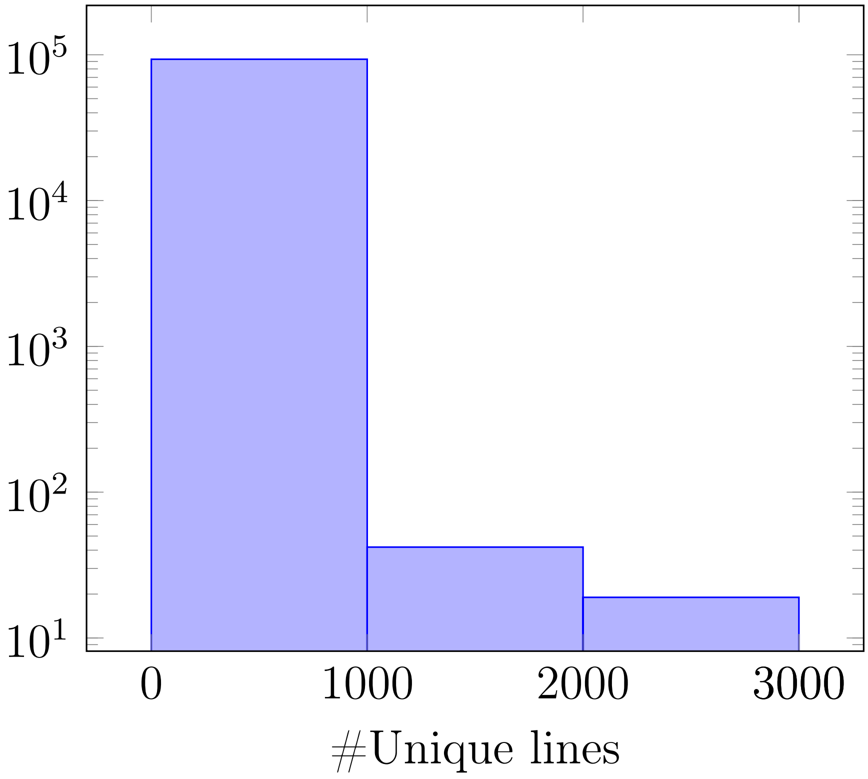


Figure 5.3: Number of lines differing between source and package

Table 5.6: Number of processed packages and artifacts
The processing time threshold is set for a package. We exclude artifacts that predate the creation time of a Github repository

Data Processing Step	**Result**
Top-most downloaded packages	4000
Number of processed artifacts (processing time < 5 minutes)	109 062
Number of artifacts in GitHub (excluded very early artifacts)	15 810
Number of final artifacts (automatically linkable to source)	93 252

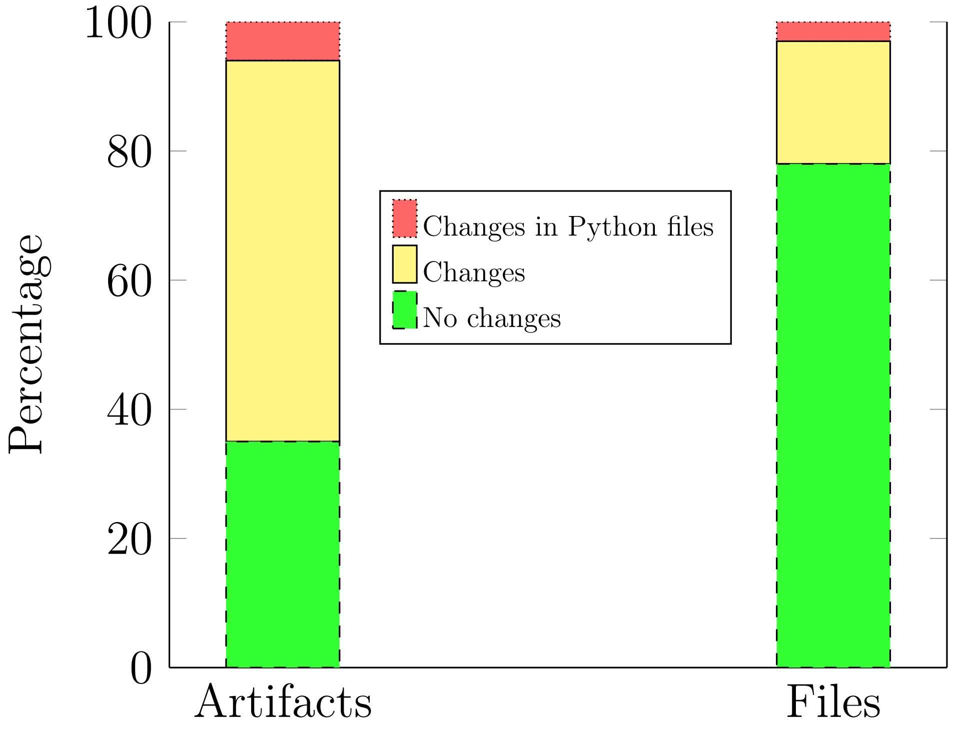


Figure 5.4: Percentage of different kinds of changes in artifacts and files

5.4 RQ3.2: Differences between source code and package repositories

To answer RQ3.2, first, we compared the code distributed in PyPI artifacts with the corresponding source code repositories. Figure 5.4 shows that 65% of artifacts and 22% of files present in PyPI have differences with respect to the source code repository. For example, malicious code might be injected during the package release process. However, only 5.8% of artifacts and 2.6% of files have changes in Python files, while 59% of artifacts and 19% of files have differences in other files. These findings suggest that it might be promising to limit the process checking for malicious injections to those artifacts and files that have discrepancies, as the other artifacts cannot have malicious injections during the release process. In this work, we focus on the changed Python files as they might be the target of attackers for injecting executable malicious commands.

Metadata files greatly impact the number of discrepancies between source code and package artifacts. Table 5.7 shows that a median artifact has four metadata files[4] and

[4] We identified metadata files as generated by packaging tools (e.g., *WHEEL*), dependency declaration files (e.g., *requirements.txt*), and documentation files (e.g., *README.md*)

Table 5.7: Differences between package artifacts and their source code repositories
Unique files are the files having different hashes while the number of lines are the total number of lines in an artifact

#Files	Mean	Min	Q25%	Median	Q75%	Max
Number of Unique Phantom Files						
Python	9	1	1	1	6	994
Metadata	4	1	3	4	5	19
Number of Phantom Lines						
Python	19	2	2	4	12	1988
Metadata	8	2	6	8	10	38

Table 5.8: Top different phantom Python files in our sample.
Phantom files are present in the package source code but have different content than the omonimous file in the source code repository. The same file name might occur multiple times in the same package with different paths. __init.py__ and setup.py are the most common phantom files.

Filename	Number of Phantom files	Percentage (%)
__init__.py	36 480	14.5
setup.py	7414	3
_version.py	4152	1.7
version.py	3260	1.3
utils.py	2354	1
v1.py	1498	0.6
v2.py	1498	0.6
base.py	1404	0.6
client.py	1050	0.4
exceptions.py	1008	0.4

nine Python files (twice more). This difference is also visible at code line level: a median artifact has 2-8 lines and 18 lines in phantom metadata files and Python files, respectively.

We observe that nearly 15% of Python files that have differences with respect to the source code repository are the initialization *__init__.py* and installation *setup.py* files (Table 5.8). This happens most likely because the building tools introduce additional information (e.g., timestamps, versions) into these files during the packaging process. Similarly, the *_version.py* and *version.py* files are used to automatically identify the package version from a Git tag or release.

Table 5.9 shows the top ten regular and API calls related to networking and system in the Python files that differ from the source code repository. Many files have calls to

Table 5.9: Top ten API and Sensitive calls in modified Python files
API calls are grepped from the line contents using a set of regular expressions. We exclude some internal calls of the packages.

Top API Calls	Occurences	Networking & System Calls	Occurences
__init__	72 413	urlopen	793
isinstance	55 115	socket.socket	711
datetime	37 393	subprocess.Popen	670
ttinfo	37 258	exec	580
len	36 325	request	541
read	31 582	http.request	511
getattr	21 575	s.setsockopt	413
super	16 760	requests.post	323
hasattr	16 358	request.get	317
join	13 869	os.chmod	303
append	12 548	platform.system	292

such functions as *urlopen*, *socket.socket*, *request* to open URLs and make HTTP requests, *subprocess.Popen* and *exec* to open files. Usage of these functions could be harmful. At the same time, these functions are often used for legitimate operations, and one cannot simply mark all lines that include a call to “possibly suspicious” APIs as “actually suspicious” – there would be an unmanageable number of false alerts.

Summary: The code distributed via package repositories has many changes with respect to the code stored in the corresponding source code repository. On average there are 5.8% of artifacts and 2.6% of files have changes in Python files.

5.5 RQ3.3: LastPyMile combined with other package scanners

Combining LastPyMile with existing security scanners is essential for two reasons: First, it allows the reuse of mature detection techniques of FOSS and commercial security scan tools. Second, by doing so, developers and development organizations can use the very same tools in different stages of the security review process, which protects their investments into software licenses, the design and implementation of review workflows, and developer education.

As shown in Figure 5.1, PyPI Administrators can achieve the reuse by filtering either the input or the output of such security scanners. They can feed package scanning tools operating on single files (Malware Checks), modules, or procedures (Bandit Checks) with input containing *phantom lines*, which is expected to reduce the number of irrelevant alerts and the tool's runtime. Scanning tools performing the whole program or inter-procedural analyses continue to work on the package's entire code base. Still, their output can be filtered to only show findings in phantom lines.

In this experiment, we focus on input filtering and show the results of combining LastPyMile with two well-known malware checking tools that are broadly used in the PyPI ecosystem:

- **Warehouse Malware Checks** [204] tool is used by PyPI to check suspicious code in uploaded packages. At the time of writing, the tool supports two checks: SetupPatternCheck [206] for performing regular-expression based checks of the content of *setup.py* files on package upload and PackageTurnoverCheck [205] for performing daily scans for suspicious behavior about package ownership changes. Conceptually Malware Checks is close to other open-source tools for auditing FOSS packages [117, 118] that rely on regular expressions to the whole artifact.
- **Bandit** [144] is the tool developed by the Python Code Quality Authority. Bandit was designed to find common security issues in Python code by scanning all the files included in a software artifact. For each file in the artifact, the tool creates an abstract syntax tree (AST) representation and performs rule-based analysis (plugins) of the AST nodes. Most of the Bandit rules focus on the vulnerabilities in Python code (e.g., start a process with a function vulnerable to shell injection).

For the Malware Check tool, we focus on the SetupPatternCheck check. Even though the tool currently scan only *setup.py* files, we have extended it to scan all Python files of a package artifact.

For the Bandit tool, we have used the default set of Bandit rules and then extended them with additional rules so that the tool can find all malicious lines of code injected into Python packages known to be used in typosquatting/combosquatting attacks [129, 201]. In addition, our ruleset checks for suspicious API calls (e.g., *exec*), imports (e.g., *socket*), and strings (e.g., an URL). We have open-sourced the detection rules on Github at [196].

To illustrate how the malware checking tools perform on the package artifacts without malicious payloads, we compare their outcome on three example benign artifacts that correspond to the following malicious artifacts. We collected the following malicious artifacts from the real attacks by contacting the researchers who reported the attacks:

- `urlib3-1.21.1` – The malicious code was injected into the *setup.py* file. It triggered automatic extraction of data and sending it to a remote server using the standard library `socket` [55].

Table 5.10: LastPyMile on Malware Checks and Bandit alerts
Malware Checks Alerts (X rules on Lines), while Suspicious Bandit Alerts (Y rules on Files). The setup.py column of Malware Checks Alerts is what happens now in PyPI [204].

Artifact	**Type**	In setup.py	**Problem Size**				**Malware Checks Alerts**			**Suspicious Bandit Alerts**		
			#Files	#LoCs (all files)	#LoCs (setup.py)	Coverage (setup.py)	setup.py	whole pkg	LastPy Mile	setup.py	whole pkg	LastPy Mile
urllib3-1.26.3	Benign		80	25 348	97	0,4%	1	260	0	8	1398	0
requests-2.25.1			32	9325	112	1,2%	3	57	0	9	505	0
setuptools-53.0.0			244	70 794	162	0,2%	4	2932	0	5	762	0
urlib3-1.21.1	Malicious	Y	72	20 448	197	1%	4	177	3	20	1044	12
request-1.0.117		N	3	166	52	31,3%	2	8	2	5	27	20
setup-tools-36.0.1		Y	112	31 245	304	1%	8	1289	3	21	489	12

- `request-1.0.117` – while the installation *setup.py* file contains the code to trigger the malicious execution from the *hmatch.py* file, the actual malicious functionality was implemented in the *hmatch.py* file: scanning the computer network and sending results to the remote server using the `urllib3` library [188].
- `setup-tools-36.0.1` – the malicious code was injected into the *setup.py* file triggering automatic extraction of sensitive data and sending it to the remote server via the Python built-in `socket` library.

Table 5.10 shows the results of Malware Checks and Bandit tools' scans of the selected artifacts. Since the Malware Checks tool was primarily designed to scan only *setup.py* files, we report the number of findings the tools produced on the *setup.py* file. Then we present the number of alerts when we run the tools on the whole package. Finally, we show the number of alerts the tools produced on the lines of phantom code as reported by LastPyMile. The source code of LastPyMile is available on Github at [197]. The replication package for Table 5.10 can be obtained at [203].

We observe that Malware Checks produced at most three alerts on each of the benign and malicious artifacts when only the *setup.py* file was considered. While this amount of alerts is manageable by humans, checking only the *setup.py* files allows one to have coverage of around 1% of the total code base of the analyzed artifacts, except the malicious request artifact where scanning *setup.py* has generated coverage of 31.3%.

When Malware Checks was executed on all files from the package, the number of alerts rockets to 2-3 orders of magnitude. Notably, the tool produced more alerts on the benign artifacts than on the malicious packages. This phenomenon corresponds to the more extensive code base of the legitimate artifacts.

We observe similar behavior of the Bandit tool. When applied on the *setup.py*, the tool generated alerts both on benign and malicious artifacts. However, Bandit produced significantly more alerts on the malicious artifacts. We observe many alerts when looking at the alerts generated after running the tool on the entire package. Notably, looking only at the number of alerts, one could not distinguish between benign and malicious artifacts: the number of alerts produced on the benign artifacts exceeds the number of alerts on the malicious artifacts.

After applying LastPyMile to the tool results after running them on the entire artifacts, we observe a significant reduction of the number of alerts for both tools. For example, Bandit tool produced only 12 alerts (out of 1044) after applying LastPyMile on the results of the `urlib3-1.21.1` scan. Similarly, the number of alerts produced by Malware Checks on the `setup-tools-36.0.1` reduced to 12 instead of 489. Furthermore, looking at the outcome of the benign packages, we observe that LastPyMile reduced the number of alerts to zero.

Being applied to *setup.py* files only, Malware Checks tool generates several alerts manageable by humans. However, scanning only *setup.py* files does not guarantee the artifact to be free from malicious code as 99% of the codebase is not checked. Furthermore, the number of alerts that both tools produce after scanning the entire artifacts (3249 and 2665 false alerts for Malware Checks and Bandit respectively) demonstrates that such analysis does not scale for an 'on-upload' analysis by PyPI maintainers.

In contrast, LastPyMile shows an excellent potential to improve the scanning results. First, it makes the number of alerts after running a tool on the entire artifact comparable with the current number of alerts generated by the Malware Checks. Second, we do not observe any alerts for benign artifacts, which allows us to easily distinguish benign and malicious artifacts in our manual validation of the alerts.

When running on all malicious code packages in our dataset, we were able to preserve all malicious alerts and did not introduce false positives over the current scanning process.

Those properties make LastPyMile a good candidate for software vetting processes of government organizations or other OSS consumers with high-security requirements. In addition, the review effort is manageable, even though typical development projects have dozens of dependencies with more or less frequent release and patch cycles.

Summary: LastPyMile reduces the number of alerts produced by a malware checking tool to a number that a human can check. We checked our approach against known malicious packages, and we found that LastPyMile can detect all of them. Also, it removes all the alerts from benign packages, allowing a clear distinction between benign and malicious packages.

5.6 Threats to Validity

The validity of the results reported in this chapter is impacted by several choices made during tool and experiment design.

We only consider repositories hosted on GitHub In our dataset, there are 56 packages hosted in BitBucket [28], 14 packages hosted in Gitlab, 13 packages hosted in the SourceForge [175], 19 packages are hosted in Google Code [37], and four of them had been moved to GitHub [63]. In addition, Github is the main platform reported by our interviewed developers in Chapter 3 (Observation O5). We believe that there are no significant obstacles to cover other version control systems and extend the current implementation to other Git service providers (e.g., GitLab or Bitbucket) as long as they support code commits (e.g., Apache Subversion).

We only consider the original repository of a package. The main reason for this decision is that a fork may not contain the new (recently) releases that exist in the package repository. Therefore, comparing the artifacts in the package repository and the forks of the original repository will clearly lead to mismatches that we could have avoided.

The current implementation focuses on the Python packages in PyPI and Python files in particular. The extension to other Python ecosystems (e.g., AnaConda) [10], and interpreted languages and other file types seem straightforward (e.g., Node.js/npm [125] and Ruby/RubyGems [157]). However, we only considered the top 4000 packages hosted on PyPI out of more than 250 000 packages. A more significant number of packages would need to be considered for an ecosystem analysis.

By design, LastPyMile *checks only the code absent from source repositories* even though malicious code could also be included in the versioned code, either directly or in tests. For example, this was the case of the `Pillow` Python framework that was flagged by more than 15 Antivirus vendors [152]. However, this situation lays out of the scope of the chapter, as the test files should have been spotted during the source code review.

We limit the line-by-line analysis to files with file extension .py. The main reason driving this design decision is to focus attention on files whose discrepancies, compared to what users can view in the respective source code repository, can alter the program flow (e.g., when downstream users install an artifact in their development environment or invoke its API as part of their development project).

Other phantom files might be also be used to inject malware. For example, the phantom files under the test directories are required by a popular testing framework like pytest. Another source of phantom files is the upload of modules specific to the developers' development environment. Unfortunately, they are usually not versioned with

Git.

EXAMPLE 14. The phantom files in `pydruid-0.5.4.tar.gz` *are the manually built Python packages stored in the site-packages directory. We can verify the origin of the locally installed modules by comparing their code files with the corresponding GitHub repository. We can use* LastPyMile *to check the code files in the local module called* `traitlets` *(e.g.,* `https://github.com/ipython/traitlets`*) of* `pydruid-0.5.4.tar.gz` *belong to the corresponding repository [83].*

Moreover, PyPI packages contain other executable files, e.g., Windows portable executables, OSX disk image files, or C/C++ static libraries. For example, we found many Python bytecode files (ending with *.pyc*). These files should not be uploaded to PyPI as this can make the dependent package (e.g., a Debian package) fail to compile [67]. Investigating these cases would require a distinct paper.

We only check additions of code lines in the present version, even though a vulnerability could be introduced by deleting lines from a software artifact (e.g., removing a sanitizing statement). Albeit LastPyMile does not report the deleted lines in such a case, it could detect that the files in the uploaded artifacts are different as their hashes would differ if compared to the hashes of the files stored in the corresponding source code repository. Limiting the false alerts, in this case, would require special care to avoid that the whole file being reported as different. Therefore, we leave this case for future work.

Some packages contain code automatically generated by tools like Swagger Codegen *or Python* distutils. The current implementation of LastPyMile would cause conceptually false positives as such files do not conceptually differ from phantom files. These cases of automatically generated files could be checked by applying the same code generation tool on the code files in the Github repository and comparing with the files in published artifacts.

5.7 Conclusions

We investigated the discrepancies between published artifacts and source code repositories to understand the risk of malicious injections during the software release process. Our empirical analysis of 2438 most downloaded PyPI packages shows that there exist differences between packages in PyPI and the corresponding source code repositories at different levels of granularity (artifacts, files, and lines). The differences are attributed to developers and automated tools (e.g., packaging tools) and could impact the consumers, e.g., causing compilation issues or representing a potential for containing malicious code injections.

The flexible combination of LastPyMile (as input/output filter) with other security tools offers the possibility to reduce the number of findings and the time required by vetting processes. For example, we instructed Malware Checks and Bandit to only consider phantom code as input. The resulting decrease in false alerts makes it promising to use LastPyMile as an additional check in the PyPI vetting processes with minimal impact on review efforts.

The source code LastPyMile is available on Github at [197]. The replication package is available at [203], and we plan to submit LastPyMile as a new check to PyPI [145].

6

Please hold on: more time = more patches? Automated program repair as anytime algorithms

> *"Much of the effort in modern programming goes into the testing and repair bugs."*
>
> – Fred Brooks

Automated program repair (APR) techniqus were designed to generate bug-fixing patches for software bugs automatically. The first APR tool called EXP proposed by Stumptner and Wotawa [178]. Typically, APR tools accept a buggy program and a set of test cases, and output a patch that passes all the test cases. Current evaluations of automatic program repair (APR) techniques focus on tools' effectiveness, while little is known about the practical aspects of using APR tools, such as how long one should wait for a tool to generate a bug fix. This chapter empirically studies whether APR tools are any time algorithms (e.g., the more time they have, the more fixes they generate, so it makes sense to trade off longer time for better quality). Our preliminary experiment shows that the amount of plausible patches, given exponentially greater time, only increases linearly or not at all.

6.1 Goal and Research Question

Automated program repair (APR) techniques were designed to automatically search for (i.e., generate) candidate patches for software bugs that might be plausible ones (passing all tests) and then possibly correct ones (actually fixing the bug). An essential class of search-based techniques is that of *anytime algorithms* [220], which tries to capture a necessary trade-off:

> *Anytime algorithms give intelligent systems the capability to trade deliberation time for quality of results. This capability is essential [when] it is not feasible (computationally) or desirable (economically) to compute the optimal answer [. . .].*

Our long-term **Goal** in this chapter is to understand whether APR techniques are anytime algorithms so that it makes sense to wait longer to obtain better results. An ideal tool should be efficient enough in providing results (reports or patches) to developers or other elements in the software development chain (See the discussion of Observation 16).

Empirically, Durieux et al. [49] studied 11 APR tools but only showed that most of the repair attempts resulted in an error or terminated with a timeout. By constraining the search space, Qi et al. [154] showed that the number of patch candidates can vary significantly, which might impact the number of plausible patches. In this direction, Martinez and Monperrus [113] showed that an additional time budget might result in a higher number of plausible patches. However, the authors did not investigate *how long one needs to wait.* To the best of our knowledge, we do not find a study investigating the impact of different time budgets.

In this chapter, we report the answer to the question:

RQ4.1: *By doubling the time budget of an APR tool, do we get twice more plausible patches?*

We follow the methodology used to evaluate the *anytime algorithms* [77] to compare the number of patches generated by five open-source APR tools on a set of 5 benchmarks that are commonly used in APR community for evaluation. For instance, the benchmark Defects4J [86] was used in 22 of 24 papers studing repair tools for Java [49]. Our preliminary results show that having exponentially more time, APR techniques produce only a linear or no increase in plausible patches and do not seem to have the trade-off ability for being anytime algorithms.

6.2 Benchmarks for Automated Repair of Java Programs

Different benchmarks are used to evaluate the performance of APR tools. The datasets typically contain the three following elements:

- Buggy programs
- Passing test cases
- Failing test cases

Besides, a benchmark should provide information about how to compile the bug, locations of test cases. Below are the common benchmarks used by APR community.

Defects4J [86] contains 835 bugs from 17 open-source Java projects. The bugs were mined with the support of bug tracking systems. The dataset was initially proposed to the software testing community, and it has been extensively used in program repair studies. The framework can be found on Github at `https://github.com/rjust/defects4j`.

Bugs.jar [160] contains 1158 bugs from eight Apache projects. The dataset was created using the same strategy than Defects4J, but contains high number of bugs. The dataset can be found on Github at `https://github.com/bugs-dot-jar/bugs-dot-jar`.

Bears [110] contains 251 bugs from 72 different GitHub projects with an average size of 62 597 lines of code. It was created by mining software repositories based on commit building state from Travis Continuous Integration. Bears has the largest diversity of projects compared to previous benchmarks of bugs. The dataset can be found on Github at `https://github.com/bears-bugs/bears-benchmark`.

IntroClassJava [51] contains 297 bugs from six different student projects. It is a transpiled version to Java of the bugs from the C benchmark IntroClass [99]. In the transpiled version, the projects have on average 230 lines of code. The dataset can be found on Github at `https://github.com/Spirals-Team/IntroClassJava`.

QuixBugs [104] consists of 40 Java and Python programs from the Quixey Challenge. Each contains a one-line defect, with passing (when possible) and failing testcases. Each program corresponds to the implementation of one algorithm such as QuickSort, and contains on average 190 LOCs. This is the first multi-lingual program repair benchmark. The dataset can be found on Github at `https://github.com/jkoppel/QuixBugs`.

Summary: Many program repair benchmarks are available for Java compared to other languages such as Python. In addition, the used benchmarks of different sizes come from both real and toy projects to ensure the diversity of the experiments.

6.3 Automated Program Repair Techniques

There are three main automatic program repair categories: Generate and Validate, Semantic-based and Metaprogramming-based [61].

6.3.1 Generate and Validate APR techniques

The main technique consists of the following three steps:

1. Take original program, take all the test cases
2. Modify the program (mutation or genetic program techniques)
3. Modified program is a fix if all test cases pass

JGenProg [112] is the Java implementation of GenProg [100]. The techniques use a generate-and-validate method to produce patches using a genetic programming approach. The search space consists of patches that are formed through combinations of removing code, and inserting and replacing code from elsewhere in the program under repair [112]. Listing 6.1 shows an example buggy program (The endless loop on lines 5-9). The patch for the Listing 6.1 is shown in Listing 6.2. The complete program after applying GenProg is shown in Listing 6.3.

JKali [112] is the implementation of Kali [155] for Java. They attempt to come up with candidate patches by removing or skipping statements. In particular, the operators implemented in Kali are removal of statements, modification of *if* conditions to *true* and *false*, and insertion of *return* statements [49].

```
1   void gcd(int a, int b) {
2   if (a == 0) {
3   printf("%d", b);
4   }
5   while (b != 0)
6       if (a > b)
7           a = a - b;
8       else
9           b = b - a;
10  printf("%d", a);
11  exit(0);
12  }
```

Listing 6.1: Starting program

```
void gcd_2(int a, int b) {
printf("%d", b);
exit(0);
```

```
            }
            }
```

Listing 6.2: Proposed fix

```
void gcd_3(int a, int b) {
            if (a == 0) {
            printf("%d", b);
            exit(0); // inserted
            a = a - b; // inserted
            }
            while (b != 0)
                  if (a > b)
                        a = a - b;
                  else
                        b = b - a;
            printf("%d", a);
            exit(0);
            }
```

Listing 6.3: Fixed program

Other techniques belong to this repair category are ACS [214], ARJA [218], CapGen [209], Cardumen [113], DeepRepair [210], Elixir [161].

6.3.2 Sematics-based APR techniques

Semantics-based APR techniques uses symbolic execution and test suites to extract semantic constraints, and uses program synthesis to synthesize repairs that satisfy the extracted constraints [97]. The main steps of the techniques in this category are:

- Take original buggy program and all the test cases
- Encode the program formally or explicitly (e.g., convert a program into a formula)
  - behavioral analysis
  - problem generation
  - fix generation
- Solve the formula [1]

[1] Solution is guarantted to solve the problem (bug), but is not guarantted to be satisfactory

Nopol [215] is a repair tool for Java. Nopol repairs buggy conditional statements (i.e., *if-then-else* statements). In particular, the approach takes a buggy program as well as a test suite as input and generates a patch with a conditional expression as output.
DynaMoth [50] performs a dynamic synthesis of patches for repairing conditional bugs. Although the tool is integrated into Nopol that also targets buggy and missing *if* conditions, DynaMoth uses the Java Debug Interface to access the runtime context and collects variable and method calls instead of using an SMT formula to generate a patch.

6.3.3 Metaprogramming-based APR techniques

The idea of the techniques is to tranform the program under repair with automated code transformation, so as to obtain a metagram. Typically, the main algorithm consists of the following steps:

1. Take original program, take all the test cases
2. Create metaprogramming description of a program
3. Run the program
4. Use runtime information and metaprogramming description to identify bugs and generate a fix

NPEFix [38] repairs null pointer exceptions at runtime by using two strategies. The first strategy assigns an alternative value (which can be a valid value that is stored in another variable or a random value) for a null dereference. The second strategy skips the execution of the null dereference, by either skipping a single statement or skipping the complete method. All strategies are applicable for any arbitrary objects, including instances of library classes, and instances of domain classes.

6.4 Evaluation of APR Techniques

Various evaluation studies investigate the success of APR tools on different benchmarks. Table 6.1 shows factors that have been investigated on the success of APR tools in generating patches. While most of the studies focus on the internal aspects of the APR tools (e.g., representation of genetic operators, localization size, fault space size), external factors (e.g., developer reported defect severity, number of files touched by developers in report, time needed to wait for a patch) did not get much of attention.

Table 6.1: Factors affecting Success/Unsucccess Patches

Evaluation criteria	**APR Studies**
Fault localization techniques	[49, 98, 153]
Fix space	[98, 155]
Human time to repair	[98]
Human repair size	[98]
Defect difficulty/severity	[98, 121]
Test suites	[170]
Time budgets	Ours [199]

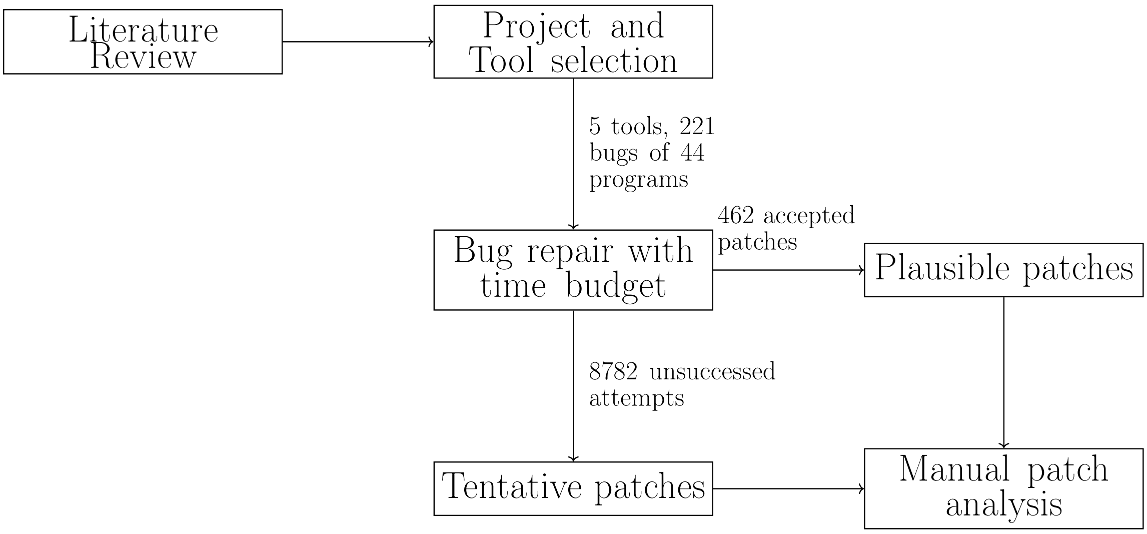

Literature Review
Project and Tool selection
5 tools, 221 bugs of 44 programs
462 accepted patches
Bug repair with time budget
Plausible patches
8782 unsuccessed attempts
Tentative patches
Manual patch analysis

Figure 6.1: Experiments of different APR tools on different benchmarks constrained by time budgets

6.5 Preliminary Experiments on APR tools constrained by time budgets

Figure 6.1 shows the setup of our experiment. For the APR tool selection, we used the RepairThemAll framework [49] and selected five tools for Java programs belonging to three repair technique families: *generate-and-validate*, *semantic-driven*, *metaprogramming-based*. The legend of Figure 6.3 shows the final set of APR tools.

We selected the five most popular benchmarks having both a set of buggy programs with known locations of software bugs and a set of test cases to validate the generated patches. For each benchmark, we randomly selected a project and extracted all its bugs.

Table 6.2: Repair Benchmarks used in this study
Java is chosen due to its popularity in the automatic program community. For example, popular APR tools such as GenProg or Kali have been ported to Java, and popular benchmarks such as Defects4J are collected from Java projects. Moreover, after the interviews in Chapter 3, we observed that Java is also one of the four most popular programming languages used by professional developers.

Benchmark	Project	#Bugs	Language
Defects4J	Chart	26	Java
Bears	spring-projects-spring-data-commons	15	Java
Bugs.jar	Accumulo	88	Java
IntroClassJava	smallest	52	Java
QuixBugs	40 projects	40	Java

For the QuixBugs benchmark, we selected all available projects as they contain only one bug per project. This selection resulted in 221 bugs from 44 projects (Table 6.2). This study focuses on APR techniques designed to fix bugs in Java programs as a state-of-the-art benchmark dataset. We selected the most popular samples that contain both a set of buggy programs with known locations of software bugs and a corresponding set of test cases to validate the generated patches. For each benchmark, we randomly selected a project and extracted all the bugs affecting this project. For the QuixBugs benchmark, we selected all the available projects as they are relatively small (e.g., the benchmark contains only one bug per project).

To benchmark the APR tools, we give each tool exponentially longer deadlines and count the corresponding cumulative patches. A generated patch is measured successful if it passes all the specified test cases for the particular software program. Thus, the anytime algorithm should provide more successful patches in proportion to the increased amount of available time [77]:

- each tool takes as same input set of the studied programs with known bugs and the same starting parameters for each run (i.e., the predefined seed);[2]
- we terminate the repair process if a given time budget is exceeded or the first successful patch is generated;
- after each run, we double the time interval that a tool has for generating plausible bug fixes.

[2] NPEFix was an exception as it does not support such an option.

6.6 Findings

Our experiment performed 9244 repair attempts, 462 of the attempts resulted in a plausible patch, while 8782 attempts terminated without generating a patch.

Figure 6.2 shows the fractions of the total number of patches generated by the selected APR tools. We observe that within one minute, the tools fixed 9% of the total number of bugs. When we doubled the time (e.g., two minutes), the number of plausible patches increased to 13.6%. This number of patches already deviates from the expected increase in the number of patches: if the APR tools are anytime algorithms, the expected fraction of patches after two minutes is 18%. The difference between the expected fraction of patches and the actually registered ones increases with time. We observe 19% of bugs fixed after running the tools for four minutes (36% is expected) and 28% of bugs fixed after 16 minutes (72% is expected). Hence, we observe a sub-linear increase in the fraction of patches while the tool running time increased exponentially.

Figure 6.3 shows the fraction of patches each tool generated after a particular time interval. We observe that Nopol fixed the biggest fraction of bugs: already after the first minute, it generated patches for 5% of bugs. However, each consecutive increase of the tool's time budget reduces the number of newly generated patches: +2% after running for the second minute, +2% after four minutes, and +1% after running for 16 minutes. Similarly, jKali generated 2% of patches after running for one minute and then was able only to double the number of fixes (5% of bugs) while had 16x more time.

jGenProg and NPEFix fixed a small fraction of bugs after running for one minute (1% of total bugs). After waiting 4x time for jGenProg and 16x time for NPEFix, we obtained more fixes. However, the number of patches increased modestly while the tools required a substantially longer time to generate them (5x patches required 16x time).

DynaMoth demonstrated a somewhat different behavior. It produced no patches after running for one minute then generated patches for 2% of bugs after two minutes. The next doubling of the time interval leads to just one additional bug fix. After 16 minutes, DynaMoth patched 7% of bugs, which is the second-best result within the evaluated tools. Even if we observe a stepped increase in the number of fixes, it still does not correspond to the exponential growth in the amount of time required for the tool to produce them.

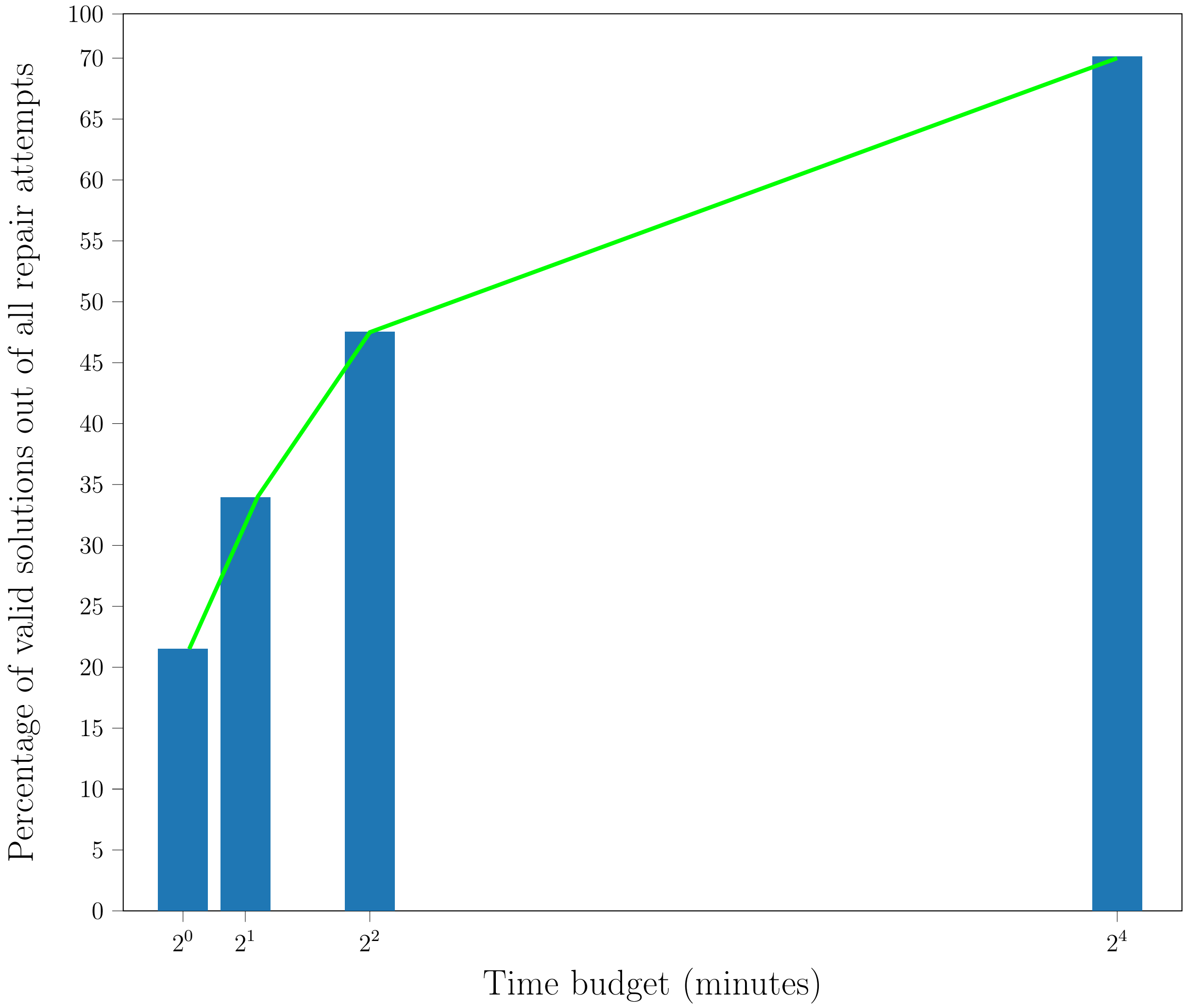


Figure 6.2: Total generated patches by all APR tools

Valid solutions are plausible patches that pass all test cases. Plausible patches are accepted by most studies in APR evaluation [105]

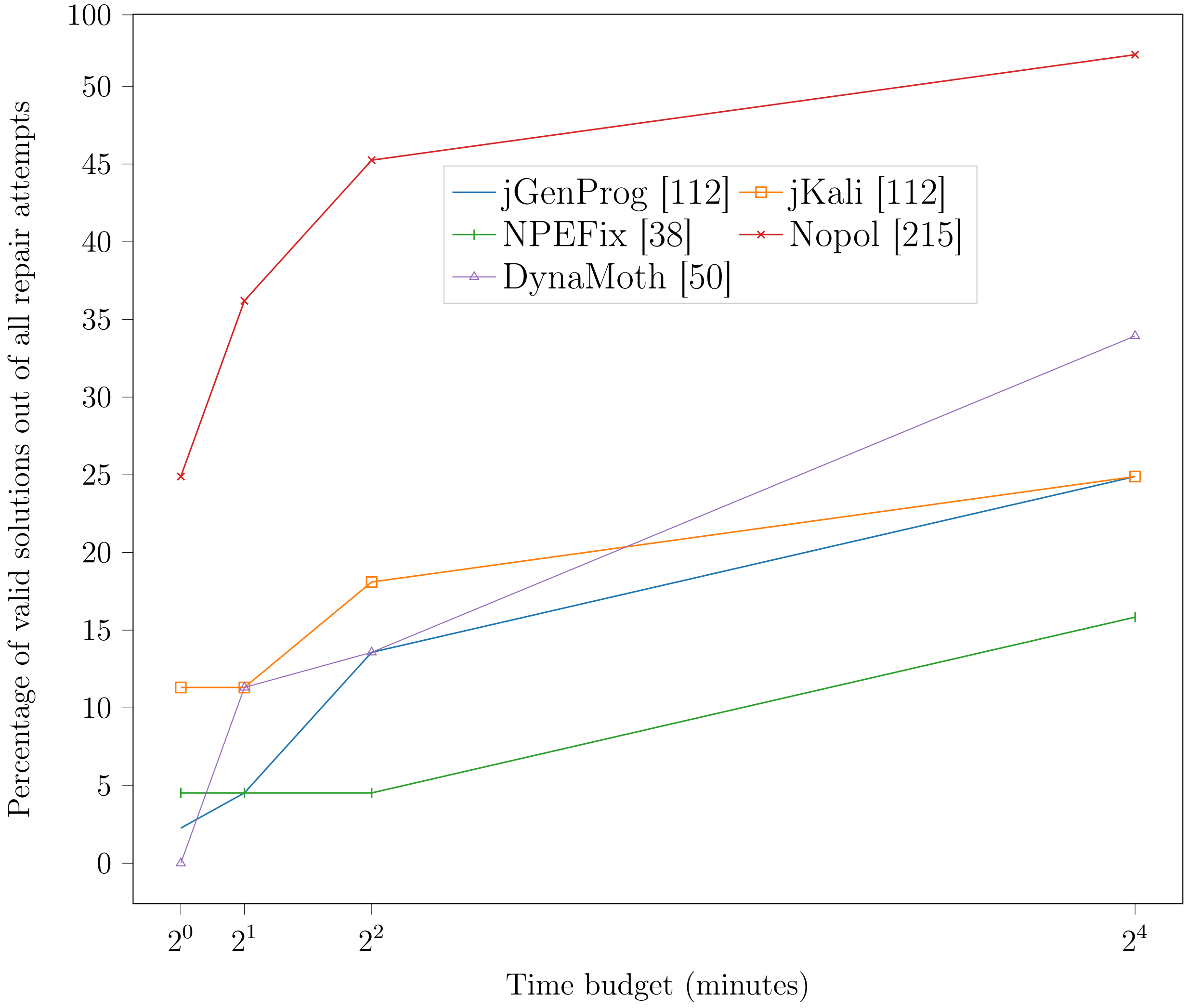


Figure 6.3: Generated patches by each individual APR tool

Valid solutions are plausible patches that pass all test cases. Plausible patches are accepted by most studies in APR evaluation [105]

Table 6.3: Manual Validation of the Generated Plausible Patches

Time budget	Benchmark	Project	Bug ID	Tool	Correct/Incorrect
1 min	QuixBugs	QuixBugs	DEPTH_FIRST_SEARCH	jKali	Incorrect
4 minutes	QuixBugs	QuixBugs	QuixSort	Nopol	Incorrect
16 minutes	Defects4J	Chart	5	Nopol	Correct
16 minutes	QuixBugs	QuixBugs	FIND_IN_SORTED	Nopol	Correct
16 minutes	QuixBugs	QuixBugs	FIND_IN_SORTED	Nopol	Correct

6.7 Manual Validation of generated patches

In this section, we examine the success and failed attempts as well as the correctness of the plausible patches.

6.7.1 Reason of Failed Repair Attempts

Giving a minute for the tool JGenProg to fix the bug in Chart 5 only allows it to generate the first variant. The execution trace is shown below.

```
---- Gzoltar fault localization
---- Creating spoon model
---- Creating program variant #1, [Variant id: 1, #gens: 19]
```

By giving JGenProg a very small time budget, it was only able to locate the fault and create a variant. More time is needed to verify the variant against the test cases and explore more variants.

6.7.2 Success repair attempts

We randomly select some plausible patches and compare with the human patches. Table 6.3 shows the number of generated patches that are correct and incorrect by our assessment.

```
// source/org/jfree/data/xy/XYSeries.java
public XYDataItem addOrUpdate(Number x, Number y) {
    if (x == null) {
        throw new IllegalArgumentException("Null ’x’ argument.");
    }
    // if we get to here, we know that duplicate X values are not permitted
    XYDataItem overwritten = null;
    int index = indexOf(x);
    if (index >= 0 && !this.allowDuplicateXValues) {
        XYDataItem existing = (XYDataItem) this.data.get(index);
        try {
            overwritten = (XYDataItem) existing.clone();
            }
        catch (CloneNotSupportedException e) {
            throw new SeriesException("Couldn't clone XYDataItem!");
            }
        existing.setY(y);
        }
    else {
        // if the series is sorted, the negative index is a result from
        // Collections.binarySearch() and tells us where to insert the
        // new item...otherwise it will be just -1 and we should just
        // append the value to the list...
        if (this.autoSort) {
-          this.data.add( -index - 1, new XYDataItem(x,y) );
+          this.data.add(new org.jfree.data.xy.XYDataItem(x, y));
         else {
            this.data.add(new XYDataItem(x, y));
         }
         // check if this addition will exceed the maximum item count...
         if (getItemCount() > this.maximumItemCount) {
            this.data.remove(0);
         }
     }
     fireSeriesChanged();
     return overwritten;
 }
```

Listing 6.4: Patch generated for Chart-5 by running JGenProg for four minutues

```
// source/org/jfree/data/xy/XYSeries.java
public XYDataItem addOrUpdate(Number x, Number y) {
    if (x == null) {
        throw new IllegalArgumentException("Null 'x' argument.");
    }
+   if (this.allowDuplicateXValues) {
```

```
+       add(x, y);
+       return null;
+   }

    // if we get to here, we know that duplicate X values are not permitted
    XYDataItem overwritten = null;
    int index = indexOf(x);
-   if (index >= 0  && !this.allowDuplicateXValues ) {
+   if (index >= 0) {
        XYDataItem existing = (XYDataItem) this.data.get(index);
        try {
            overwritten = (XYDataItem) existing.clone();
            }
        catch (CloneNotSupportedException e) {
            throw new SeriesException("Couldn't clone XYDataItem!");
            }
        existing.setY(y);
        }
    else {
        // if the series is sorted, the negative index is a result from
        // Collections.binarySearch() and tells us where to insert the
        // new item...otherwise it will be just -1 and we should just
        // append the value to the list...
        if (this.autoSort) {
            this.data.add(-index - 1, new XYDataItem(x,y));}
         else {
            this.data.add(new XYDataItem(x, y));
         }
         // check if this addition will exceed the maximum item count...
         if (getItemCount() > this.maximumItemCount) {
            this.data.remove(0);
         }
     }
     fireSeriesChanged();
     return overwritten;
 }
```

Listing 6.5: Human Patch generated for a bug in Chart-5 program

Listing 7.4 shows a patch generated by jGenProg for the bug ID 5 of the Chart project with a timeout of four minutes. The bug is caused by not checking duplicate values before adding, as shown in Listing 7.5. jGenProg, however, fixed the bug by creating a new instance of the class representing the pair of values, which is clearly not correct.

6.8 Threats to Validity

We only measure the number of plausible patches regarding increasing number of time budgets. However, the number of patches generated along with increasing time budgets should be highly relevant to multiple factors, including patch generation strategies, patch validation strategies, and implementations of the APR tools.

We only consider five APR tools in our study. The extension to other APR tools seems straightforward (e.g., tools already provided by RepairThemAll framework jMutRepair). However, we only considered a small fraction of bugs in the benchmarks. More bugs would need to be considered for a more comprehensive analysis.

We limit our experiment and analysis to Java projects. The main reason is that Java is a popular language and is most focused on by the APR community.

We fit the *seed* parameter to ensure the reproduciblity, however the performance of some APR tools (e.g., JGenProg) can be affected by the predefined *seed* parameter.

We measure APR tools' performance with anytime algorithm, however APR tools could be considered as other algorithms.

We only run a single replication for each run because it is very time-consuming to run multiple experiment replications. For example, running 11 repair tools on 2141 bugs took almost a year of continuous execution on the big cluster of machine [49].

6.9 Conclusions

We evaluated the automated software repair tools with different time budgets. We found that by giving exponentially more time, the number of patches only increases linearly or not at all. If no quick fix is generated (within the first four minutes), one is unlikely to be generated.

Our code and replication data are available on Github at `https://github.com/assuremoss/Automated-Program-Repair/tree/main/Anytime-Algorithm-2021`.

7

What is Next?

> *"Put yourself, and your work, out there every day, and you'll start meeting some amazing people."*
>
> – Bobby Solomon

This thesis investigated different stages and actors of the software supply chain from the security point of review holistic view of software supply chain security. To do so, the thesis used both qualitative and quantitative methods to provide a set of actionable implications and techniques that developers and maintainers can use in different stages of the software supply chain. This thesis presents a suite of solutions that might support software practitioners selecting a suitable dependency, understanding the dependency, and fixing bugs in the dependency.

Several nuances are still unaddressed by our study in Chapter 3, 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 the community at large is developing the dependencies and security analysis tools our developers require. In addition, an interesting future work would be to seek more interviews with developers involved in developing FOSS projects and study developers from different development communities who share different views on software dependencies.

Chapter 4 provides an approach to link the package and its corresponding source code repository and provide a set of metrics reported by developers in the qualitative studies in Chapter 3. Future directions for the community are to understand how the activity of a package (e.g., number of releases, time to release) and the information about security fixes might affect the popularity of a package.

LastPyMile presented in Chapter 5 provides a starting point to investigate discrepancies in the code hosted on the source code repository and the code hosted within the package repository. Future research directions might be to analyze configuration files and provide more precise analysis on the discrepancies (e.g., source code dynamic analysis). Besides, attackers could hide malicious code in many other forms, such as webpages (HTML with embedded or external JavaScript) or configuration files (*requirements.txt* with a malicious dependency). We notice a high number of dependency declaration files *requirements.txt*, which contain the list of dependencies to be installed automatically with `pip install`. This could be a potential vector for adversaries to add malicious injections worth further investigations. As for the next steps, the specific approaches that work for the current package managers, such as PyPI, NPM, and RubyGems, need to be developed. In particular, the first step would be to develop an automatic approach to identify the source code repository of the uploaded software package. Then a package repository could leverage the git log or the LastPyMile approaches to identify the possible differences between the source code repository and the uploaded package. Finally, the most challenging step would be to design a fast, reliable, and precise way to check the identified differences for the presence of malicious injections.

Chapter 6 investigate the practicality aspects of APR tools (e.g., a function of time for the number of plausible patches) and explore the changes in APR tools after a while. Future directions might be to try more advanced APR techniques to determine the best one that meets the developers' needs. Future experiments will be scheduled on a more extensive cluster of machines to reduce the potential bias of a single misleading run. In addition, more researches might be needed to evaluate the applicability and efficacy of APR techniques to fix vulnerabilities. To do so, researchers should devote more effort to curating a new vulnerability dataset for program repairs.

8

Appendix

8.1 CRediT authorship contribution statement

This section details the contribution of Ly Vu Duc to the main25 published works followed by the CRediT authorship contribution statement by Elsevier [52].

- **A qualitative study of dependency management and its security implications [136]:** the identification of research questions, conducted and transcribed the interviews, coded the interviews and the code analysis and contributed to writing the paper.
- **PY2SRC: Automatic Identification of Source Code Repositories and Factors for Selecting New PyPI Packages [198]:** the identification of research questions, developed the tool, analyzed the data, manual validation of the findings, and contributed to writing the paper.
- **LastPyMile: identifying the discrepancy between sources and packages [202]:** the identiffication of research questions developed the approach, analyzed the data, and contributed to writing the paper.
- **Please hold on: more time = more patches? Automated program repair as anytime algorithms [199]:** the identification of research questions, collected benchmarks and tools, analyzed the results, and contributed to writing the paper.

8.2 Ethical Issues

The procedure complies with UNITN guidelines for human studies that do not involve processing of personal data and observation of participants.

ai sensi dell'art. 3 delle Regole deontologiche per trattamenti a fini statistici o di ricerca scientifica pubblicate ai sensi dell'art. 20, comma 4, del d. lgs. 10 agosto 2018, n. 101 – 19 dicembre 2018

8.2.1 Main Ethical Issues

Table 8.1: Main Ethical Issues

	Ethical issue	**YES**	**NO**	**Comment**
1	Will data be collected or analyzed in the course of the project? Data include interview and survey responses, observations, registrations, social media, content, and visual images	XX		Data about developers' perceptions of software dependencies and the effect of security concerns on their decisions through an semi-structure interviews.
2	Is the data collected in the public or in the private domain?			
2.1	Public domain data: includes observations made during a public event, observations of public figures, or data extracted from public online sources (including publicly accessible social media data)		XX	
2.2	Private domain data: includes observations made using (online) questionnaires, experiments, interviews, and focus groups as well as data extracted from private digital sources	XX		Online and in-person intereviews where participants are asked about their decision-making strategies for selecting, managing, and updating software dependencies.
3	Do the data you use contain personal or sensitive information?			
3.1	Personal data: It includes information that may identify a specific person, e.g. name, address, phone number, IP address, bank account number, social security number		XX	Temporary personal data (name and email) will be collected to identify the participants with the online system but this data will not be used for the research and replaced with anonymous identifiers.
3.2	Sensitive data: It includes information such as race, religion, sexual orientation, criminal record, and political preferencer		XX	
4.1	Will the research project involve merging multiple data sets?		XX	

5.1	Will participants in the research project be asked for informed consent?	XX		At the beginning of the experiment they will be asked to consent to the use of their technical answer for research purposes
6.1	Does the planned research pose potential risks to the participants during or after the research? Risks may include physical and psychological harm or discomfort.		XX	Risk may include normal fatigue when doing a 30 minutes interview session.
6.2	Does the research pose potential risks to a population or group from which participants are drawn? Risks may include stigmatization, or reputational or economical damage.		XX	
6.3	Are participants individuals who are vulnerable? Participants may be vulnerable when they depend on others for assistance in daily life, or when they experience threats or physical danger.		XX	
6.4	Will participants be exposed to material, social, or psychological recruitment incentives that are stronger than usual? E.g., are payments made that are higher than the minimum wage? Are individuals exposed to strong social or psychological pressure to participate?		XX	
6.5	Will participants be exposed to research stimuli (e.g., pictures, video, text) that may be distressing, offensive, or age-inappropriate? Research stimuli may be considered distressing or offensive if they are stronger than what participants would normally be exposed to in daily life.		XX	
7.1	Does the research pose potential risks to the researchers? Risks may include physical and psychological harm or discomfort.		XX	

8.1	Will participants be deceived in the research? Deception means that the protocol deliberately give participants information that is false.		XX	
8.2	If participants are deceived will they be debriefed afterwards?	XX		
9	Is anonymity or confidentiality to participants guaranteed?			
9.1	Anonymity: The identity of participants will remain unknown, also to the researchers.		XX	The researcher could know the identity of the participants to send them the transcripts as well as complete findings for confirmation.
9.2	Confidentiality: Researchers will know the identity of participants, but it will be unknown to others.	XX		All personal data will be replaced with anonymous identifiers.
10.1	Will research and data collection also possible outside of the country?	XX		The participants will be debriefed on where the vulnerability is present.
10.2	Outside the country but within European Union		XX	
10.3	Outside the European Union	XX		
11	There is the possibility of misusing the research instruments or the research results. This includes providing knowledge, materials and technologies that could be adapted for criminal/terrorist activities			
11.1	Misuse by participants.		XX	
11.2	Misuse by researchers.		XX	
11.3	Misuse by third parties.		XX	
12.1	Is there the possibilities to have incidental findings that impact participants or third parties		XX	
12.2	If there are incidental findings is there a procedure to disclose them to the interested parties	XX		

8.2.2 Main Ethical Mitigations

- The research collects temporary personal data (name and email) to identify the participants with the online system but this data will not be used for the research and replaced with anonymous identifiers
- 5.1: Participants in the research project will be asked for informed consent to the use of their technical answer for research purposes
- 6.1: The potential risks to the participants during the research may include normal fatigue when doing a 2 hours digital task
- 6.4: Participants are exposed to recruitment incentives that are not stronger than usual since participation is included in the course but the use of research data is entirely voluntary and does not affect the grading of the course.
- 8.1: Participants will not be deceived in the research.
- 8.2: At the end of the task participants will be debriefed on where the vulnerability is present.
- 9: Anonymity or confidentiality to participants is guaranteed. The researcher could know the identity of the participants if they correlate the authentication system with the data submitted and do not delete such information for the preparation of the research data. All personal data will be replaced with anonymous identifiers
- 10: Research and data collection also possible outside of the country as individual participants might come from industry partners around the world.
- 11: There is no possibility of misusing the research instruments or the research results.
- 11.2: Misuse by researchers of the results of the research is also impossible.
- 12.1: There are limited possibilities to have incidental findings that impact participants or third parties.

8.2.3 Data Collection and Protection

The research will involve the collection of information from the participants. This is achieved through the interviews.

The proposed research involves an initial collection and processing of personal data which will be then anonymized.

Specifically, personal data such as :

1. partificipant's name and email will be used to communicate the participants to share the findings and ask for feedback.

2. participants will then be assigned a pseaudonym reference code, which will be used instead of their personal ID for storing, reporting, and disseminating results.

Background and results of the expertise in Computer Science related topics will be collected during the experiments mentioned above that will only be used in an anonymous form.

Upon request, the participants will know where their personal data will be stored. The information related to the the personal data storage duration is included on the consent form to be signed by the participants. They will be informed that these data will not be used for commercial purposes. The collection of authentication information for students will happen through a University of Trento provided system for teaching and will follow the privacy protection rules of that system. Data resulting from the exercise will be stored and handled according to the provisions of the EU General Data Protection Regulation. Electronic records will be stored on a computer with password-protected access and backed up on media with password-protected access.

8.3 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 8.2). Then we used our tool for the dependency study (See §5.3 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 [6], 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

Table 8.2: Top 20 most starred projects from Github

Github Repository 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

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.

8.4 Appendix – Codes Distribution

Figure 8.1 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

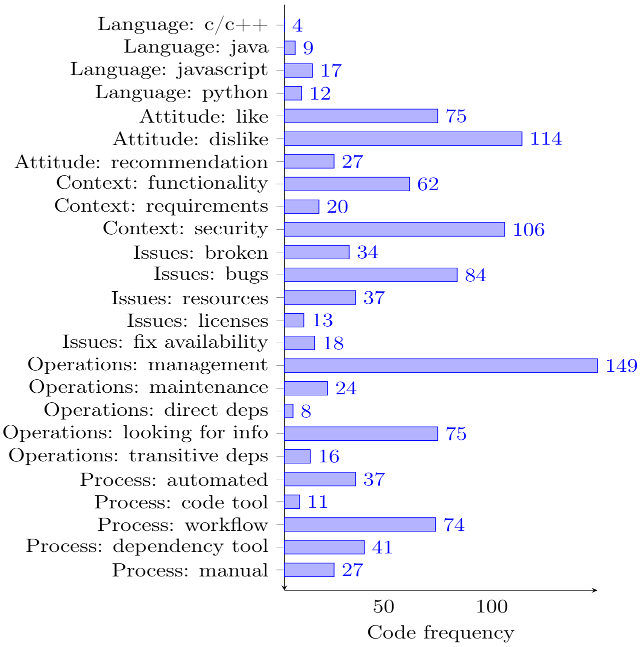


Figure 8.1: Code frequency by code groups

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)`

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 [90]: 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.

8.5 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 backend. 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.

8.6 Appendix – Complete co-occurrence table

Figure 8.2 shows the complete co-occurence table for the Section 3.3.

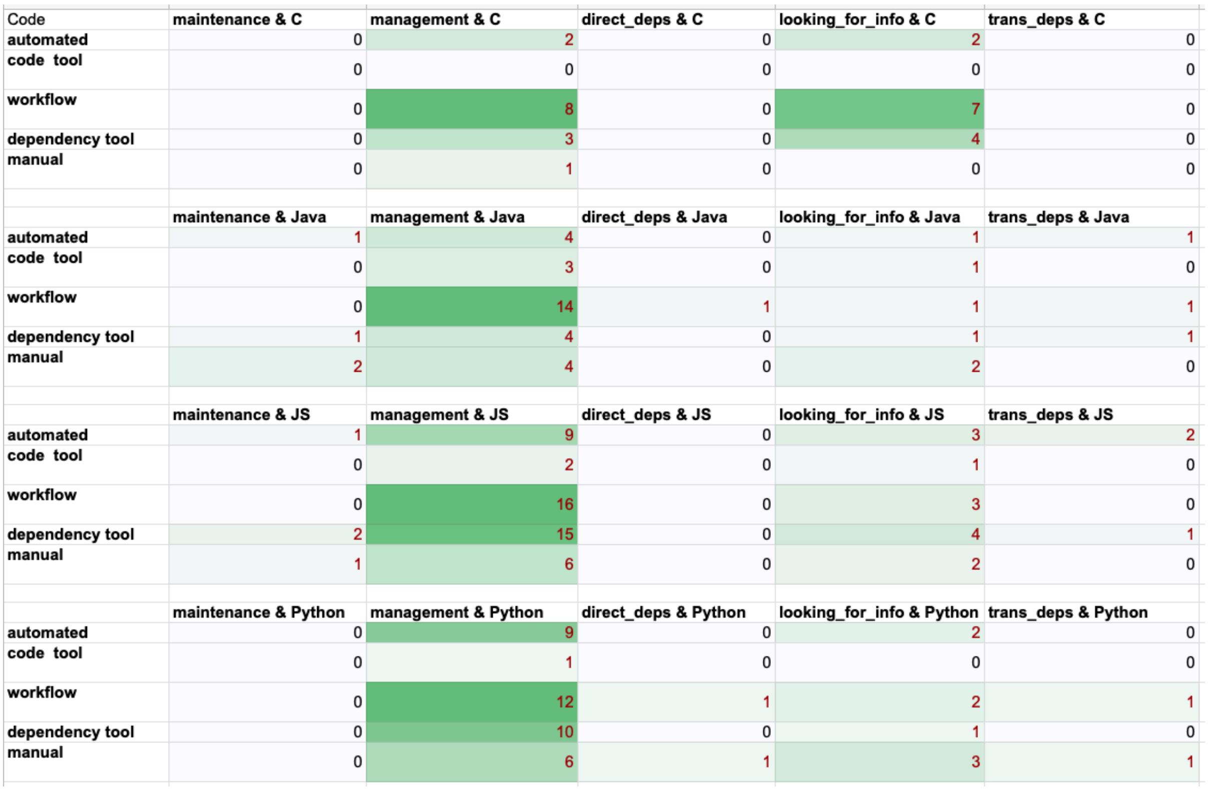
8.7 Appendix – Per language analysis

Figure 8.4 shows the distribution of codes by languages.

8.8 Appendix – Typosquatting and Combosquatting Attacks in PyPI

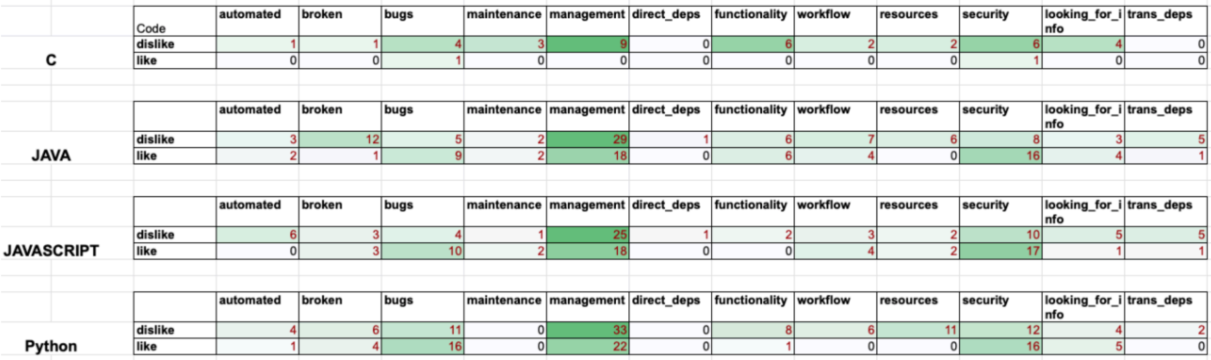
Code	automated	broken	bugs	c/c++	code tool	dependency tool	direct deps	dislike	fix availability	functionality	java	javascript	licenses	like	looking for info	maintenance	management	manual	python	recommendation	requirements	resources	security	transitive deps	workflow
automated	0	1	9	6	7	18	0	14	0	2	8	12	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 8.2: Full Co-occurence table. Available online on Zenodo at [137]



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
	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
	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
	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

Figure 8.3: Developer operations for languages. Available online on Zenodo at [137]



	Code	automated	broken	bugs	maintenance	management	direct_deps	functionality	workflow	resources	security	looking_for_info	trans_deps
	dislike	1	1	4	3	9	0	6	2	2	6	4	0
C	like	0	0	1	0	0	0	0	0	0	1	0	0
		automated	broken	bugs	maintenance	management	direct_deps	functionality	workflow	resources	security	looking_for_info	trans_deps
	dislike	3	12	5	2	29	1	6	7	6	8	3	5
JAVA	like	2	1	9	2	18	0	6	4	0	16	4	1
		automated	broken	bugs	maintenance	management	direct_deps	functionality	workflow	resources	security	looking_for_info	trans_deps
	dislike	6	3	4	1	25	1	2	3	2	10	5	5
JAVASCRIPT	like	0	3	10	2	18	0	0	4	2	17	1	1
		automated	broken	bugs	maintenance	management	direct_deps	functionality	workflow	resources	security	looking_for_info	trans_deps
	dislike	4	6	11	0	33	0	8	6	11	12	4	2
Python	like	1	4	16	0	22	0	1	0	0	16	5	0

Figure 8.4: Developer attitutes for languages. Available online on Zenodo at [137]

Table 8.3: Malicious PyPI packages in our sample.

					Levenshtein distance (*d*)	
#	**Time Appear**	**Malicious Package**	**Legitimate Package**	**Names change**	**d=1**	**d=2**
1	2016-03-02	virtualnv	virtualenv	Delete ‘e’	✓	
2	2016-03-03	mumpy	numpy	Substitute ‘n’ by ‘m’	✓	
3	2017-05-01	crypt	crypto	Delete ‘o’	✓	
4	2017-06-02	django-server	django-server-guardian-api	Delete “-guardian-api”		
5	2017-06-02	pwd	pwdhash.py	Delete ‘hash.py’		
6	2017-06-02	setuptool	setuptools	Delete ‘s’	✓	
7	2017-06-02	setup-tools	setuptools	Insert ‘-’	✓	
8	2017-06-02	telnet	telnetsrvlib	Delete ‘srvlib’		
9	2017-06-02	urlib3	urllib3	Delete ‘l’	✓	
10	2017-06-02	urllib	urllib3	Delete ‘3’	✓	
11	2017-06-03	acqusition	acquisition	Delete ‘i’	✓	
12	2017-06-03	apidev-coop	apidev-coop_cms	Delete ‘_cms’		
13	2017-06-04	bzip	bz2file	Substitute ‘2file’ by ‘ip’		
14	2017-11-23	djanga	django	Substitute ‘a’ by ‘o’	✓	
15	2017-11-24	easyinstall	easy_install	Delete ‘_’	✓	
16	2017-12-05	colourama	colorama	Delete ‘u’	✓	
17	2018-04-25	openvc	opencv-python	Swap ‘c’ and ‘v’ & Delete “-python”		
18	2018-05-02	mateplotlib	matplotlib	Insert ‘e’	✓	
19	2018-05-02	numipy	numpy	Insert ‘i’	✓	
20	2018-05-02	python-mysql	MySQL-python	Swap ‘python’ and ‘mysql’		✓
21	2018-05-03	libcurl	pycurl	Substitute ‘py’ by ‘lib’		
22	2018-05-03	libhtml5	html5lib	Swap ‘html5’ and ‘lib’		
23	2018-05-03	pysprak	pyspark	Swap ‘a’ and ‘r’		✓
24	2018-05-03	PyYMAL	pyyaml	Swap ‘a’ and ‘m’		✓
25	2018-05-10	nmap-python	python-nmap	Swap ‘nmap’ and ‘python’		✓
26	2018-05-10	python-mongo	pymongodb	Delete ‘db’ & Substitute ‘py’ by ‘python-’		
27	2018-05-10	python-openssl	openssl-python	Swap “openssl” and “python”		✓
28	2018-09-17	pytz3-dev	pytz	Insert ‘3-dev’		
29	2018-10-29	python-sqlite	pysqlite	Substitute ‘py’ by ‘python-’		
30	2018-10-30	python-ftp	pyftpdlib	Delete ‘dlib’ & Substitute ‘py’ by ‘python-’		
31	2018-10-30	python-mysqldb	MySQL-python	Swap ‘python’ and ‘mysql’ & Insert ‘db’		
32	2018-10-30	smb	pysmb	Delete ‘py’		✓
33	2018-10-31	pythonkafka	kafka-python	Swap ‘kafka’ and ‘python’ & Delete ‘-’		
34	2019-12-01	jeIlyfish	jellyfish	Substitute ‘l’ by ‘I’	✓	
35	2019-12-01	python3-dateutil	python-dateutil	Insert ‘3’	✓	
36	2018-04-25	ssh-decorate	ssh-decorate	Hijacked Package		

Bibliography

[1] Backdoor in ssh-decorator package. https://www.reddit.com/r/Python/comments/8hvzja/backdoor_in_sshdecorator_package/.

[2] In-depth comparison of files, archives, and directories. https://diffoscope.org.

[3] Tracking which wheels can be reproducibly built. https://www.redshiftzero.com/reproducible-wheels/.

[4] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering*, 25(2):1168–1204, 2020.

[5] Github Actions. Automate your workflow from idea to production. https://github.com/features/actions.

[6] B Adams. 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.

[7] Alan Agresti and Christine Franklin. *Statistics the art and science of learning from data*. Pearson Education Limited, 2018.

[8] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–45. IEEE, 2021.

[9] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. Tracing known security vulnerabilities in software repositories–a semantic web enabled modeling approach. *Science of Computer Programming*, 121:153–175, 2016.

[10] Anaconda. Anaconda: Installers & packages. `https://repo.anaconda.com`.

[11] Google APIs. Public interface definitions of google apis. `https://github.com/googleapis/googleapis`.

[12] AppVeyor. Appveyor: Ci/cd service for windows, linux and macos. `https://www.appveyor.com`.

[13] Twitter Archive. Commons: Twitter common libraries for python and the jvm (deprecated). `https://github.com/twitter-archive/commons`.

[14] Hala Assal and Sonia Chiasson. 'think secure from the beginning' a survey with software developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–13. ACM, 2019.

[15] Atlas.ti. Atlas.ti: The qualitative data analysis & research software. `https://atlasti.com`.

[16] AWS. aws-cli: Universal command line interface for amazon web services. `https://github.com/aws/aws-cli`.

[17] AWS. Pypi package six: Universal command line environment for aws. `https://pypi.org/project/awscli/`.

[18] Aadesh Bagmar, Josiah Wedgwood, Dave Levin, and Jim Purtilo. I know what you imported last summer: A study of security threats in thepython ecosystem. *arXiv preprint arXiv:2102.06301*, 2021.

[19] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 316–331. Springer, 2012.

[20] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 125–134. IEEE, 2010.

[21] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[22] William Bengtson. Python typosquatting for fun not profit. https://medium.com/@williambengtson/python-typosquatting-for-fun-not-profit-99869579c35d.

[23] Benjaminp. Metadata of the six package in pypi. https://pypi.org/pypi/six/json.

[24] Benjaminp. Pypi package six: Python 2 and 3 compatibility utilities. https://pypi.org/project/six/.

[25] Benjaminp. six: Python 2 and 3 compatibility library. https://github.com/benjaminp/six.

[26] Bertus. Detecting cyber attacks in the python package index (pypi). https://medium.com/@bertusk/detecting-cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67.

[27] Alex Birsan. Dependency confusion: How i hacked into apple, microsoft and dozens of other companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

[28] Bitbucket. Bitbucket: The git solution for professional teams. https://bitbucket.org.

[29] Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89. IEEE, 2015.

[30] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM, 2016.

[31] Ethan Bommarito and Michael Bommarito. An empirical analysis of the python package index (pypi). *arXiv preprint arXiv:1907.11073*, 2019.

[32] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016.

[33] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 516–519. IEEE, 2015.

[34] Kyriakos Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas Symeonidis. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 42–45. IEEE, 2018.

[35] Catalin Cimpanu. Backdoored python library caught stealing ssh credentials. `https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/`.

[36] Catalin Cimpanu. Two malicious python libraries caught stealing ssh and gpg keys. `https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/`.

[37] Google code. Google code archive. `https://code.google.com/archive/`.

[38] Benoit Cornu, Thomas Durieux, Lionel Seinturier, and Martin Monperrus. Npefix: Automatic runtime repair of null pointer exceptions in java. *arXiv preprint arXiv:1512.07423*, 2015.

[39] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 109–118. IEEE, 2015.

[40] Russ Cox. Surviving software dependencies: Software reuse is finally here but comes with risks. *Queue*, 17(2):24–47, 2019.

[41] Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. On the security cost of using a free and open source component in a proprietary product. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 190–206. Springer, 2016.

[42] Cleidson de Souza and David Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the 30th ACM/IEEE*

International Conference on Software Engineering (ICSE), pages 241–250. IEEE, 2008.

[43] Debian. Reproducible builds. `https://reproducible-builds.org/`.

[44] Debian. Reproducible builds: Wiki. `https://wiki.debian.org/ReproducibleBuilds`.

[45] Charlie Denton. Python wheels. `https://pythonwheels.com/`.

[46] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2187–2200. ACM, 2017.

[47] Tapajit Dey and Audris Mockus. Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 66–69. ACM, 2018.

[48] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*, 2021.

[49] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 302–313. ACM, 2019.

[50] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (AST)*. ACM, 2016.

[51] Thomas Durieux and Martin Monperrus. Introclassjava: A benchmark of 297 small java programs for automatic repair. Technical Report #hal-01272126, University of Lille, 2016.

[52] Elsevier. Credit author statement. `https://www.elsevier.com/authors/policies-and-guidelines/credit-author-statement`.

[53] Elsevier. Scopus expertly curated abstract & citation database. `https://www.elsevier.com/solutions/scopus`.

[54] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, 2021.

[55] Python Software Foundation. socket — low-level networking interface. `https://docs.python.org/3/library/socket.html#module-socket`.

[56] Stichting Cuckoo Foundation. cuckoo: Automated malware analysis. `https://cuckoosandbox.org/`.

[57] The Apache Software Foundation. Introduction to the pom. `https://maven.apache.org/guides/introduction/introduction-to-the-pom.html`.

[58] The Linux Foundation. Open source security metrics. `https://metrics.openssf.org`.

[59] The Linux Foundation. Open source software supply chain security. `https://www.linuxfoundation.org/wp-content/uploads/oss_supply_chain_security.pdf`.

[60] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[61] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2017.

[62] GitHub. Glossary. https://docs.github.com/en/github/getting-started-with-github/github-glossary.

[63] Github. Where the world builds software. `https://github.com`.

[64] Gitlab. The devops platform. `https://about.gitlab.com`.

[65] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *Proceedings of the 43rd IEEE/ACM International Conference*

on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 258–267. IEEE, 2021.

[66] Leo A Goodman. Snowball sampling. *The annals of mathematical statistics*, pages 148–170, 1961.

[67] googleapis. Do not include pyc files in the pypi packages. `https://github.com/googleapis/google-auth-library-python/issues/214`.

[68] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. Investigating the reproducibility of npm packages. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 677–681. IEEE, 2020.

[69] Danny Grander and Liran Tal. A post-mortem of the malicious event-stream backdoor. `https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/`, 2018.

[70] Greenkeeper. Automated dependency management. `https://greenkeeper.io`.

[71] Robert Wayne Gregory, Mark Keil, Jan Muntermann, and Magnus Mähring. Paradoxes and the nature of ambidexterity in it transformation programs. *Information Systems Research*, 26(1):57–80, 2015.

[72] Lea Theresa Groeber, Johanna Schrader, Tamara Lopez, Sascha Fahl, and Yasemin Acar. Poster: A qualitative study on developers' security library decisions. 2018.

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

[74] Sarra Habchi, Xavier Blanc, and Romain Rouvoy. On adopting linters to deal with performance concerns in android apps. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 6–16. IEEE, 2018.

[75] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 international workshop on ecosystem architectures (WEA)*, pages 1–5. ACM, 2013.

[76] Mohanad Halaweh. Using grounded theory as a method for system requirements analysis. *JISTEM-Journal of Information Systems and Technology Management*, 9(1):23–38, 2012.

[77] Eric A Hansen and Shlomo Zilberstein. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, 126(1-2):139–157, 2001.

[78] Regina Hebig and Jesper Derehag. 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):e1863, 2017.

[79] J.I. Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis, TU Delft, 2015.

[80] Trey Herr, June Lee, William Loomis, and Stewart Scott. Breaking trust: Shades of crisis across an insecure software supply chain. `https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/`, 2020.

[81] Daniel Holth. Pep 427 – the wheel binary package format 1.0. `https://www.python.org/dev/peps/pep-0427/`, 2012.

[82] Jie Huang, Nataniel Borges, Sven Bugiel, and Michael Backes. Up-to-crash: Evaluating third-party library updatability on android. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2019.

[83] ipython. traitlets: A lightweight traits like module. `https://github.com/ipython/traitlets`.

[84] Jenkins. Build great things at any scale. `https://www.jenkins.io`.

[85] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[86] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, 2014.

[87] Maya Kaczorowski. Secure at every step: What is software supply chain security and why does it matter? `https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/`, 2020.

[88] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.

[89] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 344–353. IEEE, 2007.

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

[91] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[92] Andrey Kuyan, Sergey Gusev, Andrey Kozlov, Zhanibek Kaimuldenov, and Evgeny Kravtsunov. Experience of building and deployment debian on elbrus architecture. In *Proceedings of the 2013 Spring/Summer Young Researchers' Colloquium on Software Engineering*, 2013.

[93] Chris Lamb and Stefano Zacchiroli. Reproducible builds: Increasing the integrity of software supply chains. *IEEE Software*, 2021.

[94] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 245–256. ACM, 2020.

[95] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[96] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *Proceedings of the 2013 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pages 383–392. IEEE, 2013.

[97] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[98] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[99] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[100] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2011.

[101] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966.

[102] Kim Lewandowski and Mark Lodato. Introducing slsa, an end-to-end framework for supply chain integrity. `https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html`, 2021.

[103] Libraries.io. The open source discovery service. `https://libraries.io`.

[104] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion)*, pages 55–56. ACM, 2017.

[105] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[106] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[107] SS Jeremy Long. Owasp dependency check. `https://github.com/jeremylong/DependencyCheck`.

[108] Lutoma. Psa: There is a fake version of this package on pypi with malicious code. `https://github.com/dateutil/dateutil/issues/984`.

[109] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 381–392. IEEE, 2017.

[110] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[111] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[112] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 441–444. ACM, 2016.

[113] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Proceedings of the 10th International Symposium on Search Based Software Engineering (SSBSE)*, pages 65–86. Springer, 2018.

[114] Mark Mason. Sample size and saturation in phd studies using qualitative interviews. In *Forum qualitative Sozialforschung/Forum: qualitative social research*, volume 11, 2010.

[115] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.

[116] Heather Meeker. Open source licensing: What every technologist should know. ttps://opensource.com/article/17/9/open-source-licensing/, 2017.

[117] Microsoft. Applicationinspector: A source code analyzer. `https://github.com/microsoft/ApplicationInspector`.

[118] Microsoft. Oss find source: Attempts to locate the source code (on github, currently) of a given package. `https://github.com/microsoft/OSSGadget/wiki/OSS-Find-Source`.

[119] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94. IEEE, 2017.

[120] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[121] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering*, 23(5):2901–2947, 2018.

[122] Lucas Hermann Negri. Pypi package peakutils: Peak detection utilities for 1d data. `https://pypi.org/project/PeakUtils/`.

[123] Maximilian Nothe. Who has already dropped python 2 support? `https://maxnoe.github.io/who-dropped-python2/`.

[124] npm. Maven central: Official search by the maintainers of maven central repository. `https://search.maven.org`.

[125] npm. The package manager for node.js. `https://www.npmjs.com`.

[126] Octoverse. Top languages over the years. https://octoverse.github.com/.

[127] National Institute of Standards and Technology (NIST). Security and privacy controls for federal information systems and organizations, sp 800-53, revision 5, september 2020. `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf`, 2020. Online; accessed 21 Feb 2021.

[128] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. If you've seen one, you've seen them all: Leveraging ast clustering using mcl to mimic expertise to detect software supply chain attacks. *arXiv preprint arXiv:2011.02235*, 2020.

[129] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 23–43. Springer, 2020.

[130] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*, pages 1–6. ACM, 2020.

[131] ossf. criticality_score: Open source project criticality score. `https://github.com/ossf/criticality_score`.

[132] Amantia Pano, Daniel Graziotin, and Pekka Abrahamsson. Factors and actors leading to the adoption of a javascript framework. *Empirical Software Engineering*, 23(6):3503–3534, 2018.

[133] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. ACM, 2018.

[134] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 2020.

[135] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. Preliminary findings on foss dependencies and security: A qualitative study on developers' attitudes and experience. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 284–285. IEEE, 2020.

[136] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1513–1531. ACM, 2020.

[137] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. Code analysis tables for developers interviews on dependencies paper. Zenodo, 2021.

[138] Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*, pages 1–6. ACM, 2017.

[139] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the 26th ACM conference on Computer Supported Cooperative Work (CSCW)*, pages 1371–1380. ACM, 2012.

[140] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE, 2015.

[141] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018.

[142] Nikita Popov. Changes to git commit workflow. `https://news-web.php.net/php.internals/113838`.

[143] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

[144] PyCQA. Security oriented static analyser for python code. `https://pypi.org/project/bandit/`.

[145] PyPA. Adding new checks. `https://warehouse.pypa.io/development/malware-checks.html#id6`.

[146] PyPA. bdist_wheel is not idempotent. `https://github.com/pypa/wheel/issues/248`.

[147] PyPA. Glossary. `https://packaging.python.org/glossary/`.

[148] PyPA. An overview of packaging for python. `https://packaging.python.org/overview/`.

[149] PyPA. The package installer for python. `https://pip.pypa.io/en/stable/`.

[150] PyPA. Pypi: The python package index. `https://pypi.org`.

[151] PyPA. Setuptools build system. `https://pypi.org/project/setuptools/`.

[152] python pillow. Fix possible malware in one of the test files. `https://github.com/python-pillow/Pillow/issues/251`.

[153] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, Yudong Qi, and Chengsong Wang. Empirical effectiveness evaluation of spectra-based fault localization on automated program repair. In *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC)*, pages 828–829. IEEE, 2013.

[154] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 254–265. ACM, 2014.

[155] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 24th International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36. ACM, 2015.

[156] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*, pages 71–81, 2018.

[157] rubygems. Rubygems: your community gem host. `https://rubygems.org`.

[158] Antonino Sabetta and Michele Bezzi. A practical approach to the automatic classification of security-relevant commits. In *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 579–582. IEEE, 2018.

[159] David Saff and Michael D Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, pages 281–292. IEEE, 2003.

[160] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13. ACM, 2018.

[161] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE, 2017.

[162] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[163] SAP. fosstars: Ratings for open source projects. `https://github.com/sap/fosstars-rating-core`.

[164] sass. libsass: A c/c++ implementation of a sass compiler. `https://github.com/sass/libsass`.

[165] sass. Pypi package libsass: Sass for python: A straightforward binding of libsass for python. `https://pypi.org/project/libsass/`.

[166] Khaironi Y Sharif, Michael English, Nour Ali, Chris Exton, JJ Collins, and Jim Buckley. An empirically-based characterization and quantification of information seeking through mailing lists during open source developers' software evolution. *Information and Software Technology*, 57:77–94, 2015.

[167] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[168] Ivan Pashchenkp Simone Pirocca, Duc-Ly Vu. py2src: Automatic identification of source code repositories and factors for selecting new pypi packages. `https://github.com/simonepirocca/py2src`.

[169] Slovak. skcsirt-sa-20170909-pypi. `https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/`.

[170] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 532–543. ACM, 2015.

[171] Snyk. How snyk finds out about new vulnerabilities. `https://support.snyk.io/hc/en-us/articles/360003923877-How-Snyk-finds-out-about-new-vulnerabilities`.

[172] Snyk. Vulnerability db. `https://snyk.io/vuln/`.

[173] sonarqube. sonarqube: Your teammate for code quality and code security. `https://www.sonarqube.org`.

[174] Sonatype. State of the software supply chain report reveals best practices from 36,000 open source software development teams. `https://www.sonatype.com/press-release-blog/2019-state-of-the-software-supply-chain-report-reveals-best-practices-from-36000-open-source-software-development-teams`, 2019.

[175] Sourceforge. Sourceforge: The complete open-source and business software platform. `https://sourceforge.net`.

[176] Steve Stagg. Building a botnet on pypi. `https://hackernoon.com/building-a-botnet-on-pypi-be1ad280b8d6`, 2017.

[177] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*. Sage, 1990.

[178] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *IEA/AIE*, pages 155–160, 1996.

[179] Swagger. Swagger codegen: Api code & client generator. `https://swagger.io/tools/swagger-codegen/`.

[180] Synopsys. Open source security and risk analysis report. `https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf`, 2020.

[181] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Spellbound: Defending against package typosquatting. *arXiv preprint*, 2020.

[182] TIOBE. Tiobe index for january 2022. https://www.tiobe.com/tiobe-index/.

[183] Santiago Torres-Arias. *In-toto: Practical Software Supply Chain Security*. PhD thesis, New York University Tandon School of Engineering, 2020.

[184] Travis. Travis ci: Test and deploy your code with confidence. `https://travis-ci.org`,.

[185] Nikolai Philipp Tschacher. *Typosquatting in programming language package managers*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2016.

[186] Hataichanok Unphon and Yvonne Dittrich. Software architecture awareness in long-term software product evolution. *Journal of Systems and Software*, 83(11):2211–2226, 2010.

[187] urllib3. Pypi package urllib3: Http library with thread-safe connection pooling, file post, and more. `https://pypi.org/project/urllib3/`.

[188] urllib3. urllib3: Documentation. `https://urllib3.readthedocs.io/en/latest/`.

[189] urllib3. urllib3: Http library with thread-safe connection pooling, file post, and more. `https://pypi.org/project/urllib3/`.

[190] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 644–655. ACM, 2018.

[191] Dirk van der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. Schrödinger's security: opening the box on app developers' security rationale. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 149–160. IEEE, 2020.

[192] Hugo van Kemenade. Top pypi packages. `https://doi.org/10.5281/zenodo.4486832`, 2021.

[193] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. Context is king: The developer perspective on the usage of static analysis tools. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49. IEEE, 2018.

[194] Adrian Vladu. Python on windows arm64. `https://cloudbase.it/python-on-windows-arm64/`, 2021.

[195] Laurie Voss. npm and the future of javascript. `https://slides.com/seldo/npm-future-of-javascript`, 2018.

[196] Duc-Ly Vu. A fork of bandit tool with patterns to identifying malicious python code. `https://github.com/lyvd/bandit4mal`.

[197] Duc-Ly Vu. Lastpymile: An approach for identifying the discrenpacy between packages and sources. `https://github.com/assuremoss/lastpymile`.

[198] Duc-Ly Vu. py2src: Towards the automatic (and reliable) identification of sources for pypi package. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1394–1396. IEEE, 2021.

[199] Duc-Ly Vu, Ivan Pashchenko, and Fabio Massacci. Please hold on: more time= more patches? automated program repair as anytime algorithms. In *Proceedings of the 2nd IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2021.

[200] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2093–2095. ACM, 2020.

[201] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Typosquatting and combosquatting attacks on the python ecosystem. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020.

[202] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021.

[203] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Lastpymile replication package. `https://zenodo.org/record/4899935#.Yb46-C8w1QI`, 2021.

[204] Warehouse. Malware checks. `https://warehouse.readthedocs.io/development/malware-checks/#malware-checks`.

[205] Warehouse. Malware checks pakcage turnover patterns. `https://github.com/pypa/warehouse/tree/master/warehouse/malware/checks/package_turnover`.

[206] Warehouse. Malware checks setup patterns. `https://github.com/pypa/warehouse/tree/master/warehouse/malware/checks/setup_patterns`.

[207] Warehouse. Warehouse codebase. `https://warehouse.readthedocs.io/application.html`.

[208] Joshua Wehner. Working with submodules. `https://github.blog/2016-02-01-working-with-submodules/`, 2016.

[209] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

[210] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490. IEEE, 2019.

[211] wiki.python.org. Web frameworks. `https://wiki.python.org/moin/WebFrameworks`.

[212] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, pages 1–26, 2012.

[213] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 351–361. ACM, 2016.

[214] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[215] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol:

Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1), 2016.

[216] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Proceedings of the 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[217] Robert K Yin. *Qualitative research from start to finish.* Guilford publications, 2015.

[218] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2018.

[219] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the diversity of software package popularity metrics: An empirical study of npm. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 589–593. IEEE, 2019.

[220] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3), 1996.

[221] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, pages 995–1010. USENIX, 2019.