Mitigation of Vulnerabilities and Incompatibility in Open-source Ecosystem

Lyuye Zhang

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

20/03/2024

. . . .

Date

. . . .

Lyuye Zhang

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

21/03/2024

. . . .

Date | Prof. Yang Liu

Authorship Attribution Statement

Please select one of the following; *delete as appropriate: This thesis contains material from 4 papers [1–4] accepted at conferences in which I am listed as an author.

Chapter 4 is published as Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556956.

URL https://doi.org/10.1145/3551349.3556956.

The contributions of the co-authors are as follows:

- Prof. Liu provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was substantially revised by Dr Xu.
- I designed the algorithm and finished the implementation. I also prepared and analyzed the evaluation data.
- A/Prof. Fan, A/Prof. S. Chen, A/Prof. B, Chen, and Dr. C. Liu assisted in revising the manuscript.

Chapter 5 is published as Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. Compatible remediation on vulnerabilities from third-party libraries for java projects. In Proceedings of the 45th International Conference on Software Engineering, ICSE '23, page 2540–2552. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00212.URL https://doi.org/10.1109/ICSE48619.2023.00212.

The contributions of the co-authors are as follows:

- Prof. Liu provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts. The manuscript was revised by Dr. C. Liu and A/Prof. Chen.
- I designed the algorithm and finished the implementation. I also analyzed the evaluation data.
- Dr. Xu and A/Prof Fan revised the manuscript.
- Mr. Zhao and Mr. Wu assisted in labeling the original dataset.

Chapter 6 is published as Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 191–203. IEEE Computer Society, 2023

The contributions of the co-authors are as follows:

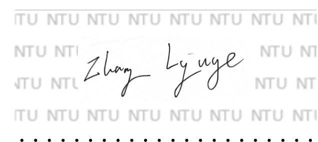
- Prof. Liu provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts.
- I designed the algorithm and finished the implementation. I also prepared and analyzed the evaluation data.
- A/Prof. Fan, A/Prof. S. Chen, and Dr Xu assisted in revising the manuscript.
- Mr. Zhao and Mr. Y. Zhang assisted in labeling the original dataset.

Chapter 7 is published as Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Yang Liu, Song Huang. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In Proceedings of the 46th International Conference on Software Engineering, ICSE '24.

The contributions of the co-authors are as follows:

- Prof. Liu and Prof. Huang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts.
- Dr. Hu and I wrote the scripts for the empirical study and analyzed the derived data.
- Dr. C. Liu and Dr. Yang assisted in revising the manuscript.

20/03/2024



. . . .

Date Lyuye Zhang

Acknowledgements

The first person I would like to thank is my Ph.D. supervisor, Prof. Liu Yang. His invaluable guidance and unwavering support have been instrumental throughout my research journey. Prof. Liu's deep passion for conducting meaningful research has been truly inspiring, and his boundless curiosity for the unknown has continually motivated me to explore new horizons. I will never forget the night my paper was rejected, and he stayed up until 1 a.m. discussing how I could enhance my storytelling and strengthen the narrative of my work. Prof. Liu's unique perspective on the future of research deeply impresses me. His ability to identify critical issues and foresee emerging trends has consistently steered our team toward cutting-edge research. Finally, I am profoundly influenced by his commitment to making research impactful and translatable to practical applications. This principle has become one of my primary goals in pursuing research today. For all this and more, I am deeply grateful for his mentorship and support.

I would also like to express my heartfelt gratitude to Dr. Liu Chengwei, a senior colleague who has been an incredible mentor throughout my Ph.D. journey. From my very first research project to my oral defence slide revision and the most recent work, he has been a constant source of guidance and inspiration. For countless problems I encountered along the way, Dr. Liu has always been my go-to person. His unwavering support and approachability have been instrumental in my growth. Through his patience and long-term mentorship, I have learned so much—ranging from writing and debugging code to orchestrating independent research papers. I deeply appreciate his tireless 24/7 support. I would also like to express my sincere appreciation to Dr. Xu Zhengzi, whose tremendous support and encouragement have been invaluable to my research journey. Dr. Xu possesses exceptional writing skills, from which I have learned a great deal. His feedback and insights have significantly contributed to improving my ability to articulate ideas effectively.

I would also like to extend my heartfelt gratitude to Prof. Chen Sen, Prof. Fan Lingling, and Prof. Chen Bihuan for their invaluable guidance and support throughout

my research journey. Their insightful discussions and feedback on critical issues have been instrumental in shaping work I undertook. Their rigor and dedication to research have taught me how to think critically, especially from the perspective of a reviewer. I deeply appreciate the time and effort they have devoted to mentoring me and their meaningful contributions to my growth as a researcher.

I would also like to thank my coauthors, Prof. Hu jinchang, Mr. Wu Jiahui, Mr. Zhao Lida, Mr. Zhang Yiran, Mr. Sun Yuqiang, Ms. Sun Kairan, Dr. Wu Daoyuan, Dr. Liu Ye, Dr. Zhangjian, Dr. Wang Chong, Dr. Zheng Yaowen, Dr. Sun Xiaoyu, and Dr. Tian Haoye, who have been very helpful for orchistrating my papers regarding the ideation, methodology, and experiment designs. The last but not the least, Mr. Kaixuan Li who have been pretty supportive during my down time.

I would also appreciate the support from professors in NTU and my colleagues, Prof. Li Yi, Prof. Zhang Tianwei, Dr. Shi Ling, Dr. Wu Yueming, Dr. Li Yuekang, Dr. Zhang Cen, Dr. Liu Yi, Mr. Li Tianlin, Dr. Li Zhiming, Mr. Wu Bozhi, Mr. Huang Yifan, Mr. Sun Dianxiang, Dr. Nie Liming, Ms. Wu Fenghua, Dr. Ge Jingquan, Ms. Wang Qingwen, Mr. Xu Xiufeng, Dr. Liu Han, Dr. Lou Xiaoxuan, Dr. Chen Kangjie. The long-term support from Mr. Tan Suanhai has also been indispensable for my research.

Lastly, I want to thank my parents for their unconditional love and support, which have been my greatest source of strength throughout my Ph.D. journey and my life. I love you both deeply.

"A person who never made a mistake never tried anything new."

—Einstein, Albert

To my dear family whose endless love and support have been my strongest foundation.

Abstract

The rapid development of Open-source software (OSS) Ecosystem enhances the efficiency of software development by providing Third-party libraries (TPLs) for developers to avoid re-inventing the wheels. However, the usage of TPLs sometimes introduces vulnerabilities, as the TPLs may have security loopholes that could be exploited by attackers. Regarding the security aspect of OSS, researchers have made tremendous advancements by studying, detecting, evaluating, and fixing vulnerabilities within OSS ecosystem and TPLs. Towards a software project that leverages TPLs, Software Composition Analysis (SCA) has been proposed to detect TPLs and the potential vulnerabilities associated with them. By reporting the vulnerabilities to users, SCA serves as an inspector for TPLs used within software projects. Additionally, SCA tools are responsible for providing suggestions to mitigate the security risks, called remediation, such as adjusting the versions of the open-source libraries. We take Maven as an example, which is a popular package manager for Java projects, to illustrate the incompatibility and vulenrability issues within individual projects and the entire ecosystem. Having explored the Maven ecosystem from micro and macro perspectives, we propose a series of tools to address the issues. Then I delved into another package manager, Golang, to study the vulnerability life cycles in the Golang ecosystem.

For an individual Maven projects, upgrading the versions of its dependencies helps reduce the risks of vulenrability. However, the adjusted dependencies by remediation inevitably introduce incompatibility risks. Modern SCA tools fail to consider the comprehensive compatibility risks while adjusting the versions of libraries. Some cause syntactic issues due to the missing signatures of symbols or methods. Other incompatibilities can cause different behaviors which lead to abnormal execution or even crashes. The syntactic issues can be detected by compilation and existing API checking tools. But the semantic breaking issues caused by inconsistent behaviors can only be revealed by tests that are limited by the coverage. To bridge the gap, I propose a Semantic Breaking Issue Detector (*Sembid*) to statically detect the semantic issue across upgrades over exposed APIs. The detector is

able to efficiently detect APIs affected by semantic breaking between two versions for individual open-source libraries.

Besides the compatibility issues, OSS could be susceptible to software vulnerabilities. SCA tools have been developed to detect and remediate such vulnerabilities. Nontheless, modern SCA tools only provide individual suggestions on how to resolve the vulnerabilities without a holistic solution for global optimization of all dependencies. Upgrading to secure versions is not always straightforward considering that versions are affected by various vulnerabilities and the upgrades may result in new threats, such as the incompatibility risks. To achieve the global optimization and ensure the usable dependency graph, we propose Compatible Remediation of Third-party libraries (*Coral*) that accurately wipes out vulnerabilities without affecting benign dependencies with incompatibility issues. *Coral* takes in an initial dependency graph generated by the Maven client tool and returns the secure and compatible dependency configurations.

Integrated *Sembid*, *Coral* can detect all types of incompatibilities to further enhance the usability of remediation within individual projects. But for the entire Maven ecosystem beyond individual libraries or projects, remediating the vulnerabilities presents another set of challenges, beyond the scope of SCA. From the study of the vulnerability propagation and evolution in the Maven ecosystem, I found vulnerabilities could be persistent in the ecosystem even after patch versions are released. This issue persists within the Maven ecosystem due to its inherent design limitations, and it will not be resolved simply through the ecosystem's natural evolution. Hence, I conducted an empirical study to quantitatively evaluate the prevalence and root causes of persistent vulnerabilities. It turned out that the inflexible fixed dependency version specification blocks the propagation of patches along dependency paths. Accordingly, I proposed a tool called *Ranger* to restore version ranges from inflexible fixed versions and facilitate the automatic remediation of vulnerabilities in the ecosystem. The evaluation substantiates that *Ranger* could automatically remediate 90% of vulnerabilities within downstream libraries while ensuring compatibility.

Besides legacy package manager like Maven, a package manager, Go Module, introduces brand-new mechanisms to resolve the legacy issues that occur in legacy ecosystems. However, benefits come with a price. As a pioneer decentralized package manager, Go Module has implemented several new features, such as allowing

fixing commits instead of traditional versions as the dependency version specification and library version indexing systems for decentralized released libraries. These new implementations introduced potential lags of pushing patches of the vulnerable to downstream users, which is referred to as fixing lag. I conducted an empirical study to evaluate how Golang mechanisms affected the life cycles of vulnerabilities in the Golang ecosystem. I developed an algorithm to quantitatively model the life cycles of vulnerabilities. After locating the lagged vulnerabilities, I also submitted the inquiry about the reasons for the lags to the library maintainers. From the study and inquiries, interesting finds and insights were obtained.

Contents

Acknowledgements ix

Abstract xiii

List of Figures xxiii

List of Tables xxv

Symbols and Acronyms xxvii

1 Introduction 1
- 1.1 Overview . . . 2
  - 1.1.1 Semantic Breaking Issue Detector . . . 2
  - 1.1.2 Compatible Remediation of OSS Vulnerabilities . . . 3
  - 1.1.3 Persistent Vulnerabilities in Maven . . . 4
  - 1.1.4 Study of Vulnerability Life Cycle . . . 4
- 1.2 Major Contributions . . . 5
- 1.3 Organization of the Thesis . . . 6

2 Background 7
- 2.1 Terminology . . . 7
- 2.2 Background of OSS Security and Maintenance . . . 8
  - 2.2.1 Open-source Software . . . 8
  - 2.2.2 Software Compostion Analysis . . . 9
  - 2.2.3 OSS Remediation . . . 9
  - 2.2.4 Compatibility Definitions . . . 10
  - 2.2.5 Incompatibility Risks . . . 10
  - 2.2.6 Remediation of Existing Tools . . . 11
  - 2.2.7 Semantic Breaking Detection . . . 12
  - 2.2.8 Vulnerability Propagation in Maven Ecosystem . . . 12
  - 2.2.9 Vulnerability Patching Lag in Golang Ecosystem . . . 13
  - 2.2.10 Introduction on Concepts and Tools . . . 13

3 Literature Review 17

3.1 SCA Remediation . . . 17
3.1.1 **SCA remediation tools** . . . 17
3.1.2 **Component and vulnerability detection** . . . 18
3.1.3 SCA Application in OSS Ecosystem . . . 18
3.2 Software Compatibility and Semantic Versioning . . . 19
3.2.1 Study of Semantic Versioning Compliance . . . 19
3.2.2 API Compatibility Checking . . . 19
3.2.3 Behavior Similarity of Java Programs . . . 20
3.3 Vulnerability Propagation in Maven . . . 20
3.3.1 Persistence of Vulnerabilities . . . 20
3.3.2 Remediation for Maven Vulnerabilities . . . 21
3.3.3 Dependency Versioning in Modern Ecosystem . . . 21
3.4 Empirical Study for OSS Ecosystem . . . 22
3.4.1 Vulnerability Analysis in OSS Ecosystem . . . 22
3.4.2 Technical Lag Studies . . . 23

4 Semantic Breaking Detection **25**
4.1 Overview . . . 25
4.2 Background and Motivation . . . 26
4.2.1 Semantic Versioning Rules . . . 26
4.2.2 Motivating Example . . . 26
4.3 Empirical Study . . . 27
4.3.1 Study of Root Causes of Semantic Breaking . . . 27
4.3.1.1 **Causes of SemB** . . . 27
4.3.1.2 **What Changes Will NOT Cause SemB?** . . . 28
4.3.2 Study of Benign Changes . . . 29
4.3.2.1 **Bug Fixes (393 cases) Categories** . . . 30
4.3.2.2 **New Functionality (191 cases) Categories** . . . 31
4.4 Methodology . . . 31
4.4.1 Grouping Clusters from Call Graphs . . . 32
4.4.1.1 **Generating Call Graphs for Candidate APIs** 32
4.4.1.2 **Grouping Clusters** . . . 33
4.4.2 Deriving Dependencies Summaries . . . 33
4.4.2.1 **Data Dependencies Summary** . . . 34
4.4.2.2 **Control Dependency Summary** . . . 34
4.4.2.3 **Exception Summary** . . . 36
4.4.3 Matching Patterns for Benign Changes . . . 36
4.4.4 **Measuring Semantic Diff** . . . 38
4.4.5 Checking Impact of Semantic Breaking . . . 39
4.4.5.1 **Verifying Triggerability** . . . 40
4.4.5.2 **Verifying Propagatability** . . . 41
4.5 Evaluation . . . 41
4.5.1 Evaluation Setup . . . 42

4.5.2 RQ1: SemB Detection Accuracy 43
4.5.3 RQ2: Effectiveness of Detecting SemB against Unit Tests . . 46
4.5.4 RQ3: Study of Compliance with SemVer 47
4.6 Threats to Validity 51
4.7 Limitations 52
4.8 Conclusion of Semantic Breaking Detection 52

5 Compatible Remediation of Vulnerabilities 53
5.1 Overview 53
5.2 Motivating Example 54
5.3 Preliminary Studies 55
5.3.1 Study of Remediation Strategies of Existing Tools 55
5.3.2 Study of Users' Concerns towards Remediation Suggestions . 57
5.4 Methodology 59
5.4.1 Problem Formulation 59
5.4.2 Overview 60
5.4.3 Constructing Dependency Graph and Call Graph 60
5.4.4 Partitioning Dependency Graph 61
5.4.5 Optimizing Subgraphs 62
5.4.6 Backtracking 66
5.4.6.1 **Hard Backtracking** 67
5.4.6.2 **Soft Backtracking** 67
5.5 Evaluation 68
5.5.1 Preparation 68
5.5.1.1 **Data Collection** 68
5.5.1.2 **Tools and Environments Preparation** 70
5.5.2 RQ1: Comparison with Other Remediation Tools 71
5.5.2.1 **Evaluation Metrics** 71
5.5.2.2 **Comparison Results** 71
5.5.3 RQ2: Effectiveness of Improvement on Global Optimization 74
5.5.4 RQ3: How many fixable/unfixable CVEs in Maven 76
5.5.4.1 **Preparation of data** 76
5.5.4.2 Results of RQ3 77
5.6 Threats of Validity 78
5.7 Conclusion 79

6 Persistent Vulnerability Study 81
6.1 Overview 81
6.2 Preparation for Empirical Study 81
6.2.1 Background of SemVer in Maven 82
6.2.2 Infrastructure of Study 82
6.2.2.1 Dependency Graph for Maven 82
6.2.2.2 Search Algorithm 82

6.3 Empirical Study . . . 84
6.3.1 RQ1: Analysis of Persistent Vulnerabilities . . . 84
6.3.1.1 Log4Shell Analysis . . . 85
6.3.1.2 Other Java Vulnerability Analysis . . . 87
6.3.2 RQ2: Study of Underlying Causes . . . 89
6.3.2.1 Distribution of the causes . . . 89
6.4 Methodology and Evaluation . . . 92
6.4.1 RQ3: Review of Existing Solutions . . . 92
6.4.1.1 Study of the Usage of Ranges . . . 93
6.4.1.2 Study of the Version Overidding . . . 94
6.4.2 RQ4: Methodology and Evaluation of *Ranger* . . . 95
6.4.2.1 Requirements of the Solution . . . 95
6.4.2.2 Design of *Ranger* . . . 96
6.4.2.3 Evaluation of *Ranger* . . . 99
6.5 Discussion . . . 102
6.6 Threats of Validity . . . 103
6.7 Conclusion . . . 104

7 Golang Vulnerability Life Cycle Analysis 105
7.1 Overview . . . 105
7.2 Background and Motivating Example . . . 105
7.2.1 Background . . . 105
7.2.2 Motivating Example . . . 106
7.3 Methodology . . . 107
7.3.1 Analytical Infrastructure Construction . . . 107
7.3.1.1 **Vulnerability** . . . 108
7.3.1.2 **Dependency Relation** . . . 108
7.3.1.3 **Indexing time of version** . . . 108
7.3.1.4 **Commit history** . . . 108
7.3.2 Fixing Lag Analysis . . . 109
7.3.2.1 **Lead Time** . . . 109
7.3.2.2 **Version Lag** . . . 110
7.3.2.3 **Index Lag** . . . 110
7.3.2.4 **Dependent Fix Time** . . . 110
7.4 Empirical Study . . . 111
7.4.1 RQ1: Vulnerability Impact Analysis . . . 113
7.4.1.1 **Data Preparation** . . . 113
7.4.1.2 **Impact of Golang Vulnerabilities** . . . 113
7.4.2 RQ2: Patch Lagging Analysis . . . 115
7.4.2.1 **Data Preparation** . . . 115
7.4.2.2 **Analysis of Lag of Version** . . . 116
7.4.2.3 **Analysis of Lag of Index** . . . 118
7.4.3 RQ3: Dependents Vulnerability Fixing . . . 119

7.4.3.1 **Data Preparation** 119
7.4.3.2 **Analysis of Fixes by Dependents** 120
7.4.4 RQ4: Fixing Lagging Inquiry 122
7.4.4.1 **Absent Patch Version Releases** 123
7.4.4.2 **Delayed Patch Version Releases** 124
7.4.4.3 **Not Pushing Patch Version to Go Module Index** 125
7.4.4.4 **Not Fixing the Vulnerability** 127
7.5 Discussion 127
7.5.1 Recommendations 127
7.5.2 Limitations and Threats to Validity 129
7.6 Conclusion 130

8 Future Work 131

9 Conclusion 135

List of Author's Awards, Patents, and Publications 137

Bibliography 139

List of Figures

1.1 The Overview of the Thesis . . . 3

4.1 Overview of *Sembid* . . . 32
4.2 (a) Data & Control Dependencies Summaries in the semantic graph of Motivating Example (b) Example of subgraph comparison of Node *var==1* at height 1 and 2 . . . 35
4.3 Triggerabitliy and Propagatability Analysis . . . 40
4.4 Comparison between Unit Tests and *Sembid* Over All APIs During Upgrades As Well As the Ground Truth . . . 47
4.5 Proportions of Version Pairs Affected by SemB and SynB of *Patch* and *Minor* Upgrades . . . 48
4.6 Average Number and Percentage of APIs Over Three Upgrades . . . 48

5.1 Common-lang3 build failure with *Dependabot* remediation . . . 54
5.2 Overview of *Coral* . . . 60
5.3 Dependency Graph Vertical (a) and Horizontal (b) Partitioning . . . 61
5.4 Example of the Version Selection of a Dependency . . . 64
5.5 Demographics of the Data Set of RQ1 and RQ2 . . . 70
5.6 Ascending Order by Numbers of CVEs of Original per Project . . . 74
5.7 Time Consumption of *Baseline C* and *Coral* . . . 75
5.8 Distributions of Fixable/Unfixable Vulnerabilities . . . 77

6.1 Heatmap of Proportion of Affected Libraries by Log4Shell . . . 85
6.2 Accumulated Affected and Patched Libraries for Log4Shell . . . 87
6.3 Distributions of Normalized Half-lives and New Release Span . . . 89
6.4 Scenarios of Different Causes . . . 90
6.5 Proportions of Each Cause . . . 90
6.6 Usage of Vulnerability Related Version Ranges . . . 93
6.7 Overview of *Ranger* . . . 99
6.8 Number of Vulnerable Lib-vers over Months after Applying *Ranger* to Dependents at 1-10 Depths . . . 100

7.1 Overview of the Empircal Study . . . 108
7.2 Fixing Lag . . . 109
7.3 Distribution of Affected Depts . . . 114
7.4 Impact over Time of Vulnerabilities of 2019 and Before . . . 114

7.5 Distribution of LT_{ver} and Lag_{ver}[1] 117
7.6 Distribution of Lag_{index} 118
7.7 Distribution of Vulnerability-Dependent 120
7.8 Distribution of Vulnerability-Dependents 121

8.1 Roadmap of Future Work 132

List of Tables

4.1 Benchmark Accuracy of SemB Detection based on APIs 44
4.2 Execution Time per API (ms) 49

5.1 Comparison of State-of-the-art SCA Tools Providing Remediation . 55
5.2 Comparison of *Coral* among State-of-the-art Remediation Tools . . 73

6.1 Counts of POMs with *dependencyManagement* 95
6.2 Results of *Ranger* 100

Symbols and Acronyms

Symbols

$\mathcal{R}^n$	the n-dimensional Euclidean space
$\mathcal{H}$	the Euclidean space
$\Vert \cdot \Vert$	the 2-norm of a vector or matrix in Euclidean space
$\Vert \cdot \Vert_G$	the induced norm of a vector in G-space
$\Vert \cdot \Vert_E$	the induced norm of a vector or matrix in probabilistic space
$\odot$	the Hadamard (component-wise) product
$\otimes$	the Kronecker product
$\langle \cdot, \cdot \rangle$	the inner product of two vectors
$\circ$	the composition of functions
∇f	the gradient vector
$\mathcal{C}^k$	the function with continuous partial derivatives up to k orders
$x_{i,k}$	the i-th component of a vector x at time k
$\bar{x}$	the vector with the average of all components of x as each element
$\mathbf{1}$	all-ones column vector with proper dimension
$\mathcal{C}$	the average space, i.e., $span\{\mathbf{1}\}$
$\mathcal{C}^\perp$	the disagreement space, i.e., $span^\perp\{\mathbf{1}\}$
$\Pi_\Vert$	the projection matrix to the average space $\mathcal{C}$
$\Pi_\perp$	the projection matrix to the disagreement space $\mathcal{C}^\perp$
$O(\cdot)$	order of magnitude or ergodic convergence rate (running average)
$o(\cdot)$	non-ergodic convergence rate
$\mathcal{N}_i$	the index set of the neighbors of agent i

Acronyms

OSS	Open-source Software
TPL	Third-party library
SCA	Software Composition Analysis
SemB	Semantic Breaking
SynB	Syntactic Breaking
Sembid	Semantic Breaking Issue Detector
PR	Pull Request
API	Application Programming Interface
PM	Package Manager
NVD	National Vulnerability Database
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
CPE	Common Platform Enumerations
CVSS	Common Vulnerability Scoring System
DG	Dependency Graph
DC	Dependency Conflict
NPM	Node Package Manager
SemVer	Semantic Versioning
Pypi	Python Package Index
CFG	Control Flow Graph
DFG	Data Flow Graph
PDG	Program Dependence Graph
CG	Call Graph
DDS	Data Dependency Summary
CDS	Control Dependency Summary
ES	Exception Summary
SSA	Static Single Assignment
IR	Intermediate Representation
DBS	Disjunctive Boolean Summary
WL	Weisfeiler-Lehman
LOC	Line of Code

TN	True Negative
TP	True Positive
FN	False Negative
FP	False Positive
BCEL	Byte Code Engineering Library
SoftVer	Soft Version Constraints
MCR	Maven Central Repository
POM	Project Object Mode
BFS	Breadth-First Search
NRS	New Release Span
LT	Lead Time
VD	Vulnerability-Dependent
SAST	Static Application Security Testing

Chapter 1

Introduction

Open-source software (OSS) is a cornerstone of contemporary software development, instrumental in avoiding redundant efforts by serving as dependencies in myriad projects. The vast and interconnected ecosystem of OSS, which encompasses billions of libraries, is sustained through global collaborative efforts. Despite its successes, this ecosystem is not immune to security vulnerabilities that lurk within its expansive network. Security breaches in these libraries can have extensive implications, adversely affecting not only the OSS community but also its myriad of end-users. Such incidents underscore a pressing challenge that demands immediate and comprehensive strategies to mitigate potential risks [5]. My research, as one of the first, confronts this challenge head-on, focusing on enhancing security and maintainability in OSS through two primary scenarios: first, by fortifying the Software Supply Chain, ensuring the meticulous management of secure, defect-free OSS for individual software projects; and second, by pioneering strategies in OSS governance to bolster the security and maintainability of the ecosystems, thereby protecting an extensive network of downstream users. The objectives of my research are twofold: firstly, to identify and address compatibility issues within software systems; and secondly, to remediate vulnerabilities that compromise their security.

1.1 Overview

For OSS library management for user applications, Software Composition Analysis (SCA) [6] has been proposed to detect and report the OSS components associated with disclosed vulnerabilities. There have been proposed many SCA tools including academic [7] and commercial ones [8–13]. Upgrading or downgrading libraries is inevitable in OSS dependency management, but the incompatibility risks brought by it may be not completely revealed. To avoid this, my first work extended to preempting and circumventing incompatibility issues that arise from remediation, particularly through upgrading. This work aimed to detect behavioral compatibility, an advanced type of compatibility—a breakthrough that not only earned a Distinguished Paper Award but also found practical application through its implementation. Besides the compatibility issues, typically, component detection and vulnerability reporting are well implemented in various SCA tools, but the suggestions to provide vulnerability remediation have no universally accepted solution. While existing tools identify problems, they fall short of offering tangible solutions for vulnerability remediation. In contrast, as illustrated in Figure 1.1, my second research work has pioneered a comprehensive methodology for resolving vulnerabilities within dependency graphs—a breakthrough that not only earned a Distinguished Paper Award but also found practical application through its implementation at a commercial company. Nonetheless, remediation often modifies an application's dependencies, potentially leading to new issues like incompatibility. From the perspective of the OSS ecosystem, addressing ecosystem security necessitates a broader approach, as not every developer can or will enforce remediation automatically. Recognizing this, my research has probed the mechanics of package managers of ecosystems, leading to the development of novel solutions designed to mitigate persistent vulnerabilities in the Maven [14] ecosystem and curtail the life cycles of vulnerabilities in the Golang [15] ecosystem. These strategies have since been operationalized through their integration as plug-ins within industry settings.

1.1.1 Semantic Breaking Issue Detector

As illustrated in the top left square of Figure 1.1, my research focused on identifying Semantic Breaking issues (behavioral incompatibilities) from OSS packages,

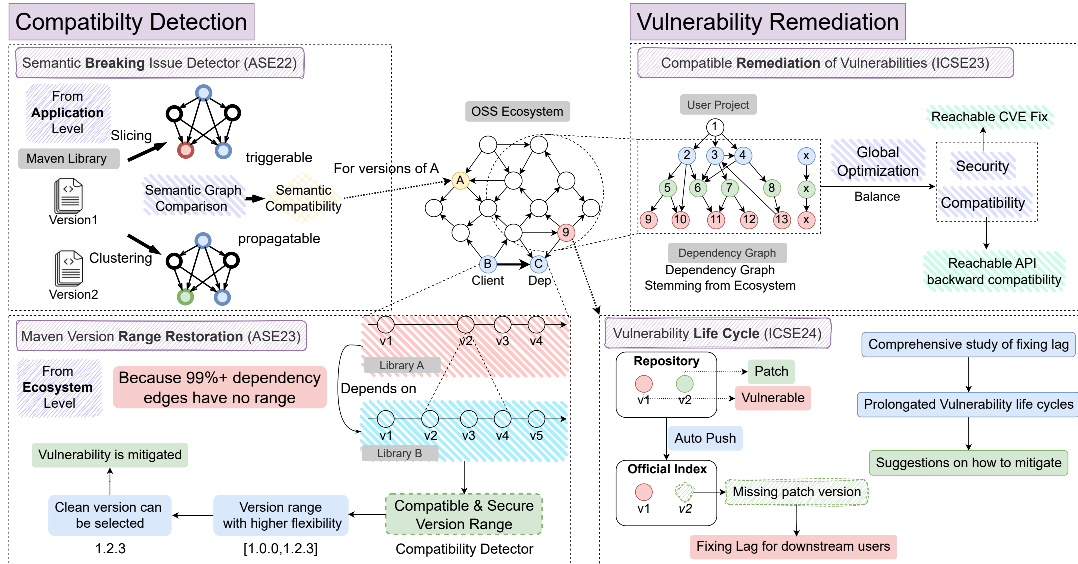


FIGURE 1.1: The Overview of the Thesis

which derived the first published tool to detect behavioral incompatibilities. Upon examining a significant number of popular packages from the ecosystem, the tool demonstrated a remarkable capacity for uncovering 23.35% more previously concealed incompatibility issues within real-world packages prevalent in the ecosystem in Section 4.5.4. Packages often undergo multiple sequential releases, potentially introducing breaking changes in later versions. Ensuring backward compatibility is crucial for the seamless use of OSS packages; otherwise, software that updates its dependencies could experience glitches or unexpected behaviors. While the academic community acknowledges the complexity of behavioral compatibility compared to the simpler syntactic compatibility, I delved deep into understanding its intricacies. By examining patterns and primary causes from real-world issues sourced from forums and issue boards, I crafted an innovative approach rooted in static analysis to detect incompatibilities among package versions. Community developers can utilize our tool as a preventive measure against introducing new incompatibilities.

1.1.2 Compatible Remediation of OSS Vulnerabilities

As illustrated in the top-right box of Figure 1.1, my research aimed at addressing vulnerabilities in individual OSS projects, ensuring that the security fixes also maintained compatibility. The process usually called remediation could help users

to smoothly handle overwhelming security alarms of their projects. We developed a remediation strategy that successfully overcame the challenges and fixed the most vulnerabilities. Our tool also found that about 78.45% of ecosystem vulnerabilities could be fixed without compromising compatibility in Section 5.5.4.2. This insight stressed the need for our tool's automated solution, given the vulnerability-prone nature of numerous packages, which if left unchecked, can jeopardize OSS package security. By remediating these vulnerabilities, we can lighten the load on maintainers and promote more widespread use of OSS packages.

1.1.3 Persistent Vulnerabilities in Maven

While most vulnerabilities are amenable to resolution, a certain subset persists within the ecosystem. My third published paper as illustrated in the bottom-left box of Figure 1.1 concentrated on the identification and mitigation of these persistent vulnerabilities, with the goal of bolstering the self-healing capabilities of OSS ecosystems. Our extensive empirical research, which encompassed 479k packages and 1,861 vulnerabilities within the Maven ecosystem [5], revealed that upwards of 50% of vulnerabilities exert a sustained impact on downstream users in Section 6.3.1. This impact, though diminishing over time, does so at a decelerating rate, suggesting these vulnerabilities may never be fully eradicated. In response, we devised a strategy aimed at curtailing the spread of vulnerabilities throughout the ecosystem. The evaluation indicated that our approach could autonomously free over 90% of downstream libraries from vulnerabilitiesin Section 6.4.2.2.

1.1.4 Study of Vulnerability Life Cycle

Shifting from the established Maven ecosystem, my fourth project pivoted to examine the vulnerability lifecycle within the young ecosystem, Golang, in the bottom-right box of Figure 1.1. Distinctively, the Golang ecosystem represents a pioneering foray into a decentralized OSS registry, a departure from previous, more centralized ecosystems. This innovative structure intrinsically results in delays in patch distribution, consequently prolonging the lifecycle of vulnerabilities within the ecosystem. Our comprehensive analysis in Section 7.4.1, encompassing roughly 1,200 disclosed vulnerabilities — the entirety of known security breaches within

Golang packages — sought to characterize the lifecycles of Golang vulnerabilities, particularly the latency of patch adoption among downstream users. Our findings indicated that approximately 30% of vulnerabilities remain unpatched due to the decentralized framework, resulting in delays ranging from one day to fourteen months in Section 7.4.2.2. To mitigate this, we advocated the implementation of an automated alert mechanism designed to promptly push newly available patches to downstream users.

1.2 Major Contributions

Our main contributions can be stated as follows:

- *Semantic Breaking Issue Detector*: I have proposed the first static detector to detect the Semantic Breaking APIs of Java libraries between a pair of versions. I have also conducted a study to reveal and categorize the root causes of SemB. Witin the popular libraries in the Maven ecosystem, I conducted an empirical study to unveil the prevalence of SemB across upgrades and found that 1.10% of APIs from *Patch* and 4.06% of APIs from *Minor* upgrades were affected by SemB. They are 2-4 times more than SynB APIs (0.38% and 1.04%). These findings could be found in Section 4.5.4.
- *Compatible Remediation of OSS Vulnerabilities*: I proposed *Coral* as a remediation tool for Maven projects to handle the global optimization for enhanced security and compatibility. I have studied the concerns of users regarding remediation suggestions by analyzing Pull Requests (PRs) and found that 51.31% of cases were related to incompatibility. I empirically compared and analyzed strategies of popular remediation tools regarding their support of compatibility and prioritization for the reference of other researchers. The evaluation and finding could be located in Section 5.3.2 and Section 5.5.2.2.
- *Persistent Vulnerabilities in Maven*: I first developed *Ranger* to restore compatible and secure version ranges which could automatically mitigate the persistent vulnerabilities in the Maven ecosystem. I conducted an empirical study to substantiate the persistence of vulnerabilities and quantitatively revealed their underlying cause and the effectiveness of countermeasures. I

further implemented a monitoring system based on an up-to-date dependency graph and a search algorithm to locate the libraries that block vulnerability fixes and suggest remediation for relevant developers and downstream users. The experimental results of *Ranger* is demonstrated in Section 6.3.1 and Section 6.4.2.2.

- *Study of Vulnerability Life Cycle*: We quantitatively substantiated that the vulnerabilities have significantly impacted the Golang ecosystem. Then we conducted a large-scale study to evaluate the impact of time lags on vulnerability life cycles in the Golang ecosystem. Finally, we interacted with maintainers and users through inquiries to gain insights into the reasons for delayed patching, and offered recommendations to potentially shorten the vulnerability life cycle as in Section 7.4.2.2.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter II introduces the background of the research work. Chapter III reviews the related research works of remediation strategies, library incompatibility studies and vulnerability analysis at the ecosystem level. Strategies of both academic and commercial remediation tools are elaborated regarding their pros and cons. The studies of incompatibilities brought by software evolution are reviewed and summarized. The insights revealed by other research work for both Maven and Golang ecosystems are discussed. Chapter IV describes the methodology of *Sembid* in detail. The methodology and evaluation setup as well as findings of *Sembid* are demonstrated. Chapter V provides the background of SCA and three types of compatibility. The role that remediation plays in the SCA is discussed. And the detailed implementation and evaluation of tool *Coral* is elaborated. Chapter VI elaborated on the empirical study of persistent vulnerabilities in Maven and the methodology of *Ranger* with the evaluation. Chapter VII delves into the details of an empirical study of the life cycles of vulnerabilities in the Golang ecosystem. Chapter VIII discusses the future works. Chapter IX summarizes the research works regarding the mitigation of vulnerabilities and incompatibility for open-source ecosystems.

Chapter 2

Background

2.1 Terminology

- **Vulnerability**: A software bug leading to security issues in the software system.
- **Open-source software (OSS)**: The publicly available libraries from the open-source community. OSS can be in any format, such as source code, bytecode, and binary.
- **Package Manager (PM)**: Package managers are organizations that collect and uniformly manage the OSS published by other developers or organizations. Usually, PMs are categorized by programming languages, such as NPM [16] for Javascript, Maven [14] for Java, Go [15] for Golang.
- **Third-party libraries (TPLs)**: The software published in package managers that could be directly used by any developers.
- **Remediation**: The operations to eliminate vulnerabilities within TPLs used in client projects. Usually, it is committed by upgrading/downgrading the versions of TPLs or applying security patches.
- **Compatibility**: The assurance that the software system returns the same output given the same input as expected after the upgrade/downgrade of TPLs. Note that only the output returned including exceptions thrown are

taken in account in this thesis, while intermediate states are not. Compatibility is classified into **Syntactic** and **Semantic** compatibility. **Syntactic** compatibility is the compatibility of the signatures of APIs, while **Semantic** compatibility is the compatibility of the internal behaviors of APIs.

The semantic versioning [17] is defined as follows:

- **Patch version Z (x.y.Z)**: *"MUST be incremented if only backward-compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior."*
- **Minor version Y (x.Y.z)**: *"MUST be incremented if new, backward-compatible functionality is introduced to the public API. It MUST be incremented if any API is deprecated."*
- **Major version X (X.y.z)**: *"MUST be incremented if any backward-incompatible changes are introduced to the public API. It MAY also include minor and patch level changes."*

2.2 Background of OSS Security and Maintenance

2.2.1 Open-source Software

To facilitate software reuse, OSS, also known as Free open-source software, has emerged and become popular in modern software development. The basic unit in OSS is often referred to as libraries, or third-party libraries (TPLs). As revealed by recent studies, the percentage of TPLs of modern software projects has been climbing [5]. Due to the wide usage of TPLs, the package manager (PM) has emerged to serve as the entry point for developers to access TPLs. PMs are usually categorized by programming languages, such as Maven [14] for Java. The Maven central repository [18] hosting reusable TPLs for developers has $471k$ TPLs and continues to update daily as of the date of thesis writing. Given the large amount of TPLs, TPLs often depend on one another to avoid re-inventing the wheel. Hence, most of these TPLs are connected with each other to form gigantic graphs with dependency relationships as edges and individual TPLs as nodes. These graphs

are usually referred to as an ecosystem. Typically, when a software project uses some TPLs, it depends on multiple other TPLs indirectly.

2.2.2 Software Compostion Analysis

To manage TPLs introduced from OSS ecosystem, Software Composition Analysis (SCA) has been proposed to accurately detect the versioned TPLs of a software project. While the metadata of TPLs is revealed, any security risks mapped to TPLs in the pre-collected database are reported to users as well. Sometimes, potential license conflicts are also reported. Typically, vulnerability mappings are pre-collected and parsed from the public vulnerability database, such as National Vulnerability Datbase [19], OSV [20], GitHub Advisory [21], and Snyk Vulnerablitiy Database [22]. Then, these vulnerabilities are mapped to OSS libraries and reported whenever the re-use of OSS libraries is detected.

2.2.3 OSS Remediation

Due to the component re-using, the TPLs are inevitably susceptible to vulnerabilities that may compromise the dependent projects that rely on the TPLs. Since TPL are open-source, the vulnerabilities are publicly visible when they are published by the National Vulnerability Database (NVD) [19], a widely used vulnerability database of open-source libraries and Operating Systems.

To avoid the exploits of vulnerabilities, the maintainers of the TPLs usually patch the vulnerabilities once they are aware of them by applying patches or upgrading the dependencies to a secure version to keep their software systems secure. The upgrading also provides feature introduction and other bug fixes to enhance the functionality of TPLs. However, the upgrading performed by users is limited to direct dependencies. Typically, a Top 10 project on Github, *seata-server* with 19 direct dependencies of commonly used Java libraries had a dependency graph (DG) with 167 libraries. These dependencies are susceptible to 140 CVEs, which makes the upgrading complex because it is difficult to fix all vulnerabilities and avoid introducing the new by only upgrading direct dependencies. Thus, a globally optimized strategy is needed to provide an optimal solution for remediation.

2.2.4 Compatibility Definitions

According to the definition of software compatibility [23], incompatible changes lead to changed output, including exceptions, given the old input. We further classify it into **Syntactic** and **Semantic** breakings. **SynB** is caused by inconsistent signatures of APIs across upgrades, which usually cause *ClassNotFoundException* and *NoSuchMethodError*. **SemB** is caused by APIs' inconsistent internal behaviors with consistent signatures. There are two major differences between SemB and SynB. First, the compilation and linking in SynB can be checked by compilers based on specific rules, but inequivalent operations in SemB are too ambiguous to be accurately identified. Second, SynB is verified by compilation and linking [24] before the execution, while SemB is mostly disclosed by runtime issues, which are more challenging to be detected without the execution. Therefore, practitioner can only locate SemB issues with regression test based on pre-defined testing. However, testing is subject to its coverage of inputs. In this thesis, we only discuss the compatibility of Java programs.

2.2.5 Incompatibility Risks

Although the upgrading is useful against vulnerabilities, it may also introduce backward-incompatible changes leading to the breaking of dependent projects. To guide the compatible upgrading, most modern dependency managers embrace semantic versioning [17], which stipulates *minor* and *patch* version upgrades should be backward compatible, but it is still difficult to guarantee that the semantic versioning rules have been strictly applied to all libraries, especially in ecosystems like Maven [14] that have evolved for decades. For example, *Hadoop-hdfs* [25], a widely-used data processing framework of Apache, was susceptible to an API incompatibility issue [26]. The issue broke many implementations of users after *patch* upgrades. Not limited to popular TPL like *Hadoop*, the API incompatibility issues commonly exist in Maven ecosystem. As revealed by Ochoa et al. [27], 20.1% of *non-major* upgrades in Maven central repository [18] have actually introduced breaking changes which could result in massive unexpected incompatibility issues

during upgrades. The incompatibility includes the signature-based issues that can be directly revealed by compilers and the incompatible behavioral issues that maintain the integrity of signatures. The signature-based incompatibility issues were widely discussed, but there is no effective way to detect the behavioral incompatibility issues. Thus, towards a secure and issue-free usage of TPLs, the developers not only need the eliminate the vulnerabilities of TPL dependencies, but also minimize the incompatibility issues brought by the upgrades. In the Java ecosystem, the assurance of compatibility is more important and harder to achieve than other languages, because according to our statistics of POM files from Maven Central Repository [18] over 99% of default dependency versions are given as a specific version instead of a range, which disables the flexibility of fixing vulnerabilities without tampering with the compatibility.

2.2.6 Remediation of Existing Tools

Remediation in open-source libraries for Java has been widely implemented in academic and commercial Software Composition Analysis (SCA) tools which target uncovering vulnerabilities and providing mitigation suggestions. Academic tools, such as steady [28], OWASP dependency check [11], and Vulcon [29] were published as open-source projects in the community. The Commercial tools, such as Dependabot [10], are not open-source so the remediation strategy is not publicly accessible. The existing tools are prone to various problems. *Steady* takes incompatibility issues into account, but it does not provide an optimal solution but a set of candidates to be selected by users. The lack of knowledge of users may result in a sub-optimal remediation. OWASP fails to return remediation results for projects with complex dependency graphs (DGs) which account for 495 out of 610 investigated most popular Java projects on GitHub. The Vulcon focuses on prioritizing the most critical vulnerabilities by the defined metrics regarding the exposure and fixing time, but it fails to provide a solution to fix the vulnerabilities dwelling in dependencies. The Commercial tools only provide the fixing of direct dependencies without changing the indirect dependencies. Thus, the remediation of commercial tools can introduce new vulnerabilities in indirect dependencies.

2.2.7 Semantic Breaking Detection

As for the compatibility assurance of the remediation, only Steady supports compatibility issues prediction out of the aforementioned tools. But it only provides the possibility of the error, *Class/Method not found*, which is only one of the issues of signature-based incompatibility. Defined by Oracle [30], signature-based compatibility refers to the Syntactic Compatibility. Without the remediation, there are existing tools proposed to detect the Syntactic Incompatibility [31–37]. They only aim at detecting the incompatibility issues caused by inconsistent signatures of methods, classes, and fields used in the dependencies during upgrades. However, they fail to detect the breaking behavioral changes, known as Semantic Breaking (SemB) for dependency upgrades. Issues induced by SemB can compromise the software system with abnormal output, regression failures, and crashes during runtime without breaking any signature-based compatibility. The current solutions regarding SemB issues rely on regression tests [38, 39] so that they share the same limitation of regression tests, which is the coverage of the test cases.

2.2.8 Vulnerability Propagation in Maven Ecosystem

Vulnerabilities in OSS libraries could propagate to downstream users. A famous library, *log4j-core*, serving as a fundamental library, swiftly responded by releasing patch updates after the exploitation of Log4Shell [40]. Downstream users have taken prompt action to adopt these patch updates, as reported by Google [41]. Despite a year's worth of advancements, Log4Shell continues to impact numerous downstream applications and persist within the Maven ecosystem, as reported by many reports and news [42–46]. Given that over 2,000 vulnerabilities from Maven libraries have been disclosed by the National Vulnerability Database (NVD) [19], it is possible that numerous other vulnerabilities persist and pose a threat to the Maven ecosystem.

Aiming for this urgent threat, many researchers [47–53] studied the vulnerability impact within the Maven ecosystem and substantiated vulnerabilities have extensively proliferated in downstream libraries. A few of them have recognized the persistence of vulnerabilities over time and provided insights into potential solutions [47–49, 54]. Wu et al. [47, 55] revealed that the reachable vulnerabilities are

more likely to be addressed. Developers' reluctance to upgrade vulnerable dependencies due to potential breaking changes has been highlighted by Pashchenko et al. [49] who also discovered that developers prioritize handling vulnerabilities in direct dependencies rather than transitive ones [56]. Moreover, Industrial standards have been proposed to promote remediation, such that OpenSSF [57] proposed the best practice guidance [58] and a tool, Scorecard [59], for developers on managing vulnerabilities in dependencies. Plumber [54] aims for persistent vulnerabilities in Node Package Manager (NPM) with limited applicability to Maven due to the rare usage of version ranges in Maven.

2.2.9 Vulnerability Patching Lag in Golang Ecosystem

As the first attempt at decentralized registries, Golang embraces the Git system to manage dependencies [60]. Specifically, unlike traditional centralized registries where developers mostly define dependencies by released version tags, Golang developers can specify package versions flexibly using individual commits. Though Golang Index [61] was introduced to mirror packages, like a centralized proxy, for convenience, there are still huge differences in broadcasting newly released versions (i.e., patch versions), between Golang and other mature ecosystems. For instance, patch versions could be available to all users, or even automatically integrated, during new installations (i.e., NPM dependency tree changes [62]). However, in the Golang ecosystem, only versions explicitly utilized by users can be cached by Golang Index according to the documentation [63]. Once they are released by maintainers in traditional registries, while according to the Golang documentation [63], only patched versions that are manually and explicitly integrated by users could be cached by Golang Index. In this case, methodologies and solutions concluded from traditional ecosystems could be compromised in Golang, and none of the existing work has yet researched the vulnerability life cycle under the special Golang dependency mechanism.

2.2.10 Introduction on Concepts and Tools

Maven: Maven [64] is a client-side tool for managing dependencies, compiling

code, and running tests in Java projects. It uses a `pom.xml` file to specify project dependencies, which Maven then resolves automatically, creating a dependency graph that helps identify conflicts by the `mvn dependency:tree` command. To compile the project, Maven uses the `mvn compile` command, which converts source code into bytecode. For testing, Maven supports frameworks like JUnit and TestNG, allowing developers to run tests with `mvn test` for unit tests or `mvn verify` for integration tests. This lifecycle automation ensures consistent builds and simplifies dependency management, making Maven essential for efficient project development and maintenance.

Maven Central Repository: The Maven Central Repository [18] is a large, publicly available repository where developers can find and download millions of libraries and dependencies for their Java projects. When a Maven project specifies dependencies in the `pom.xml` file, Maven automatically checks the Central Repository to download the specified libraries, ensuring that developers have easy access to up-to-date, widely used packages. If a dependency is not available locally, Maven retrieves it from the Central Repository and caches it, which speeds up future builds. This repository is crucial for simplifying dependency management and enabling efficient, consistent builds across different development environments.

Maven Dependency Versioning: In Maven, versioning in dependencies allows projects to specify the exact version of each library they rely on, ensuring compatibility and stability. When a dependency version is defined in the `pom.xml` file, Maven will retrieve that specific version from its repositories. The version can be a range or a single version tag. For the version range in use, Maven can have more flexibility of version resolution as any version in the range is allowed.But for the individual version tag, Maven must resolve the version exactly unless a conflict occurs. If no version is specified, Maven may default to the latest version, depending on the configuration. Version conflicts can arise when different dependencies require different versions of the same library; in such cases, Maven uses a "nearest-wins" strategy, prioritizing the version closest to the project in the dependency hierarchy. This versioning control helps maintain consistent builds and minimizes compatibility issues across environments. However, users can override this default behavior using the `<dependencyManagement>` section in the `pom.xml` file. This section lets you specify exact versions for dependencies, ensuring consistent versions across the project, even for transitive dependencies, thus preventing potential

compatibility issues.

Go Module: Go modules [65] manage dependencies in Go projects, enabling version control and dependency tracking directly in the source code. A go.mod file defines the project's dependencies and their versions, which Go automatically resolves and downloads. To fetch and install all dependencies, use the command `go mod tidy`, which removes unused dependencies and adds any missing ones. For compiling the project, the `go build` command generates an executable, and `go test` runs test files to verify code functionality. Go modules streamline dependency management, build consistency, and testing, making them essential for effective Go project development.

Chapter 3

Literature Review

3.1 SCA Remediation

SCA has been a popular research topic in recent years. Researchers have invested much effort in studying and improving the two major procedures: component and vulnerability detection and vulnerability remediation. In the following subsections, the discussion is divided into three parts: SCA remediation tools, component and vulnerability detection, and the study of the open-source software ecosystem.

3.1.1 SCA remediation tools

A limited number of research works [28, 66–69] attempted to study and enhance the remediation strategy. Alfadel et al. [66] found for the Javascript projects at Github 34.58% of PRs created by *Dependabot* [70] were not merged due to five reasons: (1) Duplication (2) Dependency conflict by peer requirements (3) Test failures (4) Internal errors (5) Disobeying rules/standards, which substantiate our findings in Section 5.3.2. *Steady* [28, 68] has been developed for years to be a code-centric and usage-based SCA tool, which has been proved effective by Imtiaz et al. [71]. Soto et al. [69] found 22.6% of upgrades by *Dependabot* were recommended for bloated dependencies. 22.6% does not contradict our result 3.92% because 22.6% consists of all bloated dependencies, while ours were only those found and addressed. These works except for *Steady* mostly focused on the evaluation of remediation tools, which left a blank of remediation strategy enhancement filled by *Coral*.

3.1.2 Component and vulnerability detection

Many researchers and practitioners [11–13, 56, 67, 71? ? –113] have studied the component and vulnerability detection. Imtiaz et al. [71] studied 9 commercial SCA tools and found the reported vulnerabilities vary substantially, which revealed that the vulnerability database was the key differentiator. Dann et al. [72] reviewed six commercial and academic SCA tools regarding their ability to handle the dependency modification types. By testing 7k+ Java projects, they found the re-bundle modification in Maven dependencies was not supported by any tools. *Vuln4real* [56, 67] was proposed to exclude the false alarms of vulnerabilities by identifying the vulnerabilities in lagging, development-only, and unreachable dependencies, which significantly reduces false alerts.

3.1.3 SCA Application in OSS Ecosystem

Apart from SCA techniques, researchers [29, 114–128] have applied SCA tools in the OSS and associated vulnerabilities in the OSS ecosystem, conclusions of which can be used to guide the designs of SCA tools. Decan et al. [114] studied NPM and Rubygems package managers and found that 33% and 40% of vulnerabilities respectively had their fixes within the same major release. Plate et al. [122] proposed new metrics to determine the criticality of vulnerabilities regardless of the types and languages of vulnerabilities, which helps with the automated impact assessment of new vulnerabilities. Imtiaz et al. [115] studied the characteristics of security fixes at 6 major package managers, namely, the semantic versions, release notes, and the time lag between fixes and releases, and offered 4 recommendations for the better practice of security releases. Ponta et al. [116] manually collected 625 publicly disclosed vulnerabilities for Java projects, which was also used in the Section 6.3 as the *Steady* data set at the latest version.

3.2 Software Compatibility and Semantic Versioning

Semantic versioning [17] has been proposed to enhance compatibility of software upgrades. However, it has not been strictly adhered to. Numerous research work has studied and proposed solutions to address the issue.

3.2.1 Study of Semantic Versioning Compliance

Many research works [27, 129–133] have studied the compliance of SemVer since its release. Raemaekers et al. [130, 131] found around 1/3 of all releases introduce at least one breaking change in seven years release history of Maven Central Repository. Decan et al. [132] studied 4 ecosystems (Cargo, NPM, Packagist, and Rubygems) to understand to what extent developers rely on SemVer to determine dependency constraints and found situations varying greatly among them. Ochoa et al. [27] revealed that 83.4% of upgrades of Maven comply with SemVer rules, and most breaking changes do not affect clients with only 7.9% of clients affected. The works drew conclusions based on signature-based incompatibility instead of SemB, which is not complete. Thus, *Sembid* is required to provide a more comprehensive analysis of the disobeying of SemVer by including SemB into the picture.

3.2.2 API Compatibility Checking

Only a limited number of research works [38, 39] regarding SemB of Java program were published in recent years. Consequently, their results are all subject to the commonly known drawback of unit tests, the coverage. Therefore, SemB detection based on unit tests cannot provide a comprehensive and scalable evaluation on the entire dependency library. Mostafa et al. [39] conducted an empirical study on behavioral incompatibility phenomena in popular Java libraries and analyzed published issues from Jira. DeBBI [38] used cross-project testing to amplify the testing coverage to detect Behavioral Incompatibility. Their implementations relied on unit tests, thus subject to the coverage. But *Sembid* relies on static analysis so that *Sembid* can conduct a more comprehensive analysis. Towards analyzing or detecting the signature-based API compatibility issues of Maven or Android

programs, massive empirical studies [134–141] have been conducted. RAPID [134] detects the status of incompatible APIs in the Android ecosystem. Huang et al. [135] studied callback compatibility issues of Android and developed a tool based on CFG to detect such issues. Jezek et al. [137] evaluated 9 commonly used syntactical incompatible API detection tools. Apidiff [142] determined the incompatibility at the name level of methods used in the target library. CiD [143] tried to alert the users by modeling the life cycle of APIs used in specific versions, while ACRYL [144] was a complementary method for CiD based on an alternative data-driven approach. The studies and detection tools based on the signature of APIs did not entail the semantics, thus, they cannot be used to detect SemB APIs like *Sembid*.

3.2.3 Behavior Similarity of Java Programs

Instead of the compatibility of the upgraded library, research works [145–148] have studied the similar behavior of two arbitrary programs. Li et al. [145] introduced the formal definitions of various Behavioral Compatibility and proposed metrics to measure the similarity of behaviors of programs in the scenarios of online education. DyCLINK [146] compared the similarity based on dynamic instruction graphs created by observing the program execution. Finally, although the semantic extraction approach to detect similar-behavior programs shares something in common with *Sembid*, similar behavior does not infer any Behavioral Compatibility. Behavioral Compatibility needs different insights to be modeled.

3.3 Vulnerability Propagation in Maven

Researchers have delved into studying the vulnerability propagation in the Maven ecosystem.

3.3.1 Persistence of Vulnerabilities

Researchers [47–49, 52, 54] have evaluated the propagation of vulnerabilities within the Maven ecosystem and recognized the long-term persistence of some vulnerabilities. Developers tend to address reachable vulnerabilities more than unknown

ones due to the potential for exploitation, as revealed by Wu et al.[47]. Pashchenko et al.[49] found that upgrades to vulnerable dependencies are often delayed due to potential breaking changes. Li et al.[48] conducted a similar quantitative study using a dependency graph integrated with vulnerabilities. Benelallam et al.[52] proposed the Maven dependency graph that has been widely used for ecosystem vulnerability analysis. Plumber [54] proposed by Wang et al. is a viable approach to address persistent vulnerabilities in NPM but not applicable to Maven because it relies on the pre-defined version ranges prevalent in NPM. Although insights have been highlighted, they have not proposed any tailored solution for persistent vulnerabilities for Maven.

3.3.2 Remediation for Maven Vulnerabilities

Regarding Maven vulnerability remediation, many solutions have been proposed [2, 6, 50, 53, 57–59, 149–154]. Du et al. [53] constructed a patch tracing system to locate patches to remediate the vulnerabilities. Industrial organization, OpenSSF [57], has proposed the best practice guidance [58] and a tool, Scorecard [59], for developers on managing vulnerabilities in dependencies for developers. Software Composition Analysis (SCA) [6, 77, 155–157] tools have also been widely adopted to assist in the mitigation of vulnerabilities persistent in users' projects. However, these studies focused on user-oriented remediation by developers instead of ecosystem-wide vulnerability mitigation.

3.3.3 Dependency Versioning in Modern Ecosystem

Many researchers have recognized the significance of the dependency versioning scheme for the security and stability of open-software ecosystem. [27, 41, 130, 132, 158, 159] Dietrich et al. [158] studied 17 package managers to investigate the dependency versioning recommended by them and found Maven heavily uses SoftVer leading to low flexibility of dependency versions. Google [41] released a blog about persistent vulnerabilities like Log4Shell and pinpointed that the SoftVers could be a cause of the persistent vulnerabilities. Decan et al. [132] reviewed the SemVer compliance in 4 popular package managers and summarized guidance for developers to better comply with SemVer. These works highlight the limitations

of dependency versioning in modern ecosystems, including the lack of flexibility in dependency management within Maven. As a solution, we proposed *Ranger* to restore the flexibility of version ranges within the Maven ecosystem.

3.4 Empirical Study for OSS Ecosystem

Research works have been conducted to study the security, sustainability, and maintenance in the OSS ecosystem.

3.4.1 Vulnerability Analysis in OSS Ecosystem

Numerous research studies [78, 160–165] have delved into the security of open-source software ecosystems. Decan et al. [114] conducted an empirical study to unveil the severity of vulnerability propagation in the NPM ecosystem. Alfadel et al. [166] analyzed security vulnerabilities in the PyPI ecosystem and found over 50% were patched after public disclosure. Zhang et al. [3] analyzed the persistent vulnerabilities in the Maven ecosystem and proposed *Ranger* to restore secure version ranges against the vulnerabilities. Wu et al. [47] studied the reachability of Maven vulnerabilities and found 73% of vulnerabilities are not reachable and safe for downstream users. Reid et al. [167] leveraged a file-level code reuse detector to locate the extensive vulnerable reused code in software projects in multiple languages, which severely jeopardizes the security of OSS. Fitzgerald et al. [168] proposed a knowledge flow graph that connects libraries, files, projects, authors, and code instances together to measure the multi-facet Free/libre open source ecosystem. Tan et al.[169] found that over 80% of affected CVE-Branch pairs remained unpatched in OSS projects. Xu et al.[170] studied CLV issues in PyPI and Maven ecosystems, identifying 82, 951 projects dependent on vulnerable C project versions. Shahzad et al.[171] measured a large vulnerability dataset, showing the prevalence of DoS and EXE vulnerabilities and an increase in SQL, XSS, and PHP vulnerabilities.

Existing studies mainly examine ecosystem security by either reasoning on dependency networks constructed from centralized package managers or identifying the

distribution of vulnerable code across artifacts. Instead, Golang, as the first ecosystem managed by decentralized registries while integrated with centralized indexing, its unique vulnerability life cycles and management strategies are firstly studied in this thesis. Wang et al. [172] developed HERO to investigate dependency management in Golang, achieving a high 98.5% detection rate on a Dependency Maze issue benchmark. Cogo et al. [173] studied same-day releases in npm, finding important changes despite the restricted time frame, with some releases being error-prone due to large quick changes. Kula et al. [174] explored developers' library dependency updates, discovering that 69% of interviewees were unaware of vulnerable dependencies. Zimmermann et al. [175] studied the potential impact of individual packages on large parts of the ecosystem due to vulnerable or malicious code in third-party dependencies.

3.4.2 Technical Lag Studies

Researchers [176–182] attempted to compare the time of the CVE publish, security fix and publication of a release that includes the fix. Imtiaz et al. [176] studied technical lag, including time between fix and release, documentation in release notes, code change characteristics, and time between release and advisory publication. They found that the median security release becomes available within 4 days, 61.5% of security releases come with release notes documenting security fixes, and 13.2% indicated backward incompatibility through semantic versioning, with 6.4% mentioning breaking changes in the release notes. Chinthanet et al. [178] investigated lags between vulnerable and fixing releases. They found that fixing releases are rarely standalone, with up to 85.72% of bundled commits unrelated to the fix. Stale clients require additional migration effort, even with quick package-side fixing releases. Decan et al. [177] empirically studied technical lag in the npm dependency network, finding that package releases suffer from technical lag without benefiting from the latest dependency updates. Previous studies ignored Golang's unique characteristics, where versions must be utilized before appearing in the Golang Index, setting it apart from other languages in versioning systems.

Chapter 4

Semantic Breaking Detection

4.1 Overview

Semantic Breaking is an advanced type of breaking changes beyond Syntactic Breaking, and there is an absence of effective tool to detect SemB other than unit tests. To bridge the gap, *Sembid* is introduced as a static approach to detect SemB issues in *Patch* and *Minor* upgrades. The tool is designed to detect SemB issues by measuring the semantic diff between the old and new versions of APIs. The tool is based on the observation that SemB issues are caused by the changes in the execution logic and the calculation of the output. The tool constructs the inter-procedural semantic graphs of the APIs to model the relationship between the input and output. The tool then measures the semantic diff between the old and new versions of the APIs by calculating the topological similarity of the semantic graphs based on the Weisfeiler-Lehman (WL) graph kernel. The tool further checks the impact of the SemB issues by analyzing the triggerability of the SemB issues and the propagation of the breaking changes to the API's output. The tool is evaluated on a dataset of 180 real-world SemB issues and 500 successful upgrades. The results show that the tool can effectively detect SemB issues with high precision and recall. The tool is also compared with existing tools and shows better performance in detecting SemB issues.

4.2 Background and Motivation

4.2.1 Semantic Versioning Rules

The Semantic Versioning [17] has been introduced in Section 2.1

The breaking changes are not allowed in *Patch* and *Minor* upgrades. Hence, to enhance the compliance with SemVer rules, *Sembid* aims at detecting breaking changes based on API over *Patch* and *Minor* upgrades to alert developers and users. Unlike SynB, SemB can hardly be detected by existing tools so *Sembid* focuses on detecting SemB to bridge the gap.

4.2.2 Motivating Example

The example in Listing 4.1 is used to demonstrate our motivation. The code was collected from a Jira issue [183] of *http-core* [184]. The upgrade caused the decoder to be stuck in an infinite loop when it reaches the end of the buffer if the input buffer contains a character '\r' that is never consumed. First, the change was not documented as a breaking, which suggests the breaking was unexpected. Second, the unit tests failed to uncover the case. Last, the *Minor* upgrade from version 4.2 to 4.3 intuitively indicated backward compatibility. Therefore, the judgment of the author is not always reliable so a static tool with better coverage is needed to detect potential breaking issues for *Patch* and *Minor* upgrades.

```
1
2  -if (this.contentDecoder != null &&
3  -    (this.session.getEventMask()&SelectionKey.OP_READ)>0) {
4  +if (this.contentDecoder != null) {
5  +    while ((this.session.getEventMask()&SelectionKey.OP_READ)>0) {
6          handler.inputReady(this, this.contentDecoder);
7          if (this.contentDecoder.isCompleted()){
8               resetInput();
9  +            break;
10 +        if (!this.inbuf.hasData())
11 +            break;
12 }}}
```

Listing 4.1: A Motivating Example from httpcore:4.2-4.3

4.3 Empirical Study

In this section, to understand the root causes of SemB and identify the patterns behind it, I first study the causes of SemB by analyzing the real-world cases. Then, as the SemB could be introduced intentially by the developers for benign causes, the code patterns of benign changes are further summarized for detection.

4.3.1 Study of Root Causes of Semantic Breaking

In order to avoid the over-fitting pattenrs, the real-world cases of SemB and the cases that involve no SemB must be analyzed for comparison. Next, the positive and negative cases are compared to identify the root causes of SemB.

4.3.1.1 Causes of SemB

Mostafa et al. [39] studied the Behavior Backward Incompatibilities in Java software, which also refers to the breaking APIs beyond signatures like SemB. They categorized the immediate causes into three: usage change (32.77%), e.g. enable/disable poor input; better output (55.74%), e.g. output format change; and other reasons(11.49%), e.g. internal structure changes.

Despite the immediate causes, Mostafa et al. [39] only focused on the user side of APIs but failed to dive into the internal code to locate the root causes. Since no other works have studied the root causes of SemB, we conducted one by analyzing the internal code changes of APIs that caused SemB with the aid of existing static java analysis tools, BCEL [185] and Soot [186]. The study included 180 real-world SemB issues (126 from [39], 54 collected by ourselves in recent 3 years) with commits. We found SemB had various forms which could hardly be summarised as patterns at the source code level, but they usually occurred along with the following changes:

- (73.33%) The changed execution logic. In Listing 4.1, the conditions remained the same (L2 & L4), but the *if* statement was altered to a *while* loop, which led to an infinite loop.

- (91.67%) The changed calculation of the output. For example, in a self-increment function, the change of calculation from $a+=1$ to $a+=2$ obviously modifies the output, while the execution logic remains the same.

The two types of causes overlapped in over 60% of issues. The first change in the execution logic embodied the inconsistency in Control Flow Graph (CFG). The second change in the calculation process, embodied in the changed data flow, can reflect the inconsistency of output values. For 86% of cases, the SemB changes dwelt in internal methods called by APIs instead of entry methods. Therefore, internal code analysis is required to detect SemB.

4.3.1.2 What Changes Will NOT Cause SemB?

Since successful upgrades have no specific indicator (no-issue is not enough), we could only study successful regression tests to understand why some changes do not cause SemB. We firstly collected 20 most used Maven libraries according to Maven Repository [18] with 77 version pairs and ran regression tests by testing the new implementation with old unit tests. $20,373$ tested APIs were derived. Then, we filtered out APIs with no changes in called methods to obtain $2,191$ APIs. 500 successful cases with binary change were randomly selected. For each case, we manually analyzed the diff dwelling in the methods called by the APIs to check if there existed an input to trigger the failure of unit tests. If the input did not exist, the reasons were collected. The cases of each reason may overlap with one another. The APIs passed regression tests due to:

- (27.4%) **Inadequate input to trigger SemB.** We manually examined these cases and found there existed input that could trigger SemB. *Sembid* is designed to overcome such limitations of tests.
- (19.6%) **Refactoring.** By [187], refactoring is the process of restructuring the existing factoring without changing its external behaviors, which is prevalent in Java by [188]. In general, the relevant API-level refactoring can be categorized as inter-method and intra-method. Inter-method refactoring, e.g. Extract Method, and Inline Method, is the most common type because it is frequently used to extend API flexibility based on Java polymorphism [189].

- (2.4%) **SemB not triggerable by old input.** The inconsistent behaviors cannot be triggered by the old variables, but only by the variables in the new version. This situation mostly occurs for new functionality, because the new input serves as the option to trigger additional behaviors.
- (14.6%) **Internal SemB has no impact on API output** The changed output of breaking changes has no impact on the output of the API to be observed by users. For example, if the logs are changed, while the return value remains the same, the output of the API is considered unchanged.
- (36.0%) **Benign changes** As allowed by SemVer, benign changes, such as bug fixes, and new functionality, do not substantially break the original semantics. They usually would not break downstream projects, and thus are considered false positives.

I found that except for the first reason (limitation of unit tests), the rest is the false alarm of SemB to be ruled out. This formed a dataset of API pairs that do not cause SemB, including 363 APIs, which would be used in the evaluation in Section 5.5.2.2. For the refactoring, *Sembid* extracts inter-procedural semantic graphs to eliminate syntactic refactoring. For the third and fourth reasons, *Sembid* excludes them by checking the impact of the captured SemB changes. As for the benign changes, we further conducted a study to identify them by patterns.

4.3.2 Study of Benign Changes

Sembid aims at identifying the benign changes to avoid reporting them as breaking. Since the benign changes stipulated by SemVer are not well defined, we summarise the syntactic patterns of benign changes by manually studying commits with such intentions. Then, these patterns will be categorized by their semantic representations on PDG and CFG to be automatically identified. According to [190], the intentions of commits can be categorized into (1) Bug fixing; (2) Performance improvement; (3) Feature introduction, deletion, and modification. Since the performance improvement does not explicitly change the output, it would usually not be caught by *Sembid*, thus unnecessary to identify it. As feature deletion and modification are not allowed by SemVer, they should be directly reported instead

of being ruled out. Hence, we focused on bug fixes and new functionality. We collected 584 commits from the most starred 25 Java projects on Github to summarise the patterns by searching for the keywords in commits, which are *fix*, *correct*, *improve*, *address*, *tweak*, *clean up*, and *add/new feature/functionality*. Then, the diffs were manually categorized into several patterns.

4.3.2.1 Bug Fixes (393 cases) Categories

- **Additional/ conditions and branches.** (41.85%) The change introduces additional conditions to narrow down or broaden the input range. The conditions mostly introduce new branches of statements for additional handling. This is usually to enforce the original rules by disallowing illegitimate input.
- **Changed/deleted conditions and branches.** (13.28%) The original conditions are changed or deleted to fix unreasonable behaviors. Usually, the change handles the illegitimate input to enforce the original rules, but it sometime introduces regression errors.
- **Similar substitution.** (11.60%) Variable types or methods are substituted with similar ones with semantically equivalent method names for better implementation. The original functionality is meant to be maintained.
- **Additional *try/catches*.** (10.08%) It introduces additional *try/catch* pairs or adds *catches* to existing *try*. This kind of change handles more exceptions to avoid unexpected crashes.
- **Assignment revision.** (10.76%) The output assignments are changed partially or completely to correct wrong behaviors or improve sub-optimal behaviors. Such changes can also be found in breaking issues because they could break downstream projects unexpectedly if they are used incorrectly.
- **Auxiliary variables.** (6.68%) Auxiliary variables are introduced to control the process, which often participates in *if* conditions. For instance, a counter is used to avoid infinite loops.
- **Other.** (5.75%) It consists of *changing internal type, initializing variables, improving method/variable modifiers*, etc. They slightly change syntax without modifying the original semantics.

4.3.2.2 New Functionality (191 cases) Categories

- **New classes/methods.** (61.29%) New classes/methods are introduced as the entries for new implementations which can be called by the existing APIs.
- **Additional branches.** (22.58%) New implementations/handlers are optionally accessible via new *if* or *switch* branches.
- **Additional parameters.** (9.68%) New parameters are added to existing internal methods to control or optimize existing functionalities by providing more options.
- **Additional fields.** (6.45%) New fields of returned objects are added with corresponding data and control dependencies. They are used to provide additional information loaded in the fields.

The new functionality cases were only considered *new* if the original functionalities were intact. Thus, the New Functionality cases were mostly additive changes. These patterns will be summarized to heuristics based on PDG and CFG. The commit IDs are provided on our website [191].

4.4 Methodology

According to Gunter et al. [192], since the change of semantics can be interpreted to infer the change of output which usually leads to the breaking issues, *Sembid* is designed to infer the SemB by measuring semantic diff.

Sembid aims at detecting SemB issues over *Patch* and *Minor* upgrades based on APIs. As shown in Figure 4.1, *Sembid* consists of five major steps, (1) **Group clusters from call graphs**: Given the API pairs, *Sembid* constructs the call graphs of them, from which it groups the consecutive changed methods as clusters. (2) **Derive Dependencies Summaries**: *Sembid* backward slices the output of clusters to obtain inter-procedural dependency summaries of data/control/exception. (3) **Match patterns for benign changes**: we heuristically summarized the patterns from Section 4.3.2 for *Sembid* to identify the statements of benign changes. (4) **Measure semantic diff**: Inter-procedural semantic graph pairs are constructed

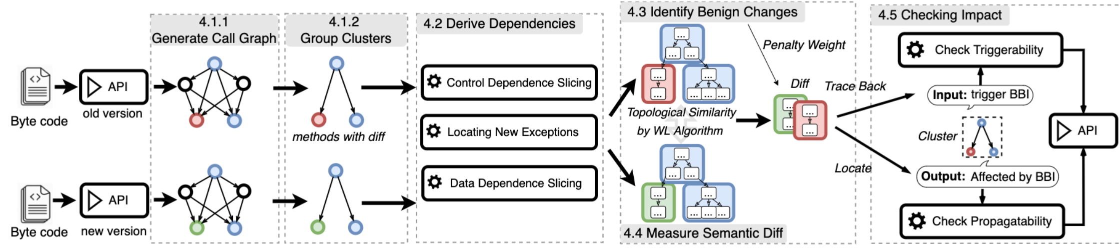


FIGURE 4.1: Overview of *Sembid*

from dependencies summaries. Semantic diff is measured by topological similarity based on subgraph isomorphism by Weisfeiler-Lehman (WL) algorithm. If changes in semantics are too large, the cluster is considered to be affected by SemB. (5) **Check the impact of SemB**: *Sembid* checks if the SemB of each cluster is triggerable as well as if the breaking change can propagate back to API's output.

4.4.1 Grouping Clusters from Call Graphs

Given the byte code, *Sembid* first identifies the APIs that are worth checking by narrowing down the API candidates to those with identical signatures only. Then, it constructs the call graphs for the APIs to derive the subsequent method calls for further analysis.

4.4.1.1 Generating Call Graphs for Candidate APIs

Sembid starts with identifying the set of API pairs that have no SynB. Specifically, $A_{cand} = \{\langle api_{old}, api_{new}\rangle \mid Sig_{api_{old}} = Sig_{api_{new}}\}$ is the API pair candidates. Note that the return type of Sig_{api} has to remain consistent, except it is a widening cast. For example, if the return type of the old API is changed from *long* to *int* in the new API, the compilation would succeed after the upgrade. To obtain A_{cand}, Soot [186] was leveraged to collect API sets A_{old} and A_{new} from class files of the old and the new libraries respectively.

Based on A_{cand}, *Sembid* generates call graphs of each API with Soot by the Spark algorithm introduced in Lhotak et al. [193]. For method bodies, Soot transforms them into Jimple [194], a typed IR. The size of the call graph can grow exponentially based on the depth, which results in the inefficiency of the analysis of methods at deep levels. According to [195], the semantics of a method decays along the calling

chain to be negligible at the depth of around 10 stacks. Thus, conservatively, we set the depth of the call graph $x \in [1, 15]$ to boost the performance.

4.4.1.2 Grouping Clusters

As discussed in Section 4.3.1.2, clusters are constructed by grouping methods that have signature or body changes to mitigate the inter-method refactoring. Given a pair of call graphs CG_{old} and CG_{new}, in terms of the method's signature and body code, methods from both call graphs can be classified as changed methods $M_{changed}$ and unchanged methods $M_{unchanged}$. According to [188], inter-method refactoring involves methods that are directly connected in a call graph. For instance, "Extract method" creates a new method to replace the removed block. Thus, *Sembid* groups as many directly connected (consecutive) methods as possible of $M_{changed}$ as clusters. The clusters are analyzed and sliced altogether as a whole to neutralize the inter-method refactoring.

4.4.2 Deriving Dependencies Summaries

For a cluster pair $\langle c, c' \rangle$, *Sembid* relies on three kinds of inter-procedural semantic summaries to model the relationship between the input and output, which are Data Dependency Summaries (DDS), Control Dependency Summaries (CDS), and Exception Summaries (ES). Each dependency in the summary is embodied as a Static Single Assignment (SSA) statement from the IR. These summaries are used to capture the factors that potentially change the output regarding the data calculation, control logic, and exception handling. Listing 4.1 will be used as a running example.

The DDS and CDS of the output of clusters are extracted based on backward slicing of the output in the Program Dependence Graph (PDG) recursively. We first define cluster $c = \{m_{root}, m_j \mid j = 0, ..., n\}$ where m is the method, and n is the number of methods, excluding the root. The ES is a set summarised from the unhandled exception exits of all methods within a cluster. Next, we define the output of a cluster as the non-local variables that are written after the execution of the cluster. The output can be in three forms: (1) Variables returned by m_{root}; (2) Class fields written in case m_{root} returns void; (3) Exceptions thrown,

as exception variable is a special non-local variable to be handled out of c. In general, the output is considered as the impact that the c imposes on the global environment. The semantic dependency summaries are robust against syntactic intra-method refactoring, such as moving, renaming, pushing down/pulling up, and splitting/merging local variables, because *Sembid* directly models the relationship among non-local variables without the interference of local variables.

4.4.2.1 Data Dependencies Summary

DDS is used to model the relationship between the input and output as an aspect of data calculation. Based on PDG, output statements are backward sliced to derive data dependence statements. If any non-local variable, such as parameter, is met, *Sembid* associates all statements met before with the non-local variable to check the triggerability later. Since the operands in SSA are prone to renaming, the operands are normalized as *var*, *parameter*, and *field* based on their roles.

To support inter-procedural analysis, when *Sembid* meets a called method m_0 within the cluster during the slicing, it dives into m_0. In m_0, the operations saving is executed with the same pattern as the root method m_{root}. The only exception is that parameters of m_0 are mapped to the local variables in m_{root} instead of non-local variables. If *Sembid* meets a method not included in the cluster, which are either unchanged methods or methods from other libraries, such as Java built-in classes, *Sembid* does not dive into it, as analysis of them does not make difference on SemB detection.

4.4.2.2 Control Dependency Summary

CDS describes the controlling semantics of the cluster, which determines the execution branching. Towards a pair $\langle c, c' \rangle$, CDS is derived by backward slicing the output of m_{root} with Control Dependency edges recursively. Boolean conditions, e.g. *if*, directly and indirectly, lead to the output will be collected. The inter-procedural analysis is conducted in the same manner as DDS. To normalize the conditions against syntactic changes, e.g. refactoring, *Sembid* calculates the Disjunctive Boolean Summary (DBS) which is a joint logic symbol of the original logic and the reversed one. For instance, $a > 0$'s DBS is $\langle > 0 \mid <= 0 \rangle$. Moreover,

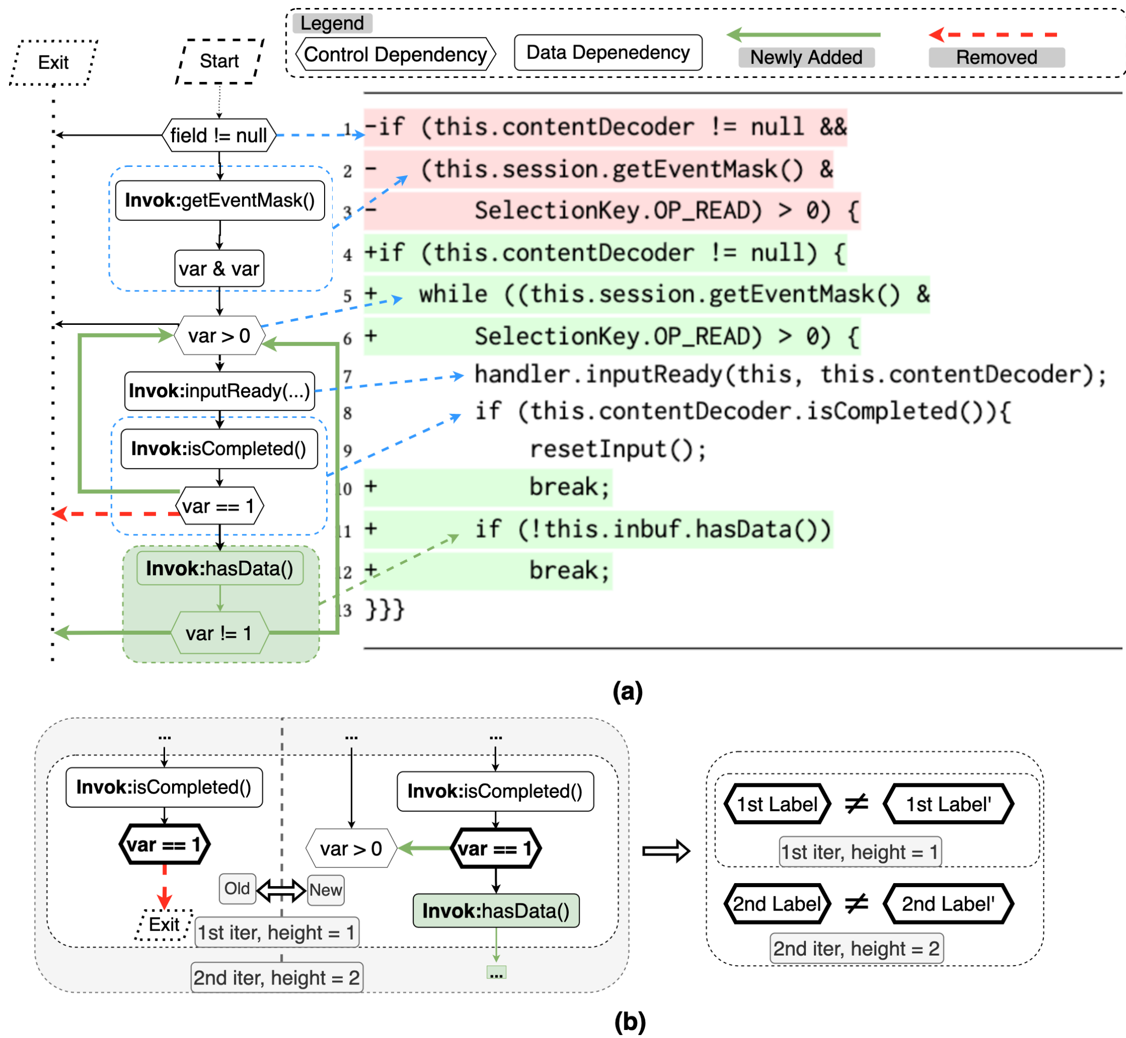


FIGURE 4.2: (a) Data & Control Dependencies Summaries in the semantic graph of Motivating Example (b) Example of subgraph comparison of Node *var==1* at height 1 and 2

the DDS of variables used in the conditions is extracted to capture the possible changes in variable values. These Data dependency statements are associated with the conditions.

Recall the motivating example, the DDS and CDS are extracted based on the output (L7 *handler*) slicing in Figure 4.2. These summaries on the left are organized based on the control flows from CFG. Since only L11-L12 introduces new conditions, the CDS changes, but DDS does not change. The switch from *if* (L2) to *while* (L5) changes control flow without changing CDS or DDS. Hence, It is necessary to include CFG in the analysis to capture all possible semantic changes.

4.4.2.3 Exception Summary

According to the study from [39], unexpected exception throwing is a common SemB issue, which is caused by un-handled new exceptions. Thus, it is necessary to check if the exceptions are consistent during the upgrade for the cluster. Because exceptions have no data dependencies, but only control dependencies, the exceptions extracted from the cluster are associated with the corresponding conditions from the CDS.

4.4.3 Matching Patterns for Benign Changes

In order to match benign changes in PDG and CFG. We first summarize semantic patterns from the syntactic source code patterns from Section 4.3.2. Given a cluster c, statements of benign changes are identified and collected to a set S^c_{ben}. According to Campos et al. [196] and Pan et al. [197], there were 9 categories of bug fixes patterns in Java. We further generalize them into 5, namely, from major to minor, controlling conditions (CC), field, method call, variable assignment, and try/catch. Also, we found the patterns of new functionality rely on the first 3 objects. Since these categories that can be identified from CFG/PDG based on IR could cover most cases of benign changes, heuristics to identify the overall benign changes were designed based on these categories.

- **CC Adjustment:** The benign changes adjust the conditions for legitimate use. If the subsequent implementations along the branch are not changed, the change is considered benign. **Steps**: (1) Get changed conditions from two lists of old and new conditions of CDS from Section 4.4.2.2. (2) Locate subsequent blocks of changed conditions by CFG and compare the DDS statements in those blocks to check if changed conditions lead to the same implementation. (4) If so, add the changed conditions as well as the backward sliced data dependency statements of them to S_{ben}, because they all contribute to the adjustment.
- **CC and *Try/Catch* Extra Handling:** Because the old implementation is not sufficient to cover all legitimate input, developers may add extra handling to the original to expand the input domain. The extra handling can either fix a corner case or introduce new functionality to support a new option. But in either way,

they are reflected as extra branches in exceptional CFG. **Steps**: (1) Locate only the new conditions in CDS and new *Catch* in ES. (2) Derive statements in two subsequent blocks of boolean conditions and *Catch* by CFG. (3) Since the extra handling would not tamper with the original implementation, one of the blocks should remain identical and the other one is new. Compare the blocks with the old implementation to get the new ones. (4) Add these new conditions/exceptions, statements from new blocks, and their data dependency statements to S_{ben}.

- **Field: Augmented Output:** In Java, the instance as output is augmented to accommodate more fields. Another case is non-static methods returning void set more fields as the output. **steps**: (1) Identify new fields by comparing the output's old and new field lists by name and type. Note that *Sembid* only captures the newly added fields, which means the old fields should remain the same. If the number of output fields does not increase in the new version, it is not considered augmented, but possible breaking changes, as the old implementation could rely on the types of original fields. (2) Backward walk through the def-use chains of the fields recursively to add relevant statements to S_{ben}. Backward slicing does not work in this case, because it returns dependencies of the output instance, instead of specific fields. Hence, fine-grained slicing at the field level is applied.

- **Method Call: Similar Substitution:** Classes or methods are substituted with similar analogies to enforce the original rules. In this case, the semantics of the substituted component is reflected by its role in the context. For example, if one statement of invocation is replaced, but the contextual statements remain the same, the role of the substitution is not changed. *Sembid* captures this semantics by using a neighbor-preserving algorithm in Section 4.4.4, which considers two nodes are identical if all 1_{st} neighboring nodes remain the same.

- **Variable Assignment Revision:** Some variables are assigned different values or new assignments are included. If the variable is local and not a data dependency of the output, it is likely to be an auxiliary variable that controls correct behaviors without directly tampering with the value of output. It is identified by (1) Obtain data dependency statements of conditions from CDS. (2) Compare them based on each condition to get new statements, and check if they belong to new variables. (3) If so, add them to S_{ben}. If the output or data dependency of output is assigned different values, we do not explicitly classify them to S_{ben}, because it is likely to introduce breaking issues by yielding abnormal output.

Theoretically, benign changes should be identified and ruled out to avoid false positives. However, the identification cannot be perfectly accurate and they sometimes still cause SemB, such as the regression issue, because either developers fail to anticipate the breaking or the downstream projects use the API illegitimately. Thus, *Sembid* fuzzily measures the semantic diff to determine the SemB with the de-emphasized benign changes instead of completely ignoring them. However, if the accumulated semantics is changed greatly, it is still considered SemB.

4.4.4 Measuring Semantic Diff

To measure the semantic diff for a cluster pair, *Sembid* constructs an inter-procedural semantic graph as Figure 4.2 by connecting Exception/Data/Control Dependencies Summaries with execution paths from CFG. The semantic graph preserves the execution logic among relevant dependency statements to model the behaviors of non-local variables of clusters. *Sembid* infers the extent of semantic change by calculating the topological similarity of semantic graphs by subgraph matching algorithm based on Weisfeiler-Lehman (WL) graph kernel [198] with weighted statements of S_{ben}. If the final value is above a threshold, the cluster is affected by SemB.

Here are the reasons for the adoption of the WL graph kernel. WL graph kernel can convert the original graph to a sequence of substructures defined as kernels that sort and compress topological and labeling information of adjacent nodes. The kernel pairs preserving the neighboring semantics can be used to calculate the semantic similarity based on the number of matched kernels. Besides, the runtime scales linearly in the number of edges better than other kernels, such as Random-Walk or Shortest-Path [198]. WL graph kernels are designed for directed discrete large graphs which suit the scenario of semantic graphs.

Inspired by the graph matching algorithm of PDG in CCGraph [199], *Sembid* relies on the subgraph isomorphism based on h iterations of WL graph kernel calculation to determine the semantic diff between semantic graphs $\langle G, G' \rangle$. CCGraph targets detecting code clone only based on PDG, while the semantics from PDG is not sufficient for SemB detection. For example, the crucial control flow change in Listing 4.1 is not reflected in PDG. Thus, *Sembid* measures the semantic diff

between sliced statements connected by control flows for better semantic representation. Also, *Sembid* further de-emphasizes the benign changes to align with SemVer rules. The procedures are described below in Algorithm 1:

- Labels of nodes in graphs are hashed as the initial values.
- In i_{th} iteration of WL algorithm [198], labels of each node as well as its neighbors in i hops are compressed into a new label by local sensitive hashing according to WL algorithm.
- In i_{th} iteration, if labels of two nodes are identical, the subgraphs of the node pair are considered as isomorphic at height i.
- After all iterations, the number of non-identical node pairs multiplies with a deteriorating weight w and benign penalty p per pair as the final graph kernel value K. K is normalized to compare against the threshold T. If $K > T$, the cluster pair has SemB. K is calculated as

$$K = \frac{\sum_{i=1}^{h} \langle l(n) | l(n) \neq l(n') \rangle * (h-i+1)/h * sizeof(S_{ben})/sizeof(G')}{min(sizeof(G), sizeof(G'))}$$

Same as [199], deteriorating weight w is calculated as $(h - i + 1)/h$ for i_{th} iteration, because the closer the neighbors of node n are, the more they affect the node n. The benign change penalty p is calculated as $sizeof(S_{ben})/sizeof(G')$ to dynamically adapt to the size of G'. Since lower T lowers the precision of *Sembid*, while higher T lowers the recall, the T is empirically set as 0.1 to balance the precision and recall. In Figure 4.2(b), the node *var==1*'s neighbors are converted to subgraphs and then compressed by WL algorithm to form a label at height 1. Height 2 is formed in the same way. Evidently, none of the label pairs of node *var==1*' is identical.

4.4.5 Checking Impact of Semantic Breaking

This procedure only proceeds when a SemB cluster is caught in the previous step. Only if both triggerability and propagatability are feasible, the SemB cluster is considered a threat to the API.

Algorithm 1: Algorithm of Measuring Semantic Diff

Input: $\langle c, c' \rangle$: clusters pair with nodes n, label $l(n)$. h: iteration number of WL algorithm. T: threshold, w_i: deteriorating weight at i_{th}, p: benign change penalty.

Output: R: Result of existence of SemB in $\langle c, c' \rangle$

```
1  foreach i_th iteration in h do
2      foreach node n in cluster c do
3          sg(n) ← WLGenSubGraph(n, neighbor(n, i_th))
4          L(n) ← Σ(l(n)|n ∈ sg(n))
5          L(n) ← sort(L(n))
6          l(n) ← l(n) + L(n)
7          l(n) ← WLcompress(l(n))
8      w_i ← (h − i + 1)/h
9      k_i ← sizeof(l(n) ≠ l(n')) * w_i * p
10 k ← Σ k_i/min(sizeOf(c, c'))
11 if k > T then
12     R ← 1
13 return R
```

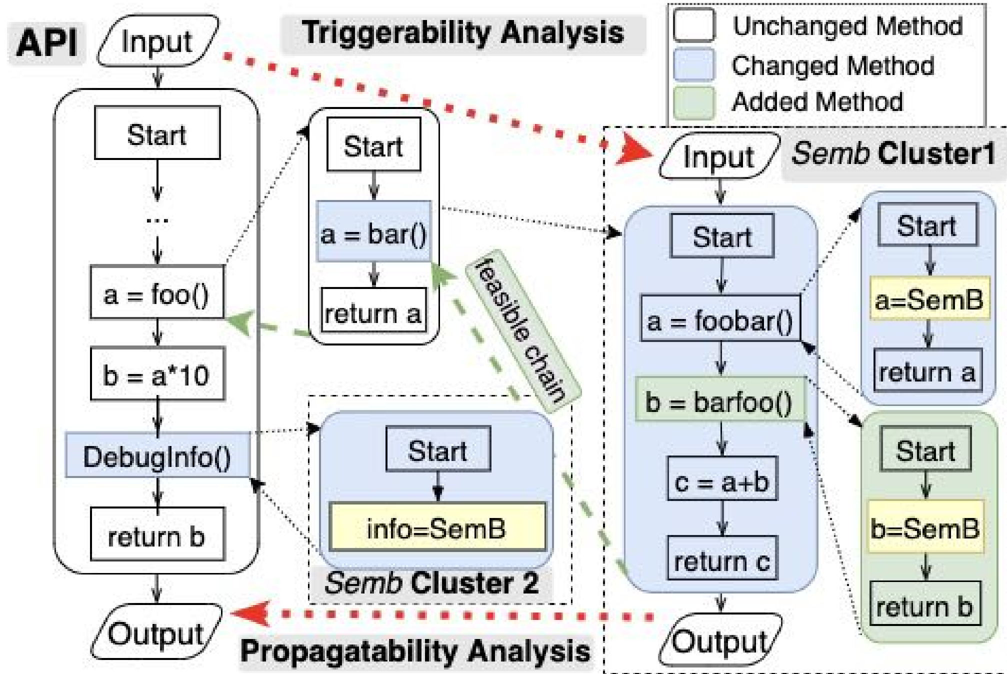


FIGURE 4.3: Triggerabitliy and Propagatability Analysis

4.4.5.1 Verifying Triggerability

Triggerability defines if SemB can be triggered by old input. In Section 4.4.2, during the backward slicing, input is associated with relevant statements. If the statements are caught as the SemB changes, the associated input would be checked, including parameters and fields. If the problematic input is introduced in the new

version, the SemB cannot be triggered, because the old implementation cannot access the new input.

4.4.5.2 Verifying Propagatability

Propagatability, as defined in this thesis, determines if the SemB output can be directly or indirectly affected by the cluster $c = \{m_{root}, m_j \mid j = 0, ..., n\}$ along call paths to API to further influence the downstream users. This is done by examining the inter-procedural data and control dependencies of the output of the API. Specifically, since SemB is introduced in the new version, *Sembid* only verifies the propagatability in the call graph of the new version. *Sembid* uses JGraphT [200] serving as the graph infrastructure which first derives all call paths by Dijkstra Algorithm [201] from the API to the m_{root} as $P = \{path_i \mid i = 0, ..., n\}$. Second, For each call path, *Sembid* calculates PDG for every method along the call path to verify if the inter-procedural dependency chain is feasible along every PDG from m_{root} to API's entry method. Finally, propagatability Pg is obtained by $iff \exists path \in P, depChain(path) == True, Pg = True$. Note that both Triggerability and Propagatability analysis are *May* analysis, yielding statically possible paths. Without the actual input, the actual triggerability and propagatability are not guaranteed.

For example, in Figure 4.3, *cluster 1*'s output is propagatable to the API's output. The output of m_{root} of *cluster 1*, $return\, c$ has a feasible dependency chain $a = bar() \rightarrow a = foo() \rightarrow return\, b$ as indicated by the green dotted line. Thus, the output of the API indirectly depends on the SemB change in *cluster 1*. In contrast, an example of an un-propagatable SemB change is *cluster 2* which yields debugging information not used as the API's output.

4.5 Evaluation

Sembid was implemented in 9.2K LOC based on Jimple IR of Soot 4.2.1 in Java. We aim at answering the following research questions:

RQ1: What is the accuracy of *Sembid* in terms of SemB detection?

RQ2: How is the effectiveness of *Sembid* against unit tests?

RQ3: How do top Java libraries comply with SemVer rules?

4.5.1 Evaluation Setup

For RQ1: To evaluate the accuracy of *Sembid*, we first construct a high-quality ground truth dataset for the benchmark with other API checking tools and the baseline.

Benchmark Dataset Collection. Since *Sembid* is designed to detect SemB across *Patch* and *Minor* upgrades, we collected broken API pairs of *Patch* and *Minor* upgrades from the 20 most used Maven libraries. The steps are (1) We located Github repositories of those libraries. (2) We conducted regression tests by running unit tests from the old versions against the implementations in the new versions to detect SemB. (3) After ruling out the SynB, the failures that caused *AssertionError*, unexpected exceptions, and crashes are considered as SemB, because *AssertionError* means the program is executed abnormally. Since Mostafa et al. [39] conducted the same regression tests on some of the libraries before 2018, we extended the dataset by extracting APIs from their testing logs in the same steps. Eventually, we derived 308 API pairs with SemB from 77 version pairs. For the dataset of compatible API pairs aligning with SemVer rules, we used the dataset from Section 4.3.1.2. The previous study has evluated 500 API pairs to understand why binary-level changes did not induce SemB via unit tests. After excluding the cases caused by the insufficient coverage of unit tests, 363 API pairs were obtained. These API pairs with binary changes without SemB can serve as the negative data set of SemB detection.

Metrics. The outcomes of *Sembid* are categorized into (1) True Positive (TP): APIs reported have SemB. (2) False Positive (FP): APIs reported do not have SemB. (3) True Negative (TN): The API not reported by *Sembid* has no SemB. (4) False Negative (FN): The API not reported by *Sembid* actually has SemB. Precision, recall, and F-measure are used as evaluation metrics.**For RQ2:** We conducted another experiment to verify the effectiveness of detecting SemB APIs over version pairs between *Sembid* and the commonly used SemB detection solution, unit tests. As the number of APIs of popular libraries is considerable (406, 826 APIs of 77 pairs), the efforts of manual ground truth checking would be overwhelming. Hence, we selected the top 4 most used libraries with 1 SemB *Patch* version

pair each, as they are more likely to have the best testing coverage. In total, 3,846 APIs were collected.

For RQ3: All semantically successive version pairs published in the last 20 years of 21 most used Java TPLs from the Maven repository [18] were collected for the large-scale analysis. In total, 546 version pairs and 1,629,589 APIs were tested. To analyze the compliance of the SemVer rules, we classified them into (1) *Patch*: 334 pairs; (2) *Minor*: 163 pairs; (3) *Major*: 49 pairs. The versions collected were stable unless none was available.

4.5.2 RQ1: SemB Detection Accuracy

To evaluate the accuracy of *Sembid* against existing tools, we have selected 5 Java API compatibility checking tools (i.e., revapi [36], japicc [37], japi-checker [34], clirr [33], sigTest, [35]), which are commonly used in benchmarks [137] and industrial software, such as Apache Httpclient [202]. Besides existing tools, we also implemented a baseline tool that relies on the same call graph construction procedures as *Sembid*, but SemB is considered as positive if any binary diff exists in any method called by APIs.

Table 4.1 presents the accuracy evaluation of *Sembid* and other tools on the benchmark data set from Section 4.5.1. *Sembid* achieves 90.26% recall, 81.29% precision, and 85.54% F-measure. It is concluded that *Sembid* outperformed other tools because these tools could only detect SynB instead of the SemB. There were two reasons why some tools could still have TP. First, tools, such as revapi, not only evaluated the compatibility based on signatures but took other features, such as method accessibility and abstraction into account. Second, some of the tools only returned the incompatible class names without method names. If any API signature from the data set had the same class name as the returned class names, we considered the SemB API was detected.

False Negative Cases Discussion. We manually examined the 30 false-negative cases and summarised 4 reasons why *Sembid* made false decisions. A detailed name list is provided in [191].

- (46.67%, 14 cases) **Secondary Output**: It is the output embodied as debugging messages, written files, and other auxiliary information conveyed by the

Table 4.1: Benchmark Accuracy of SemB Detection based on APIs

Tools	TP	FN	Recall	FP	Precision	F-measure
Sembid	**278**	**30**	**90.26%**	**64**	**81.29%**	**85.54%**
baseline	302	6	98.05%	363	45.41%	62.07%
revapi	30	278	9.74%	21	58.82%	16.71%
japicc	21	287	6.82%	14	60.00%	12.25%
japi-cker	14	294	4.55%	10	58.33%	8.44%
clirr	1	307	0.32%	0	100.00%	0.64%
sigTest	0	308	0.00%	0	N.A.	N.A.

APIs. Unlike the primary output, the secondary outputs are not global variables passed to the users' programs, but the unit tests sometimes still include them by *assertion*. Since, unlike the secondary output, users directly use the primary one which could break downstream projects, *Sembid* focuses on the primary output to cover the mainstream scenarios.

- (23.33%, 7 cases) **Falsely Identified Benign Changes**: The patterns of these changes fall into the summarised benign changes. However, no evidence suggested they are benign from commit messages or documentation. Although de-emphasizing benign changes is not perfect, it does improve the precision at the relatively low cost of false negatives.

- (20.00%, 6 cases) **Signature Reflection**: The breaking was caused by different signature reflections [203]. *Sembid* is not able to capture the behaviors that can only be triggered dynamically, and thus can complement testing for a more comprehensive detection.

- (10.00%, 3 cases) **Subtle Changes**: The changes were too subtle to be detected by the semantic diff measuring, such as the change of index number of an array, which may not always cause changed output. Although the threshold may overlook some subtle breaking changes, it reduces many more false positives.

False Positive Cases Discussion. 64 false-positive cases were listed with 3 reasons why the results were incorrect.

- (53.13%, 34 cases) **Large Accumulated Semantic Diff**: Although *Sembid* already assigned low weights to the benign changes, multiple benign changes can still have a considerable stacked impact on measuring semantic diff. Because

benign changes cannot be accurately identified and filtered out, we have to trade off between the under-fitting and over-fitting by weighting.

- (37.50%, 24 cases) **Equivalent Re-implementation**: The code with the same functioning was re-implemented in another way in the new version. One example is the Java feature evolution. Java 8 [204] introduced a feature, *stream*, for parallel aggregate operations. Although it can be used with *forEach* to work the similar way as primitive *for loop*, they are written in totally different byte code. Another example is the change from *array.getElementAt(n)* to *array.subString(n, n+1).get(0)*. Both of them return the same element in the array, but they are implemented in different ways, which resembles the *Type 4 Code Clone* [205] problem that hardly has efficient and effective solutions so far (Type 4 Code Clone refers to the two code segments share similar functionalities and semantics without any syntactic similarity). Hence, *Sembid* fails to identify the semantic equivalence between them at the current abstraction level.

- (9.38%, 6 cases) **Unhandled Exception**: They were detected as SemB APIs because the newly thrown exceptions are not correctly handled in the new version. But they are actually handled by the super-type exception catchers. *Sembid* checks the exception handling by comparing the exception signatures of the thrown and the catcher. If they are not the same or the catcher is not a general exception, such as "Exception", the thrown exception is considered not caught. In fact, if the thrown is an inherited sub-type of the catcher with different signatures, the thrown can still be caught. *Sembid* made such wrong decisions due to the lack of knowledge of the exception inheritance hierarchy.

The **baseline** tool achieved high recall but low precision, which failed to detect 6 cases of signature reflection due to its static basis. All APIs from the compatible test set were false positives.

Conclusion of RQ1: *Sembid* outperformed other API compatibility checking tools and achieved 90.26% recall, 81.29% precision, and 85.54% F-measure in terms of SemB API detection. *Sembid* achieved much better precision than the baseline tool (45.41%), which indicates that *Sembid* is able to effectively filter out the false-positive changes.

4.5.3 RQ2: Effectiveness of Detecting SemB against Unit Tests

As unit tests are widely used to detect SemB based on APIs, we compared *Sembid* against unit tests regarding the number of detected SemB APIs. A similar tool, DeSemB, was proposed by Chen et al. [38] to detect SemB based on augmented unit tests, but it requires manual analysis, and no public data or source code is available. Hence, DeSemB is not involved in the evaluation. Based on the selected libraries from Section 4.5.1, we first obtained APIs that have binary change as set $A_{changed}$. Then, we derived all tested methods from the testing source code. Because developers would not explicitly mention what methods or classes are tested, we made an overestimation by assuming all public and instantiable classes used in the tests along with their public methods are tested. Based on this assumption, we processed the testing classes in the following manner: (1) Irrelevant testing methods were filtered out according to the rules of testing frameworks. For example, if a framework, Junit, was used, methods annotated with @$before$, @$after$, @$ignore$ would be ignored. 2) From the relevant testing methods, we constructed call graphs for each of them, then directly called methods that meet the aforementioned conditions were collected. (3) During running the tests, the dynamic call graphs were calculated to derive the dynamically called APIs. APIs collected are formed as a set A_{tested} to denote the changed APIs covered by tests. A_{tested} has $3,846$ APIs.

In Figure 4.4, the results are illustrated. In total, $3,846$ APIs were evaluated in 4 version pairs. The ground truth of them was manually confirmed. It is evident that from the first bar unit tests covered averagely 16.61% of APIs, while the second bar indicates that *Sembid* can cover 100% APIs and focus on detecting potential SemB in 15.00% (577) APIs with binary changes. The unchanged APIs are naturally compatible. Apart from the coverage of APIs, even for APIs covered by unit tests, unit tests only uncovered 0.48% (16) SemB APIs of all APIs. According to the ground truth denoted by the third bar, unit tests failed to detect 79.46% (62/78) of SemB APIs, but *Sembid* successfully covered 92.30% (72/78) of SemB APIs. Although *Sembid* made some false alarms (21.73% = 20/92), generally *Sembid* detected more SemB APIs than the unit tests with 4.5 times more TP. However, *Sembid* as a static tool has its limit. As unit tests can dynamically detect SemB in certain APIs with reflection, but *Sembid* is not able to cover them statically.

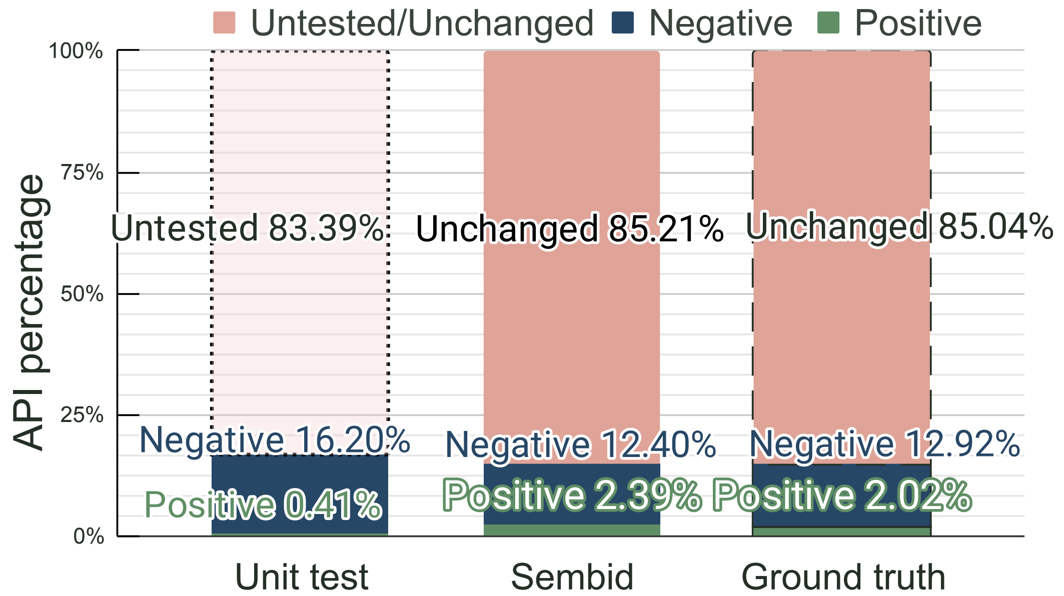


FIGURE 4.4: Comparison between Unit Tests and *Sembid* Over All APIs During Upgrades As Well As the Ground Truth

Conclusion of RQ2: Over 4 version pairs, 3,846 APIs in total, *Sembid* detected 4.5 times more SemB APIs than unit tests (72 v.s. 16) and achieved better coverage of APIs (100% v.s. 16.61%). Unit tests are able to detect 6 more SemB caused by dynamic operations than *Sembid*. It indicates that *Sembid* can serve as the complement of unit tests to detect more SemB for *Patch* and *Minor* upgrades.

4.5.4 RQ3: Study of Compliance with SemVer

To verify the compliance of SemVer rules in popular Java TPLs, we evaluated the TPLs at library, version pair, and API levels respectively. During the evaluation, both SynB and SemB are considered as evidence of breaking (either SemB or SynB is a breaking). For each version pair, the APIs affected by SynB and SemB as well as the APIs with binary changes were collected. The detection of SynB depends on the aforementioned API checking tools. The SynB APIs are the union of them. The SemB was detected by *Sembid*. The changed APIs were collected with Soot and BCEL.

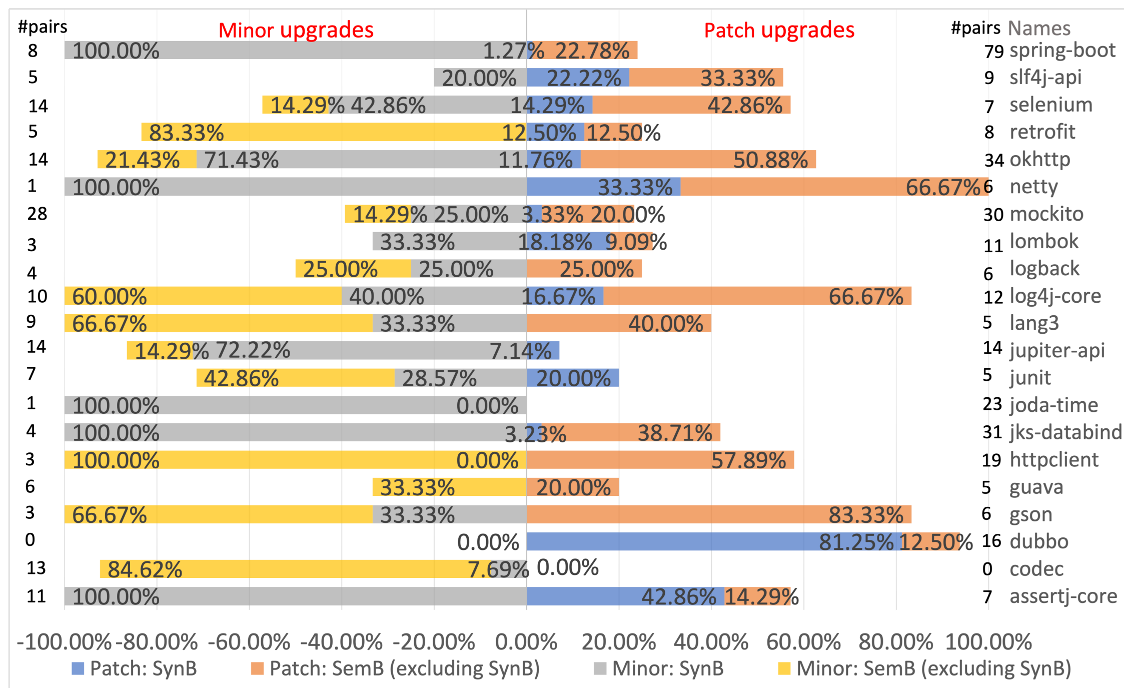


FIGURE 4.5: Proportions of Version Pairs Affected by SemB and SynB of *Patch* and *Minor* Upgrades

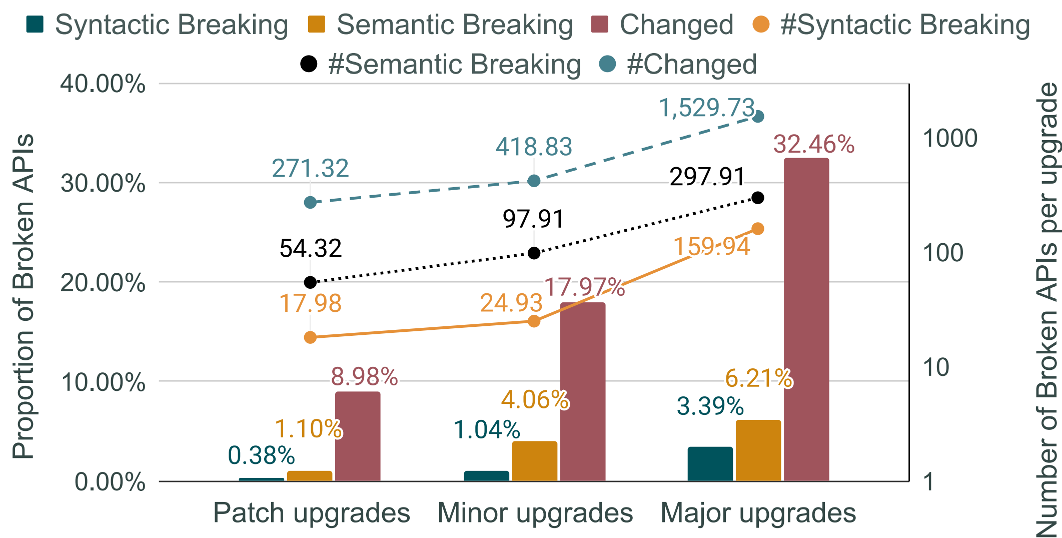


FIGURE 4.6: Average Number and Percentage of APIs Over Three Upgrades

Beginning with library and version pair levels, Figure 4.5 illustrated the proportion of SemB and SynB version pairs of 21 libraries. If one breaking API was detected in a version pair, this version pair was considered to be broken. Note that the SemB version pairs were counted when the version pairs were free from SynB. In other words, the sum of SemB proportion and SynB proportion is the proportion

TABLE 4.2: Execution Time per API (ms)

Call Graph Const.	**Detection**	**Trigger & Propagate**	**Total**
5.74	2.74	4.65	12.58

of all broken version pairs. The left side is the *Minor* upgrades, and the right side is the *Patch* upgrades. The numbers of version pairs are annotated at the ends of bars. We found that

- **Patch upgrades on average**: 14/21(66.67%) libraries were subject to SynB for at least one version pair (1.27% – 81.25%). 19/21(90.48%) libraries have at least one breaking version pair. It is seen that SemVer rules are hardly applied to popular libraries. However, the situation is getting better at the version pair level. On average, SynB affects 10.45% of version pairs, but SemB affects additional 23.35%, which makes 33.83% of version pairs affected by either breaking. The SemB version pairs are over 2 times of SynB. It suggests that SemB detection is necessary in *Patch* upgrades.

- **Minor upgrades on average**: All 20 libraries with minor upgrades are affected by any breaking. It is observed that developers are more likely to include breaking changes in *Minor* upgrades than *Patch*. At the version pair level, on average, SynB affects 37.42% of version pairs, and SemB affects additional 26.99%, which makes 64.42% of either breaking. Due to legacy reasons, libraries adopt various version release strategies, and many libraries' administrators allow breaking changes in *Minor* upgrades.

- **Particular libraries**: The performance varies tremendously among libraries, because they adopt different, even opposite version release strategies. Some libraries have a high ratio of breaking version pairs, such as *dubbo*, *netty*, *log4j*, *gson*. Because they have very few or 0 *Minor*/*Major* upgrades, they frequently make *Patch* upgrades. They usually make *Major* upgrades cautiously when the entire structure is rewritten, such that *log4j* upgrades from 1.x to 2.x, *httpclient* 4.x-5.x. Then almost all normal upgrades with breaking changes were accommodated in *Patch* and *Minor* upgrades. If *Minor* upgrades are rare, *Patch* upgrades would be mostly broken. For this kind of library, SemVer rules are not properly applied so users have to identify the strategies case by case. There are also some

libraries adopting the opposed strategy. For instance, *guava* made *Major* upgrades frequently (13 times in 7 years) with a few *Patch* upgrades. Although *guava* still has a few unexpected SemB, SemVer rules are basically well applied, thus users can upgrade it by SemVer rules with ease.

Considering version release strategies vary greatly among libraries, SemVer rules are not generally followed by popular libraries. Apart from SynB, SemB is also prevalent over *Patch* and *Minor* upgrades. For developers, *Patch* and *Minor* upgrades should check SemB too before publishing to avoid breaking downstream projects. For users, to avoid identifying version release strategies of dependencies case by case, *Sembid* can be used on dependencies to pre-check the breaking APIs before upgrading along with SynB checking tools.

The SemB and SynB proportions at the API level are illustrated in Figure 4.5, *Changed* denotes the number of APIs with binary changes. It is observed that, for SemB, *Sembid* significantly filtered out non-breaking APIs from *Changed* APIs. The *SynB* APIs were way fewer than *SemB*, which means *SynB* checking is far from enough. Moreover, there were still 1.10% APIs affected by the SemB in *patch* upgrades and 4.06% in *minor* upgrades. Since these libraries were dependencies of over 110k artifacts (libraries) in the Maven ecosystem, the unexposed SemB APIs could unexpectedly detriment the functionalities of those artifacts.

The performance of *Sembid* was measured, which is demonstrated in Table 4.2. On average, it took 12.58ms to perform a SemB detection on an API. This process consists of three stages, namely, call graph construction, detection (including harmless change filtering), and triggerability & propagatability analysis. The average time per API of them is respectively 5.74ms, 2.74ms, and 4.65ms. At the first stage, the call graph construction time comprises the call graph and sub-graphs construction. The most time-consuming part is the call graph algorithm, named Spark [206] provided by Soot, as it is more expensive but more accurate than other existing algorithms. At the second stage, the detection of SemB, consists of 3 parallel semantic summaries extraction and comparison. Since the call graphs have to be constructed for all APIs to filter out the unchanged APIs, but *Sembid* only detects SemB on the rest, the averaged detection time is smaller than the call graph construction time. As for the last stage, triggerability and propagatability analysis, although it only works when the SemB change is located in the sub-graph, it involves Dijkstra Algorithm to find paths and the iteration of all paths, which can

take fairly long if the call graph is large. Thus, the average time was longer than the second step. All in all, *Sembid* is pretty efficient considering one run for one upgrade typically takes less than 3 minutes. the time per API for most libraries are negeligible, because this stage only occurs for the sub-graphs affected by the SemB issues, which only account for 3.11% of all APIs. Thus, the fewer SemB APIs are, the less time the third stage takes.

Conclusion of RQ3: From the experiment with 1,629,589 APIs in 546 version pairs, 1.10% of APIs from *Patch* and 4.06% of APIs from *Minor* upgrades were affected by SemB. They are 2-4 times more than SynB APIs (0.38% and 1.04%). In terms of the 497 *Patch* and *Minor* version pairs, *Patch* upgrades have 33.83% breaking pairs, and *Minor* upgrades have 64.42% breaking pairs because version release strategies adopted by libraries vary greatly.

4.6 Threats to Validity

The primary threat is the approximation of using the semantic diff to infer the change of actual behaviors at runtime. Some of the false alarms are caused by this threat, because the *may* analysis of all possibilities is an overestimation of actual behaviors. Symbolic execution [207] or concrete unit testing [208] can be used to verify the compatibility by testing and checking input and output domains. However, we need to balance the trade-off between the accuracy gain and the performance loss.

Another threat is that benign behaviors filtering cannot ideally reflect the real intention of the developers, because the rules to filter out harmless behaviors were made based on the empirical summary. The commit intention classification technology can be used to facilitate the accuracy of benign change identification, but they would introduce heavy procedures, such as Machine Learning models, which handicap the scalable and efficient deployment.

The last threat is the scope of *API*. We took public methods of instantiable classes as APIs which are a superset of client-used APIs. However, since Java has no built-in indicator to mark exposed API, it is hard to accurately locate actually

used APIs. Taking advantage of usage data of TPL methods is plausible, but it is still not accurate.

4.7 Limitations

A major limitation of *Sembid* is its reliance on the statically constructed call graphs by Spark algorithm. Although it is the state-of-the-art static analysis algorithm, it is not perfect. The call graphs are not absolutely accurate, because the Java language is dynamic and reflective. Confined by static approaches, the dynamic behaviors, such as reflection, are not covered by *Sembid*, leading to false negatives.

4.8 Conclusion of Semantic Breaking Detection

We proposed *Sembid* to statically detect SemB based on APIs during *Patch* and *Minor* upgrades to enhance the compliance of SemVer rules. Experimental results demonstrated that *Sembid* achieved 90.26% recall and 81.29% precision. Another experiment proves that *Sembid* with larger coverage detected 4.5 times more APIs than the commonly used solution, unit tests. Furthermore, a study was conducted on the top 21 Java libraries for over 1.6 million APIs, and 546 version pairs to evaluate the compliance with SemVer rules at the library, version pair, and API levels, which revealed that 33.83% *Patch* upgrades and 64.42% *Minor* upgrades had at least one API affected by any breaking. And on average, there were 2-4 times more APIs affected by SemB issues than SynB issues.

Chapter 5

Compatible Remediation of Vulnerabilities

5.1 Overview

Considering that existing remediation tools do not provide a global solution that collectively resolve the vulnerabilities within an individual software projects, a comprehensive solution is required. To this end, *Coral* is presented, which is a holistic approach to remediate vulnerabilities in Maven projects. *Coral* is designed to address the challenges of the high complexity of global optimization and the ripple effects caused by the dynamic changes of the dependency graph. The problem of remediation in a dependency graph is formalized as an optimization problem [209]. The primary objective of *Coral* is to minimize the total vulnerability risks while ensuring the compatibility of the version adjustments. The compatibility is defined as the absence of incompatibility issues, including semantic and syntactic breaking and dependency conflicts. To achieve the global optimization and handle the ripple effects, *Coral* partitions the dependency graph into subgraphs and optimizes them regarding the vulnerability risks based on the pre-computed vulnerability mappings while ensuring compatibility. The optimization is conducted in a top-down manner, starting from the root user projects. The selected versions can be overthrown by the next optimization, and all selectable candidate versions are saved and fed to the next optimization. To avoid sub-optimal solutions and dead ends, *Coral* implements two types of backtracking mechanisms, hard and soft backtracking.

Hard backtracking is used to avoid dead ends, while soft backtracking is used to avoid sub-optimal solutions. The algorithm of *Coral* is presented in Algorithm 2.

5.2 Motivating Example

Dependabot [70] has been widely used as the most popular dependency security management extension at GitHub. One of the most popular Maven projects, *commons-lang* [210], adopted *Dependabot* to manage their dependencies. Nevertheless, the remediation caused build failure after upgrading [211] as in Figure 5.1. *Dependabot* has implemented the compatibility score by calculating the test passing rates from other repositories as the confidence score. However, in this case, the compatibility score was *unknown*. The compatibility score relying on knowledge of the crowd cannot guarantee a successful compilation without code-based compatibility calculation. Given the build failure, the developer did not adopt the suggestions provided by *Dependabot*, indicating a failed remediation. The root cause of such failure is that the compatibility indicator of *Dependabot* is not reliable, leading to the unexpected build failure. Instead, the ideal tool of remediation is supposed to prevent the breaking changes introduced while fixing the vulnerabilities.

Motivated by the motivating example, we studied the strategies of state-of-the-art remediation tools to understand, besides *Dependabot*, how existing tools handle incompatibility issues and how well they have done. After identifying the pros and cons of existing tools, we further studied the concerns of users regarding the remediation suggestions at GitHub to recognize what can be improved.

FIGURE 5.1: Common-lang3 build failure with *Dependabot* remediation

TABLE 5.1: Comparison of State-of-the-art SCA Tools Providing Remediation

Tool	Fix level	Fix target	Compatibility	S & C trade-off	Reachability	Dep conflict	Ripple effects	Unused dependencies
Steady	All graph	Vertex	◐	Sec first	●	○	○	○
Dependabot	Direct	Vertex	◐	Sec first	○	○	○	○
Com A	Direct	Vertex	○	Sec only	●	○	○	○
Com B	Direct	Tree	○	Sec only	○	○	◐	○

5.3 Preliminary Studies

5.3.1 Study of Remediation Strategies of Existing Tools

To understand the implicit reasons for the breaking, we first empirically compared the published remediation strategies of existing tools and then quantitatively evaluated them in Section 6.3. We only counted tools that provided actionable advice for dependencies, while tools that only offered multiple suggestions for vulnerabilities were out of the discussion because users would have to select the version out of multiple suggestions manually during decision-making for each library. The tools included *Dependabot*, *Steady*, and two popular commercial tools denoted by *Com A* and *Com B*.

- ***Dependabot***: *Dependabot* is able to create PRs to upgrade vulnerable dependencies to clean versions instead of providing an overall suggestion for the entire DG. As for the compatibility, *Dependabot* calculates the successful test rate of the upgrades from other repositories as the confidence score. However, this score can be unreliable because it is usually unavailable, and the compatibility ultimately depends on the context of the code base.

- ***Steady***: *Steady* is an open-source academic SCA tool with an open-source vulnerability database. *Steady* adjusts the versions of both direct and transitive dependencies to reduce the vulnerability risks at a fine granularity. Also, it utilizes the reachability analysis of vulnerabilities to filter out the unreachable CVEs with low risks. The reachability comprises both static and dynamic analysis, which only constructs call graphs once at the beginning. As for the version selection, *Steady* prioritizes the non-vulnerable versions, then determines the best candidate with the compatibility probability p. To derive p, it defines the reachable constructs (class, method, etc.) as *touch points* and calculates the percentage of present *touch points* in upgraded versions as p. The probability could be unreliable due to its uncertainty.

- ***Com A***: Towards a DG, *Com A* tweaks only the direct dependencies to remediate the vulnerabilities. The fundamental strategy is to upgrade the libraries with vulnerabilities to the closest non-vulnerable versions, as the closer versions usually are more likely to be compatible. The reachability is implemented by WALA [212] in a static manner to prioritize the critical reachable vulnerabilities. However, the compatibility of the remediation is not taken into account.
- ***Com B***: *Com B* conducts the remediation on the direct dependencies. The key feature is that *Com B* considers all vulnerabilities of transitive dependencies associated with the direct dependencies. Specifically, it iterates over all direct dependencies. For each, *Com B* attempts the version candidates and resolves the subsequent dependencies to measure the updated overall vulnerabilities. Then, *Com B* selects the version with the fewest overall vulnerabilities for this direct dependency. The strategy considers the *ripple effects* from the upgraded direct dependency to the upstream tree. However, as direct dependencies are usually not independent but inter-connected by transitive dependency relationships, the respective optimization of each direct dependency does not necessarily result in global optimization.

The comparison of SCA tools is demonstrated in Table 5.1. *Fix level* refers to the direct/transitive dependencies to be fixed. *Fix target* denotes the basic units that the tools optimize. *S&C trade-off* means the prioritization of determining the best candidates. The rest of the columns are summarized in the next section. *Ripple effects* is the support to handle the side effects brought by *ripple effects*. From the remediation strategies of tools, we found three major causes of incompatibility issues. **(1) No reliable detection**: Although *Steady* and *Dependabot* support compatibility scores, their results were unreliable due to inaccuracy. **(2) Lack of global optimization**: Because vertices in DG were interconnected with each other, optimizations of them were not independent. Thus, it is impractical to optimize each vertex individually without a global perspective. **(3) Lack of support of handling *ripple effects***: The optimization was conducted based on the original DG without updating structures and call graphs. Then, the optimal solutions based on the new DG were changed so that the existing tools would return sub-optimal solutions.

5.3.2 Study of Users' Concerns towards Remediation Suggestions

GitHub provides various automated SCA extensions to create PRs of security updates for dependencies, but these PRs are far from perfect, and thus sometimes rejected by users. To increase the acceptance rate of suggestions, we conducted a study to understand the concerns of users towards the remediation at GitHub by analyzing the reasons for rejected remediation suggestions and the accepted suggestions as a comparison.

Due to the lack of existing studies on Maven projects, the data set was collected by ourselves. First, we derived 9,527 projects active in the last three years with 100+ starts at GitHub. Then, 5,356 un-merged PRs created by bots were located and narrowed down to 306 PRs with human participation. Finally, we manually went through the comments in these PRs and summarized several reasons why PRs were unmerged.

- (91 cases, 29.74%) **Duplication**: The upgrades were superseded by other PRs, which were eventually merged.
- (82 cases, 26.80%) **Compilation/Test/CI failures and Dependency conflict (DC)**: The developers ran tests on the projects with upgraded dependencies, and incompatible issues occurred. Particularly, tests failed at dependency resolution, compilation, and test stages. For all PRs created by *Dependabot* in this category, compatibility scores were shown as *unknown*.
- (75 cases, 24.51%) **Incompatibility concerns**: The developers were concerned by incompatibility risks because either the upgrades had large spans, such as major upgrades, or they were known to be breaking. All compatibility scores were shown as *unknown* as well.
- (23 cases, 7.51%) **Internal errors**: Bots reported their internal errors in comments, so the users closed the PRs.
- (12 cases, 3.92%) **Unused dependencies**: The developers found the dependencies to be upgraded were not in use anymore, so the PRs were closed. The *bloated dependencies* were supposed to be ignored during the remediation.

- (9 cases, 2.94%) **Disobeying rules or absence of signed agreements**: The developers closed the PRs because the PRs failed to follow the rules of the repositories or sign the contributor agreements.

- (8 cases, 2.61%) **Unknown reasons**: The developers closed the PRs without explicitly mentioning the reasons.

- (6 cases, 1.96%) **Other**: There were various reasons: (1) Upstream projects demanded to keep the current version. (2) Java version was not compatible. (3) The PR introduced new CVEs. (4) Wrong user configuration. (5) A formatting issue.

From the result, excluding the duplicated PRs and unrelated reasons, such as internal errors, it is evident that the compilation/test failures and incompatibility concerns were the primary concerns of users (51.53%). The upgrades on unused dependencies could be avoided by the reachability analysis. The perspectives of concerns of users are demonstrated in Table 5.1. *Dep conflict* refers to the support of the detection of possible dependency conflicts raised by Maven. The *ripple effects* denotes the support of dynamically handling the *ripple effects*. *Unused dependencies* means the support of detecting and ignoring unused dependencies.

Besides the reasons for rejected PRs, merged PRs were also studied as a comparison, but they usually failed to include the reasons for acceptance. Thus, we studied the distribution of their upgrades. Since the number of merged PRs is enormous, we studied the $556,257$ PRs merged in the last two years for Maven projects. The distribution was (1) *Major*: 11.91%; (2) *Minor*: 38.34%; (3) *Patch*: 48.55%; (4) *pre-release*: 0.89%; (5) No SemVer available: 0.31%. The result indicated that most merged PRs (87.79%) did not bump the versions to major upgrades, which followed the criteria of SemVer because non-major upgrades were supposed to maintain backward compatibility. Therefore, the remediation suggestions with fewer major upgrades are more likely to be accepted by users.

5.4 Methodology

5.4.1 Problem Formulation

By summarizing the users' concerns, the problem of remediation in a DG can be defined as an optimization problem within the solution space based on the selection of versions. Then we are able to define the objectives and constraints of the remediation. The primary objective is to minimize the total vulnerability risks:

$$min \quad F_{vul} = \sum_{m=1}^{M} \sum_{vul=1}^{Vul} \theta_{vul} f_{cvss}(vul) \tag{5.1}$$

where M is the number of libraries and Vul is the number of vulnerabilities of a vertex m. f_{cvss} is the Common Vulnerability Scoring System (CVSS) [213] weight. θ_v is the reachability coefficient for vulnerability v, particularly, θ_v is larger for reachable vulnerabilities, because the reachable vulnerabilities are possible to be exploited by attackers. However, in reality, not all vulnerabilities are open-source, which also increases the difficulty for attackers. Thus, the vulnerabilities with uncertain vulnerable classes or methods are classified as *unknown vulnerabilities* whose severity is ranked between the reachable and unreachable vulnerabilities. Since different vulnerabilities result in different risks, we use CVSS, a normalized score provided by NVD, to prioritize the vulnerabilities with higher risks during calculation.

The remediation is less likely to be accepted if it breaks the users' projects, according to the study in Section 5.3.2. Thus, the pre-condition of successful remediation is the compatibility of version adjustments.

$$s.t. \quad c_{incom} = \sum_{m=1}^{M} \sum_{p=1}^{P} \theta_v * incom(v_p, v_m) = 0 \tag{5.2}$$

P is the number of parent vertices of v_m, while v_p is a parent vertex. c_{incom} is the total number of dependency relationships that cause incompatible issues. The incompatibility comprises two types of code-based breaking (semantic and syntactic breaking) and DC issues.

To achieve the global optimization and handle the *ripple effects* mentioned above, *Coral* is supposed to optimize all connected vertices altogether in a dynamically

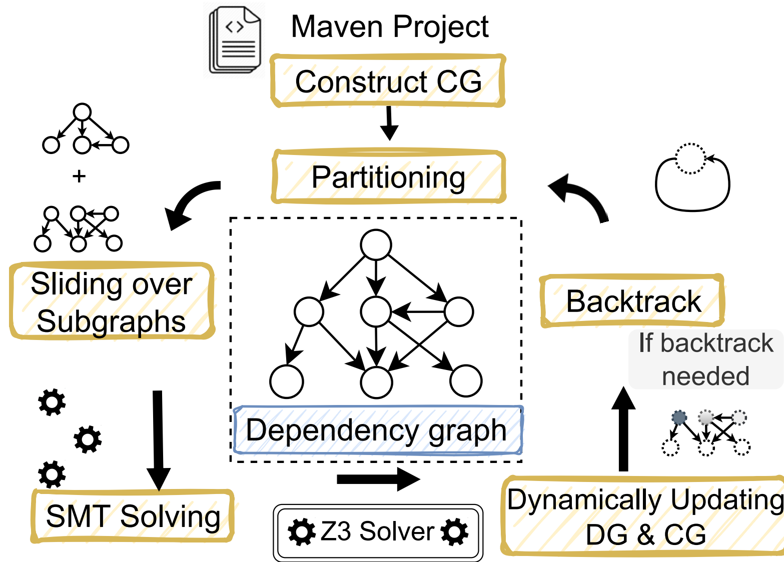


FIGURE 5.2: Overview of *Coral*

adjusted DG. These goals bring three challenges: (1) Trade-off between the security and the compatibility during decision making. (2) The time complexity increase exponentially with the size of DG as $O(n) = \prod_{n=1}^{N}$ if all solutions are to be iterated over. (3) The *ripple effects* requires dynamically updated DG.

5.4.2 Overview

Coral is implemented in four steps as illustrated in Fig. 5.2. (1) Generating DG and the call graph (CG) from the project object model (pom) file, a version control file of Maven, and class files of the project. (2) Partitioning the DG into subgraphs. (3) Optimizing the subgraphs regarding the vulnerability risks based on the pre-computed vulnerability mappings while ensuring compatibility. (4) Backtracking to parent vertices heuristically if the dead end is met. Then, the final remediation suggestions of version adjustment of all TPLs in the DG are returned.

5.4.3 Constructing Dependency Graph and Call Graph

With pom files and class files, *Coral* extracts the dependency tree by the Maven command and recovers the DG by completing the absent dependency relationships from a pre-computed dependency database. According to Maven documentation

[214], as dependencies with *test* scope are not involved in the normal use of the projects, *Coral* excludes dependencies with *test* scope from the DG. Specifically, DG is represented as $DG = Graph(V, E)$, where $V = \{e_i^x \mid i \in \{0, ..., N-1\}, x \in \{0, ..., L\}\}$ and $E = \{e_i \rightarrow e_j \mid i, j \in \{0, ..., N-1\}\}$. $\rightarrow$ denotes the direction of the calling edge, and x specifies the stack level w.r.t the DG.

The CG is constructed statically based on Soot [186] by the Spark algorithm [215] from the class files of the projects. The *main* methods are considered the entry points which serve as the start of the call graphs. If *main* methods are absent, we overestimate that it is possible to execute all methods implemented in the projects. Thus, all methods in users' projects are considered entry points. Since handling the *ripple effects* requires the dynamically updated CG to achieve real-time reachability analysis, the call edges in the CG are collected modularly. i.e. call edges are not extracted from a Uber jar [216] (root project with all dependencies) but from jars of each dependency separately and sequentially and then integrated into one graph originating from the root project. Particularly, for each dependency, the callers from the parent libraries serve as the entry points for child libraries. After the remediation, if the child libraries are suggested for other versions, the callees in them can be substituted accordingly to generate the real-time CG flexibly.

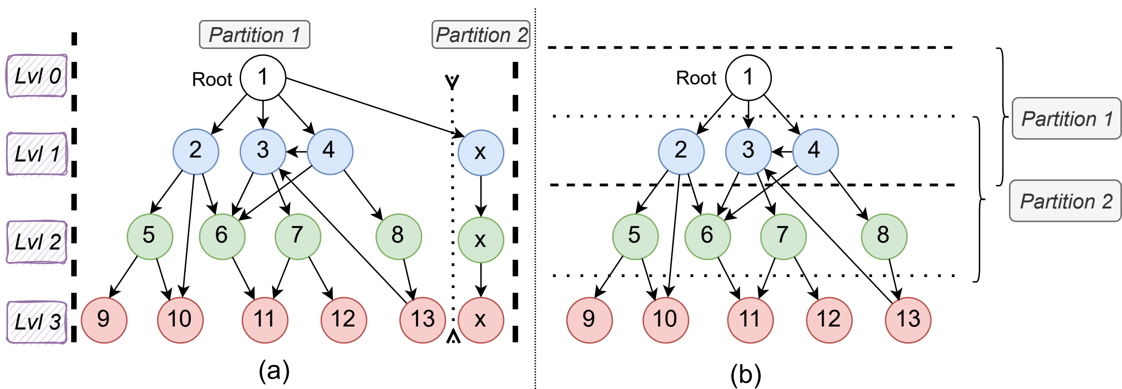


FIGURE 5.3: Dependency Graph Vertical (a) and Horizontal (b) Partitioning

5.4.4 Partitioning Dependency Graph

Due to the high complexity of optimization over the entire DG, *Coral* partitions the DG into subgraphs to reduce the size of the overall solution space. The partitioning comprises two steps, vertical partitioning, and horizontal partitioning.

As illustrated in Fig. 5.3 (a), the vertical partitioning iteratively splits the DG into multiple partitions that are not connected with each other by dependency edges except the direct relationships from the root project v_1 until all unconnected partitions are split. Since the direct dependencies do not depend on each other, optimizations on multiple partitions can be conducted independently and concurrently. For example, in Fig. 5.3 (a), *partition 1* and *partition 2* do not depend on each other. Hence, they can be partitioned to boost performance.

However, the vertical partitioning is not always sufficient, especially for the large partition at left in Fig. 5.3 (a). In this case, horizontal partitioning can further reduce the solution space. The subgraphs are partitioned by levels to preserve the semantics. According to [195], the semantics of a method decays along the calling chain, i.e. dependencies closer to the root matter more than those farther from the root in terms of the semantics or functioning they provide. For better notations, dependencies are labeled by tags called *level* to denote the smallest number of hops from the root. To better preserve the semantics of dependencies against the potential incompatibility, *Coral* split DG and group vertices at level l and $l-1$ into subgraphs as in Fig. 5.3 (b). Then, because the closer dependencies preserve more semantics, *Coral* starts the optimization from the root user projects in a top-down manner. Particularly, the lower-level dependencies should humor the upper ones in terms of the compatibility constraints as much as possible. Hence, *Coral* attempts to optimize dependencies in two adjacent levels at a time and then moves the sliding window of a partition down to the next level with a newly updated CG. With the horizontal partition, the complexity can be reduced to $O(n) = \sum_{p_{hori}=1}^{P_{hori}} \sum_{p_{vert}=1}^{P_{vert}} \prod_{n=1}^{N_p}$. The side effect is that the potential better solution with lower vulnerability risks may be overlooked for dependency edges across multiple levels. To compensate for the loss, Section 5.4.6 introduces the backtracking mechanisms to avoid sub-optimal situations.

5.4.5 Optimizing Subgraphs

In this subsection, the detailed specification of the optimization on subgraphs based on Z3 SMT solver [217] is described. As more secondary objectives were introduced to minimize the risks of incompatibility, the problem of remediation is formalized

as an Multiple Objective Optimization problem [218]. Specifically, the objectives and constraints are formally defined and explained.

In each subgraph, *Coral* conducts the optimization to minimize the vulnerability risks in the condition that the version changes are compatible. The vulnerability elimination follows the objective function in Equation (5.1). The basic vulnerability elimination strategy is to find versions with the fewest reachable and unknown vulnerabilities. Then, if more than one versions satisfy these conditions and other constraints, the versions without unreachable vulnerabilities are preferred.

Theoretically, the compatibility constraint is supposed to be strict. However, not all types of incompatibility can be accurately detected. Generally, there are three major types that *Coral* aims to resolve, namely, semantic breaking, syntactic breaking, and dependency conflicts, as discussed in Section 5.3.2. Except for semantic breaking, the rest can be detected statically and efficiently. Thus, the detection of the rest is integrated into the optimization as constraints:

$$s.t. \quad c_{synb} = \sum_{m=1}^{M} \sigma * synb(P(x'_m), x'_m) = 0 \tag{5.3}$$

The *synb*, Syntactic Breaking, is calculated based on the reachability analysis and the API compatibility checkers. For each version pair of one library, the modified APIs that can cause the failure of compilation are calculated by the three most widely used API compatibility checkers japi-compliance-checker [37], revapi [36], japicmp [219] based on the pair of jar files. Then, based on the reachability analysis, the called APIs of this library are obtained from CG. If any problematic APIs are called, the compilation would mostly fail, so *Coral* would label this candidate version as breaking and discard it.

$$s.t. \quad c_{dc} = \sum_{m=1}^{M} dc(P(x'_m), x'_m) = 0 \tag{5.4}$$

The DC issues are calculated based on Maven version rules [220]. Like other package managers, version ranges define the allowed versions for dependencies. If two version ranges required by dependents do not overlap, Maven would report *Dependency Conflict* during version resolution before the compilation. A similar logic is implemented in *Coral* to only select versions within the intersection of ranges defined by dependents. According to our analysis on the POM files crawled from Maven Central Repository [18], it is noteworthy that over 99% dependency version specifications are not determined with ranges, but single recommended

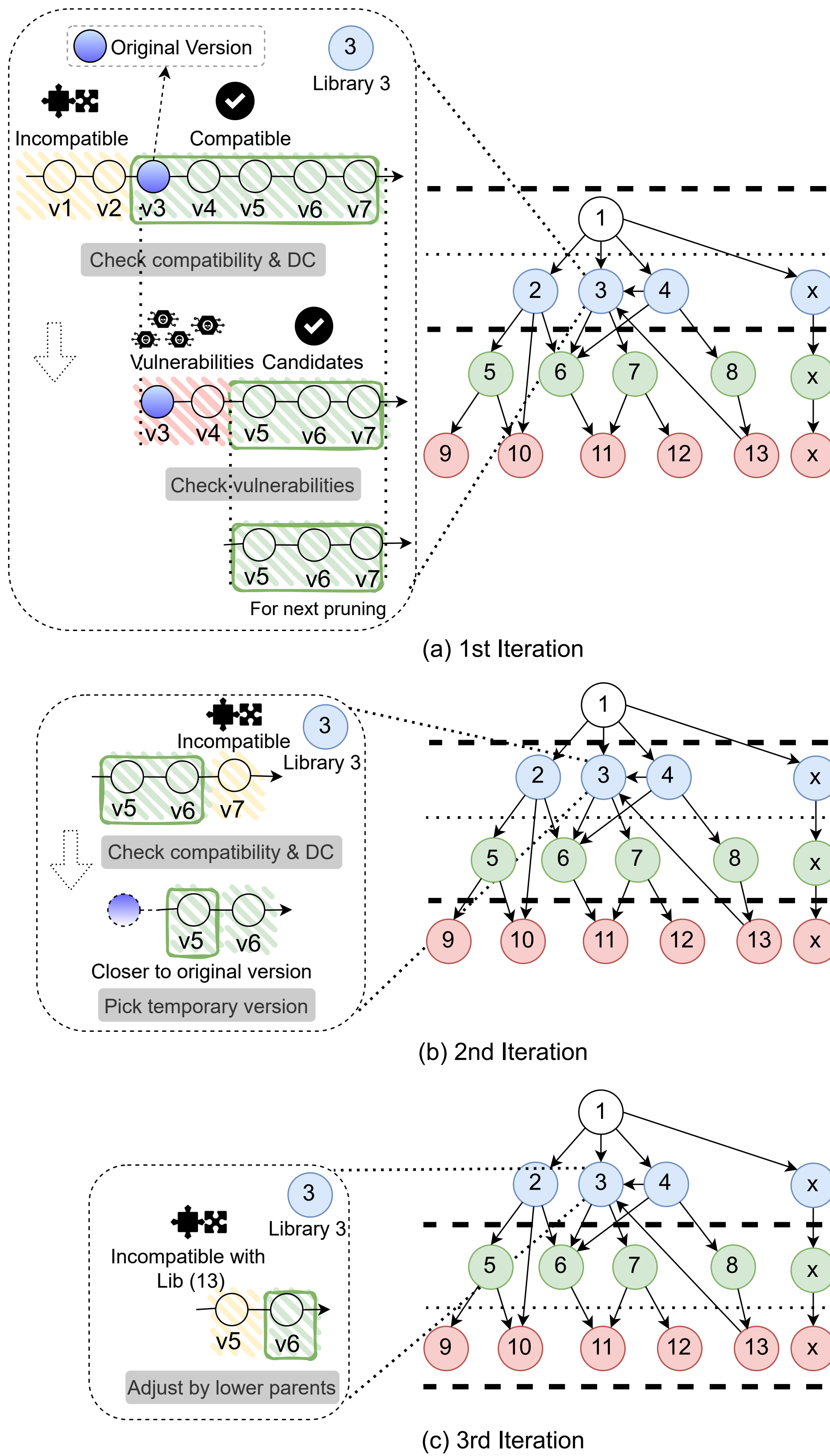


FIGURE 5.4: Example of the Version Selection of a Dependency

versions instead, which means all versions are available regardless of compatibility. In this case, *Coral* would include all versions as candidates for DC detection as Maven. The statistics were derived from POM files of 11, 859, 883 versions as of the experiment date. Initially, I crawled all pom files from the Maven Central Repository [18], and then I extract all dependencies from them in the `dependencies` section with selected versions by `effective-pom` plug-in to ensure the consistency. Finally, the versions specified for dependencies were classified as either version ranges or single versions.

Since the semantic breaking is usually revealed by unit tests subject to limited coverage according to [38], it is hard to detect it statically and efficiently. Also, it is the leading cause of unit test failures [39], which is one of the main reasons why users reject remediation suggestions. Thus, *Coral* relies on auxiliary information to infer the potential semantic breaking and minimize its probability by following the SemVer and Maven versioning guides. According to SemVer, the *Major* upgrades are allowed to break the original implementations. Hence, *Coral* avoids using *Major* upgrades/downgrades as much as possible unless they are less vulnerable and satisfy the other compatibility criteria. Thus, besides the primary objective, we add a secondary objective, f_{major}, the number of dependencies that have *Major* upgrades/downgrades:

$$min \quad f_{major} = \sum_{n=1}^{N} f_{major,x_n}(x_n, x'_n) \tag{5.5}$$

$$\text{where } f_{major} = \begin{cases} 0 & \text{if } x_n \text{ to } x'_n \text{ is not major} \\ 1 & \text{if } x_n \text{ to } x'_n \text{ is major} \end{cases}$$

Although SemVer stipulates *Minor* should not include incompatible changes, researchers from [138] found that *Minor* upgrades are not as compatible as *Patch* upgrades, which generally introduce more breaking changes. Therefore, *Coral* always prefers *Patch* upgrades rather than *Minor* if all other conditions stand. Another secondary objective function of f_{minor} is created to fulfill the purpose.

$$min \quad f_{minor} = \sum_{n=1}^{N} f_{minor,x_n}(x_n, x'_n) \tag{5.6}$$

$$\text{where } f_{minor} = \begin{cases} 0 & \text{if } x_n \text{ to } x'_n \text{ is not minor} \\ 1 & \text{if } x_n \text{ to } x'_n \text{ is minor} \end{cases}$$

Besides SemVer, Maven version control rules [220] also help identify potentially

breaking versions. First, the pre-release versions, also known as development versions, such as *alpha, beta, SNAPSHOT* versions, are unstable and prone to breaking changes, which are selected at a lower priority than *Major* upgrades. Second, the larger version spans are usually more likely to induce incompatible changes. *Coral* attempts to reduce the version span from the original version to the new version as much as possible. In terms of these two objectives, the functions of f_{dev} and f_{span} are formally given as:

$$min \quad f_{dev} = \sum_{n=1}^{N} f_{dev,x_n}(x_n, x'_n) \tag{5.7}$$

$$\text{where } f_{dev} = \begin{cases} 0 & \text{if } x'_n \text{ is not dev} \\ 1 & \text{if } x_n \text{ is not dev, } x'_n \text{ is dev} \end{cases}$$

$$min \quad f_{span} = \sum_{n=1}^{N} dist(x_n, x'_n) \tag{5.8}$$

where $dist(x, y)$ is the distance between x,y in sorted versions.

After solving with the SMT solver, each vertex in the subgraph is assigned with a selected version, and upgraded libraries in CG will be updated accordingly. However, the selected versions can be overthrown by the next optimization. Thus, all selectable candidate versions are saved and fed to the next optimization. For instance, in Fig. 5.4 (a), *Lib* 3 initially has 7 candidates and gets filtered to 3 by incompatibility and vulnerabilities. In the next iteration (b), *Lib* 3 has its candidates further filtered to 2 because of the incompatibility. Then, $v5$ is selected due to its smaller version span from the original version. However, in the third iteration (c), $v5$ is overthrown because it is not compatible with the parent library *Lib 13* at a lower level. Since the compilation and Maven resolution would fail regardless of the levels, the selected versions must follow the constraints in Equations (5.3) and (5.4). Therefore, $v5$ is discarded, and $v6$ with compatible changes is selected.

5.4.6 Backtracking

Although sequential partitions of DG reduce the complexity, they could lead to suboptimal solutions and dead ends. To mitigate such issues, two types of backtracking mechanisms are implemented in *Coral*, the hard and the soft backtracking.

5.4.6.1 Hard Backtracking

Hard backtracking is implemented to avoid dead ends. It happens during deciding the best version of a library where all versions disobey the constraints by potentially breaking the project. The backtrack targets are parent libraries of the current library. Since backtracking requires re-visiting the related vertices, the parent library at the lowest level is prioritized to reduce the efforts of re-visiting. And then, the higher ones are attempted if the lower parent triggers the backtrack again. During one backtrack, the selected version of the target parents is temporarily marked as incompatible, and other versions are attempted.

5.4.6.2 Soft Backtracking

Soft backtracking is used to avoid sub-optimal solutions. It is triggered when the version selected by the SMT solver is not the version with the lowest vulnerability risks in the version list, such as non-vulnerable versions. Like the hard backtrack, the soft backtrack prioritizes the parent libraries at lower levels. The different part is that soft backtrack does not mark the parent's current version as incompatible but unpreferrable instead. It means if other versions are proven to be not as optimal as the unpreferrable version after the backtracking, the unpreferrable would still be selected. Thus, even if versions satisfy the constraints, they could be ignored by soft backtracking. During the soft backtracking, *Coral* saves the overall vulnerabilities between the backtracked library and the target parent for future comparison. After the backtracking, *Coral* compares the vulnerabilities of the current run with the ones saved previously and adopts the run with the fewest vulnerabilities to apply the versions to backtracked libraries accordingly. Note that to avoid an infinite loop, soft backtracking would not be triggered again during one run of soft backtracking. Also, if the hard backtracking is triggered during soft backtracking, the current run would be discarded, and other versions would be attempted.

In conclusion, *Coral* was designed to overcome the challenges of the high complexity of global optimization and *ripple effects*. The algorithm is presented in Algorithm 2. *Coral* starts with vertical and horizontal partitions to split the DG into multiple parts. Then, the SMT solver is used to optimize the remediation results in each

partition in a top-down manner. If any backtrack is triggered, *Coral* backtracks to the previous vertices to avoid the sub-optimal solutions.

5.5 Evaluation

We aim to answer the following research questions:

RQ1: How is *Coral* compared with other cutting-edge remediation tools regarding security and compatibility?

RQ2: How effectively does *Coral* resolve the challenge of global optimization by subgraph partitioning?

RQ3: How many vulnerabilities CAN/CANNOT be fixed without breaking the projects in the Maven ecosystem?

5.5.1 Preparation

5.5.1.1 Data Collection

To build a data set of in-development Maven projects, we collected 301 most starred projects managed by Maven at GitHub on May 21st, 2022. We first selected Java projects with the most stars from GitHub and excluded non-Maven projects. Next, we manually modified the POM files of each project to apply the remediation suggestions from these tools. Considering the efforts of manual work, we filtered these projects with 1K+ stars. Finally, we got 301 selected projects. The demographics of the data set are illustrated in Fig. 5.5. It has the following features: (1) The code base size is non-trivial (average 22.19 kloc). (2) The range of sizes of dependency graphs is large (max 327, average 32.0). (3) The projects are affected by an adequate number of CVEs (average 27.6). (4) The projects are popular due to high star numbers.

To experiment with accurate vulnerability mappings, we periodically crawled CVE feeds from NVD [19] with a pipeline and pre-classified the language of CVEs by keyword matching. As the CVE descriptions are free-text [221, 222], it is impractical to directly extract version mappings from them. Hence, we manually triaged

Algorithm 2: Algorithm of *Coral*

Input: Dependency Graph $G\langle V, E\rangle$ (vertices V and edges E) with h levels, class files cf of the project

Output: Remediated $G'\langle V', E'\rangle$ with newly assigned versions

```
1  parts_v ← verticalPartition(G)
2  foreach part in parts_v do
3      foreach i_th in h do
4          part_h ← V_i + V_{i+1}
5          cg ← CallGraph(part_h, cf)
6          foreach v in V do
7              parents ← parentsOf(v)
8              foreach ver in versionsOf(v) do
9                  if ver has synb or DC then
10                     cand.remove(ver)
11                 if sizeOf(cand) == 0 then
12                     hardBacktrack
13                     break
14                 vuls ← vulsOf(ver)
15                 foreach vul in vuls do
16                     θ ← reachability(vul, cf)
17                 sort candidates by θ
18             s ← SMTsolver(V_i, V_{i+1})
19             if vuls(s)! = min(vuls)) then
20                 softBacktrack
21                 break
22             cg ← updateBy(s)
23             G ← updateBy(s)
24             if hardBacktrack then
25                 p ← parent_lowest
26                 p.incompatible ← ver
27                 backtrack to p
28             if softBacktrack then
29                 p ← parent_lowest
30                 runs ← saveVul(p)
31                 backtrack to p
32                 foreach r_th run in p.vers do
33                     runs ← saveVul(p_r)
34                 s ← min(runs)
35 return G' ⟨V', E'⟩
```

the mappings from reference links and associated Common Platform Enumerations (CPEs) [223]. So far on May 21st, 2022, we collected mappings for 1,759 CVEs associated with Maven libraries. In this section, the evaluation needs the reachability analysis, which requires the vulnerable methods and classes associated with CVEs. Thus, we first identified 750 CVEs (42.64% of all Maven CVEs) from 2,326 unique libraries used as dependencies in 301 projects. Then, vulnerable classes and methods of 300 CVEs were successfully identified and manually collected from the patches available at NVD links. The mappings and vulnerable methods of lib-vers and CVEs are publicly accessible on our website [191].

5.5.1.2 Tools and Environments Preparation

All tools used in the evaluation were tested with their latest versions in May 2022. *Steady* was tested with version 3.2.4 with a built-in vulnerability database including 729 CVEs. The two commercial tools were evaluated in their publicly accessible production environments. *Coral* was implemented with 6.9 kloc in Python 3.8.2 and evaluated with java $7-13$ (depends on projects), Maven 3.8.2, and Ubuntu 18.04.6.

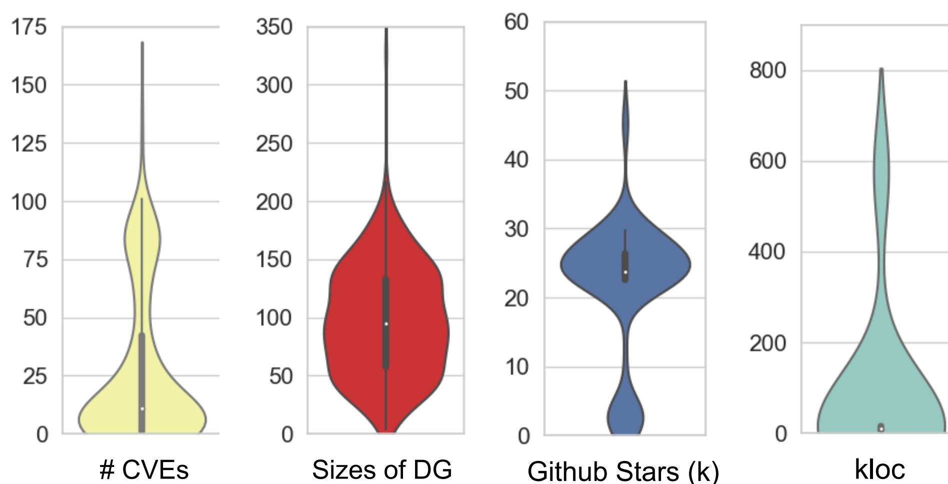


FIGURE 5.5: Demographics of the Data Set of RQ1 and RQ2

5.5.2 RQ1: Comparison with Other Remediation Tools

5.5.2.1 Evaluation Metrics

(1) *Vulnerability fixed*: The primary target of remediation is fixing vulnerabilities which are further classified in terms of reachability as remaining reachable CVEs: Vul_r, remaining unreachable/unknown CVEs: Vul_{ur}/Vul_{uk}, and total fix: Fix. (2) *Compilation*: $Fail_{comp}$. The projects with updated pom files were compiled by Maven to evaluate the correctness of Maven resolution and the compile-time compatibility. (3) *Unit test*: $Fail_{test}$. The affiliated unit tests were run against the remediated projects to evaluate the runtime compatibility. (4) *Supplementary metrics*: The number of upgraded/downgraded libraries, total version span, number of *Major* upgrades (#Major), and development upgrades (#Dev) were counted for reference.

5.5.2.2 Comparison Results

The evaluation was conducted based on the remediated projects (versions returned by remediation tools were adjusted in pom files), which along with the Maven logs, are available on our website. To emphasize the improvement gained from the version selection strategy, we added two baseline tools with naive strategies. Both baseline tools share the same partitioning and backtracking mechanisms as *Coral*. *Baseline A* always prefers the latest versions of vulnerable libraries. It is used to demonstrate the result of a common practice which is upgrading vulnerable dependencies to the latest. *Baseline B* always prioritizes the versions with the fewest reachable and unknown vulnerabilities, even if it may break the projects. *Baseline B* gave an idea of how many non-trivial vulnerabilities could be fixed without being constrained by compatibility. The comparison results with remediation tools and baselines are provided in Table 5.2. The annotation meanings are: Vul_r: number of reachable CVEs. Vul_{ur}: number of unreachable CVEs. Vul_{uk}: number of *unknown* CVEs. $Fixed_{CVE}$: number of fixed CVEs. $Fail_{comp}$: number of projects with failed compilation. $Fail_{test}$: number of projects with failed tests. *Crashes*: number of projects that tools crashed and failed to return results. *Dev*: number of development version pairs. *Major*: number of *Major* version pairsThe analysis of each metric is supplied as follows:

- **Remaining Reachable Vuls**: Due to a limited number of vulnerable methods, only 17 CVEs could be identified as reachable in original projects. It is noteworthy that *Coral* eliminated all reachable CVEs. Because *Dependabot* returned far fewer remediation suggestions than other tools, 16/17 reachable CVEs remained reachable after remediation. As *Steady*'s vulnerability database is limited, we reevaluated *Steady* with the 729 CVEs in their database and enclosed the updated numbers in brackets. Within this scenario, *Steady* had fewer reachable CVEs than before, like other tools.

- **Remaining Unreachable and Unknown Vuls**: *Coral* had much fewer unreachable vulnerabilities (reduce 87.56% of vulnerabilities) than other tools because though the unreachable vulnerabilities were considered harmless, *Coral* attempted to remove them if the constraints allowed. Unknown vulnerabilities 583 still remained in the DG for three reasons: (1) $244, 41.87\%$. The versions with fewer vulnerabilities did not satisfy the constraints. (2) $149, 25.56\%$. All versions were vulnerable. (3) $101, 17.31\%$. The more secure versions with unreachable CVEs were *Major* upgrades with overly large version spans. Regarding baselines, *Baseline A* proved that upgrading to the latest fixed only an insignificant amount of vulnerabilities. *Baseline B* suggested that 338 (4.05%) more vulnerabilities could be fixed without considering compatibility.

- **Compilation Failures**: *Coral* achieved 98.67% successful compilation rate due to detecting syntactic breaking and DC issues. The reasons for four failed cases were (1) Call graph generation failure: One of the libraries along the call chain had no call edges generated, which led to unreachable breaking methods. (2) Exception class not captured: The breaking exception class was not captured in the call graph and thus deemed unreachable. This is due to the limited exceptions captured statically, which may not cover all potential exceptions thrown at runtime. (3) Overriding not captured: The breaking methods of a class were extended and overridden in the new version, but the call graph did not reflect such overriding. For example, a project, *apollo-client* [224], had a failed compilation due to the incompatibility in its dependency, *snakeyaml*. The overriding of class, *BaseConstructor*, was not captured. (4) Ghost dependency: The breaking methods were used in an undefined dependency, so they were not captured as reachable methods. Because of the local optimization and unreliable or absent compatibility detection, the rest of the tools were subject to broken upgrades with failed compilation.

TABLE 5.2: Comparison of *Coral* among State-of-the-art Remediation Tools

Tool name	Size	Vul_r	Vul_{ur}	Vul_{uk}	$Fixed_{CVE}$	$Fail_{comp}$	$Fail_{test}$	#Crashes	#Libs	Span	#Dev	#Major
Original	33.99	17	5,363	2,954	0	0	0	0	0	0	0	0
Coral	**36.27**	**0**	**553**	**583**	**7,198**(87%)	**4**	**15**	**0**	**2,556**	**70,464**	**3**	139
Dependabot	34.93	16	5,357	2,682	262 (3%)	20	31	1	602	17,024	0	44
Steady	44.17	11(4)	1,596(955)	1,457(515)	5,253(63%)	27	36	1	2,292	75,380	4	257
Com A	34.24	7	4,199	2,410	1,469 (18%)	51	61	7	1,398	24,679	0	245
Com B	35.61	3	1,040	1015	6,277(75%)	54	70	0	6,498	134,407	0	170
Baseline A	33.81	3	4,677	2,786	869	39	45	0	2,580	16,863	7	194
Baseline B	43.11	0	422	376	7,536	54	71	0	5,860	90,931	5	329
Baseline C	35.11	0	535	547	7,252	4	12	0	2,613	56,738	1	126

Size: average size of DG in terms of nodes.
Vul_r: number of reachable CVEs.
Vul_{ur}: number of unreachable CVEs.
Vul_{uk}: number of *unknown* CVEs.
$Fixed_{CVE}$: number of fixed CVEs.
$Fail_{comp}$: number of projects with failed compilation.
$Fail_{test}$: number of projects with failed tests.
Crashes: number of projects that tools crashed and failed to return results.
#Libs: number of libraries upgraded/downgraded.
Span: total version span of all libraries.
Dev: number of development version pairs.
Major: number of *Major* version pairs

- **Unit Test Failures**: Since it is challenging to detect Semantic Breaking effectively, it is difficult to prevent Unit test failures. Thanks to the prioritization based on SemVer and Maven resolution rules, *Coral* was able to achieve the fewest failures among these tools. Note that due to private dependencies, unfinished development, special requirements of running environments, etc., 88 unit tests in original projects already failed without remediation, which was excluded from the number of failures in the table.

- **Other Statistics**: It is evident that *Com B* had many more lib-ver pairs changed because it manipulated the direct dependencies to adjust the associated trees by changing the default versions of subsequent dependencies regardless of vulnerabilities, while other tools mostly focused on the vulnerable vertices. The same reason stood for the version span. Because *Coral*, *Steady*, and *Com B* substantially changed the versions of transitive dependencies, their total version spans were larger than *Dependabot*'s.

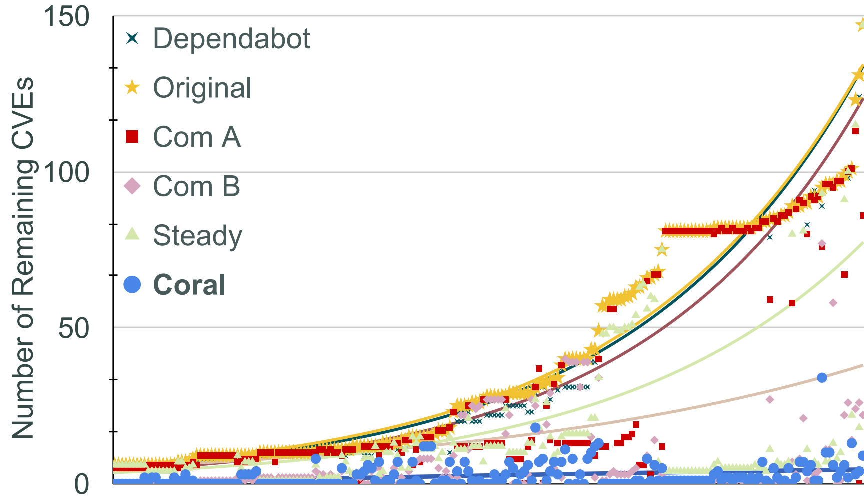


FIGURE 5.6: Ascending Order by Numbers of CVEs of Original per Project

To illustrate the distribution of remaining vulnerabilities over all projects, the remaining CVEs of all tools are presented in the scatter plot of Fig. 5.6. The x-axis is ordered by the number of CVEs of the original, which serves as the upper bound denoted by yellow stars. It is evident that *Coral* has the overall fewest remaining CVEs at the bottom of the chart, denoted by blue dots.

Conclusions of RQ1: From the evaluation in Table 5.2, *Coral* fixed 87.56% of all CVEs with all reachable removed, including 911 more CVEs than the best of the rest tools. Meanwhile, *Coral* achieved the 98.67% successful compilation rate and 92.96% successful unit test rate, which outperformed the rest of the tools. Compared with the two baseline tools, *Coral* was proven to be effective at balancing the compatibility and security by breaking 106 (35.21%) fewer projects at the cost of 338 (4.05%) fewer vulnerabilities fixed.

5.5.3 RQ2: Effectiveness of Improvement on Global Optimization

The subgraph partitioning was implemented in *Coral* to boost the performance towards the global optimization. To evaluate the effectiveness of partitioning, *Baseline C* was implemented in the same logic without two types of partitioning used by *Coral*. The same data set was used to evaluate the existing metrics and time consumption. To measure the time, we respectively ran *Coral* and *Baseline C*

ten times against each project and calculated the average time as the final result. The result is presented in Fig. 5.7, which illustrates that the *Baseline C* generally tended to spend more time than *Coral* for complete remediation. In the figure, the 301 projects are ordered by the size of DG. Each dot in the figure represents a single project. The tendency curves of both are fitted by the second-degree polynomials to avoid over-fitting.

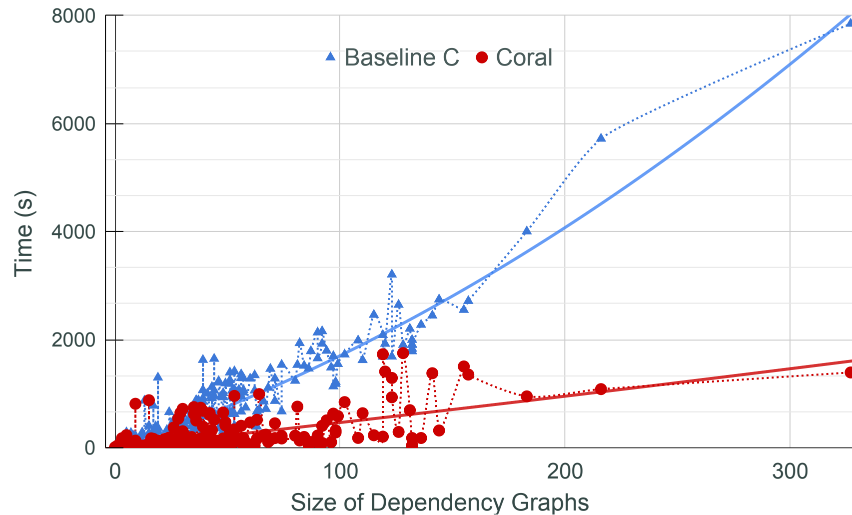


FIGURE 5.7: Time Consumption of *Baseline C* and *Coral*

To explain the fluctuations of the consumed time of *Coral*, we manually analyzed the causes of the outliers. First, the 6 lower outliers were collected and analyzed. The cause of these cases was subgraphs partitioned were pretty small (1-5 deps), and the backtracking was not triggered. Second, for 18 higher outliers, there were four major causes:

- (9 cases) **Call graph generation failures**: The Call graph generation of the Soot script failed at some dependencies of DG, which took a long time to return. Usually, the failure of one version would persist with other versions of the same library, so the total time was elongated.
- (5 cases) **SMT solver took a long time**: For these projects, both *Baseline C* and *Coral* spent a long time because the SMT Solver took a long time to finish. The direct reason for this cause was that the levels of DGs were limited, which means the DGs were more flattened than others. Thus, the partitioning of *Coral* based on levels could still include a substantial number of vertices in the SMT solver.

- (3 cases) **Multiple backtracking**: Another backtracking could be triggered during the current run of backtracking or after the current run fails. In these cases, 2 out of 3 cases had over three attempts of failed hard backtracking, and the rest triggered the hard backtracking multiple times during soft backtracking, which led to no improvement of vulnerability reduction for this soft backtracking.
- (1 case) **Jar downloading failure**: The CG generation and Syntactic breaking detection relied on the jar files of dependencies, *Coral* failed to download from Maven Repository with time-out multiple times.

The observed metrics for *Baseline C* are presented in Table 5.2. From the table, *Baseline C* has fixed 54 (0.75%) more vulnerabilities than *Coral*, which implies the global optimization without partitioning has slightly improved the vulnerability fixing. Moreover, the number of projects with failed compilation stayed the same because *Coral* handled the syntactic breaking and DC issues regardless of the partitioned subgraphs by backtracking.

Conclusions of RQ2: The comparison between *Baseline C* and *Coral* substantiates that the partitioning mechanism could substantially reduce the time consumption without introducing the compilation failures at an acceptable cost of 0.75% fewer fixed vulnerabilities, especially for the large DG.

5.5.4 RQ3: How many fixable/unfixable CVEs in Maven

We target finding out how many vulnerabilities can be fixed without breaking the compilation and how many cannot in popular Maven projects. Since *Coral* could efficiently exclude the solutions that broke the compilation with high precision (98.67%), we made an assumption that CVEs fixed by *Coral* were fixable and CVEs not fixed by *Coral* were unfixable.

5.5.4.1 Preparation of data

To conduct a large-scale study in the Maven ecosystem, we constructed a different data set from RQ1 and RQ2. Considering the balance between the representativeness and quality of the dataset, we first collected repositories with 100+ stars managed by Maven from GitHub to ensure the high quality of the dataset. Then,

we compiled them and extracted dependency trees from them by the Maven command. If both steps succeeded, the dependency trees and class files were used as input for the remediation. Eventually, we randomly selected 2,000 out of 6,898 projects (average size 103.58) for the evaluation to make sure the dataset was representative. As for CVE mappings, the same mappings were used as RQ1 and RQ2. Since collecting vulnerable methods and classes is not as straightforward as version mappings, which requires much more effort for all CVEs, we decided not to conduct the reachability analysis of vulnerabilities in the experiment.

5.5.4.2 Results of RQ3

Fixable: The fixable CVEs are 10,109 (78.45%) as in Fig. 5.8. It is inferred that around 78% vulnerabilities could have been safely eliminated from the popular Maven projects without breaking the compilation to reduce the vulnerability risks of the ecosystem. We further calculated the distribution of the CVEs regarding the levels of the libraries and the types of upgrades that removed the CVEs. Although 78% seems to be a large number, the majority of them could not be fixed without domain knowledge or the aid of *Coral*. According to Fig. 5.8, the proportion of vulnerabilities that could be fixed by adjusting direct dependencies was 11.71%, out of which 8.34% belonged to *Minor* and *Patch* upgrades.

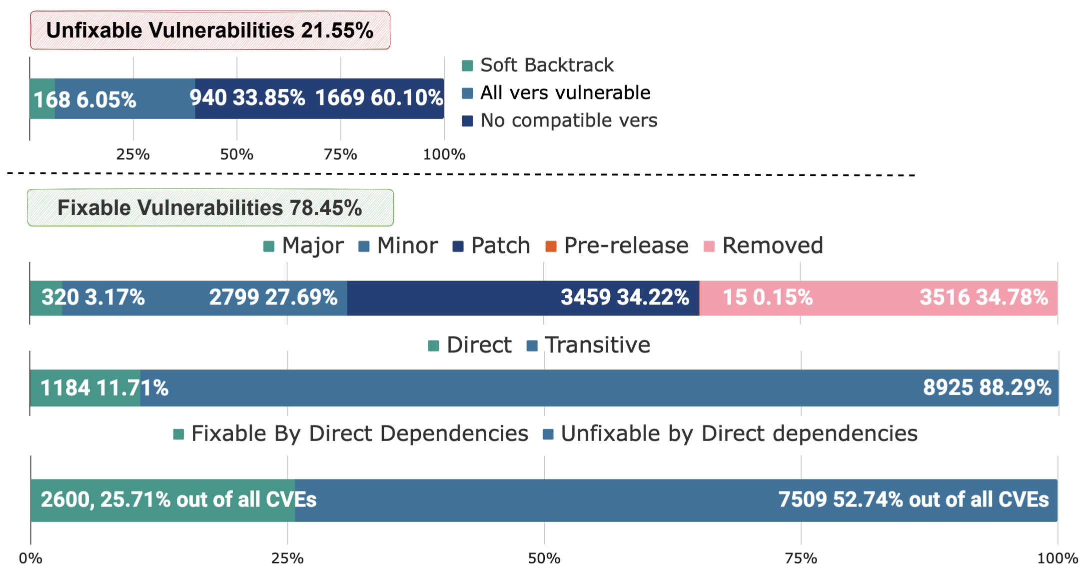


Figure 5.8: Distributions of Fixable/Unfixable Vulnerabilities

As users can straightforwardly upgrade their direct dependencies to non-major versions to fix vulnerabilities on their own, we applied this naive method for the

comparison with *Coral*. The result showed that 25.71% of CVEs can be fixed by upgrading direct dependencies. Due to *ripple effects*, not only were 8.34% in direct dependencies fixed but more CVEs in transitive dependencies were also fixed. It is implied that without the aid of *Coral*, the rest of the fixable vulnerabilities (52.74%) could not be fixed straightforwardly.

Unfixable: The number of unfixable CVEs was 2,777 (21.55%) as in Fig. 5.8, which could not be fixed by *Coral* for three major reasons, the soft backtrack, all versions of a library were vulnerable, and the secure versions were incompatible. Reflected in Fig. 5.8, it is observed that the major reason was incompatibility which accounted for 60.10%. Note that the incompatibility did not count the Semantic Breaking because it could not be reliably detected. The minor reason, soft backtrack, refers to the vulnerabilities being left unfixed because the soft backtrack could not eradicate all CVEs, but minimized the overall vulnerabilities by ignoring some CVEs.

Although unfixable vulnerabilities cannot be easily removed without breaking the projects, some of them are removable at an acceptable cost. For example, if an API is deprecated and migrated to another, users only have to invoke the updated API and upgrade to the target version to fix the issue and vulnerabilities. Thus, if efforts to fix incompatibility are acceptable, more vulnerabilities can be fixed thoroughly with minimized efforts by quantifying efforts to fix the incompatibility.

Conclusions of RQ3: Through the experiments with the most starred projects on GitHub, we found that 78.45% of vulnerabilities could be fixed without breaking the compilation. However, without the aid of *Coral*, only 25.71% could be fixed by upgrading the direct dependencies to non-major secure versions.

5.6 Threats of Validity

The threat of *Coral* is the static call graph reliance because only the static call graphs are not accurate enough to capture all possible call edges, which is one of the causes of the unit test failures in Section 5.5.2.2. One typical example of inaccurate static call graphs is that static call graphs may miss invocations made by dynamic features, e.g., reflection. Moreover, the prioritization of vulnerabilities

might overlook some reachable ones due to inaccurate static call graphs. However, to modularly and dynamically update the call graphs after each version adjustment, we could only generate static call graphs that are faster than dynamic ones. As it is impractical to run tests and generate dynamic call graphs thousands of times per project, we sacrificed accuracy for better performance.

5.7 Conclusion

We proposed *Coral* to provide remediation without breaking compatibility. The evaluation demonstrated that *Coral* outperformed other tools by fixing 87.56% of vulnerabilities and achieving 98.67% successful compilation rate and 92.96% successful unit test rate. In the ablation study, the partitioning of DG and trade-off between security and compatibility had been proved effective. Furthermore, we found that 78.45% of vulnerabilities in popular Maven projects could be fixed without breaking the compilation.

Chapter 6

Persistent Vulnerability Study

6.1 Overview

This chapter aims at presenting a comprehensive study of the persistent vulnerabilites in the Maven ecosystem. Specifially, the vulnerabilities linger in the ecosystem because, even after the patches are released, downstream projects could be transitively vulnerable by using vulnerable versions automatically resolved by Maven. The patched versions are not rapidly propagated to downstream users, leading to persistence of vulnerabilities. This blocked patch versions are not exclusively to Maven. However, Maven is severely affected because over 99% of dependencies versions are stipulated single versions instead of version ranges, resulting in confined flexibility. Therefore, I aimed to address the blocked patch versions, i.e. persistent vulnerabilities in the Maven ecosystem. In this chapter, an empirical study of vulnerability propagation through dependency relationships in the Maven ecosystem is first conducted. Then, the root cause is uncovered with a solution proposed.

6.2 Preparation for Empirical Study

To commence our study, we constructed a dependency graph using data sourced from both the Maven Central Repository (MCR) and NVD. Based on the dependency graph, we developed a searching algorithm (ALSearch) to facilitate tracking

of affected libraries throughout the course of our study.

6.2.1 Background of SemVer in Maven

Within the Maven ecosystem, most version numbers adhere to the SemVer standard [17]. This standard consists of three digits: *Major*, *Minor*, and *Patch*. Major upgrades, which change the *Major* digit, are the only type of upgrade that allow for incompatible changes. Version ranges [225] supported by Maven rely on SemVer. However, 99.21% of dependency version specifications in Maven are single versions which are called Soft Version Constraints [226] (SoftVer). The SoftVer stipulates the preferred version for a dependency so that Maven mostly resolves the preferred versions for the dependencies [158].

6.2.2 Infrastructure of Study

6.2.2.1 Dependency Graph for Maven

A dependency graph was constructed, including vulnerabilities, as an infrastructure for the empirical analysis. As of 01 Apr 2023, MCR contained 541,753 libraries and 11,859,883 versions, both of which were extracted from the MCR index [18] and added to the dependency graph as *Library* and *Version* vertices. 82,708,563 dependency edges from *Version* to *Version* were extracted from the Project Object Model (POM) files including properties specifically designed to regulate the dependency resolution. We used the approximately over 2k Common Vulnerabilities and Exposures (CVE) for Maven libraries at NVD as vulnerability data. Due to the absence of well-formatted mappings between vulnerabilities and versions, 1,861 vulnerabilities and their mappings were collected after cross-checking multiple sources from Github Advisory [21], Google Open-Source Database [227], and Snyk Vulnerability Database [22], which are available on our website [191].

6.2.2.2 Search Algorithm

We developed a precise Affected Library Searching Algorithm (ALSearch) that leverages the Maven dependency resolution rules to accurately track dependents

of vulnerabilities based on the dependency graph. Unlike the forward resolution approach used by Maven to resolve dependencies from the root to leaf vertices, ALSearch was designed to facilitate backward tracking from vulnerability vertices to dependent vertices. As ALSearch is tailored for backward tracking, its rules have been adapted accordingly, and are outlined below:

Scope is a feature to limit the transitivity of a dependency. Out of the six scopes, only *compile* and *runtime* are inheritable and tracked by ALSearch. **Optional** dependencies are not transitive, and thus should not be tracked for dependents with ≥ 2 depth. **Exclusions** are used to exclude certain versions of transitive dependencies. All transitive dependencies under the *exclusions* are excluded. Hence, if the libraries with vulnerabilities are excluded by any dependent, dependents should not be tracked. **Multiple versions selection**: If a library is used with different versions in a dependency tree, Maven would prioritize the version specified first during a Breadth-First Search (BFS) resolution from direct to transitive dependencies. ALSearch considers a target library affected only if the vulnerable versions of the affected library are closer to the target library than the non-vulnerable versions.

Incorporating the above rules, for each vulnerability, ALSearch iterates over downstream libraries in a BFS manner. During each iteration, it includes two procedures to track a dependent and validate the tracked target respectively:

- **Dependents tracking**: Check if the *Version* vertex has consecutive dependency edges pointing to any vulnerable version of a library affected by a *Vulnerability* vertex. If yes, check if the properties on dependency edges adhere to the aforementioned rules. If yes, proceed to the next procedure.
- **Dependencies validation**: Resolve dependencies of the target *Version* vertex following normal Maven dependency resolution rules in a reversed direction until the version of the vulnerable library is resolved. If the resolved version is vulnerable, the target *Version* vertex is considered an affected version.

After the iteration, the affected *Version* vertex is stored with the publishing date and depth. To boost performance, the maximum depth of the call chain is initially set to 10, based on research indicating that the semantics decline after 10 successive calls [195]. Our study later also confirms that there are significantly fewer affected libraries beyond a depth of 9. We verified ALSearch by randomly selecting 1,000

affected library versions and retrieving dependency trees of them using the *mvn deptree* command. If the library did depend on a vulnerable dependency, the library was considered affected. Only 12 (1.20%) libraries were false positives, mainly due to different OS requirements or incomplete data in MCR (discussed in Threat of Validity Section 6.6).

6.3 Empirical Study

To quantitatively assess the prevalence and underlying cause of persistent vulnerabilities in the Maven ecosystem, we conducted an empirical study to answer the following research questions:

- **RQ1**: ***How prevalent are persistent vulnerabilities in the Maven ecosystem?*** The impact of vulnerabilities over time is evaluated regarding the distribution of time spans and counts of affected libraries to demonstrate persistent vulnerabilities.

- **RQ2**: ***What are the causes of persistent vulnerabilities?*** We quantitatively uncovered the underlying factors by categorizing and analyzing 6 cases to identify the primary cause.

Dataset: the primary dataset is the dependency graph in Section 6.2.2.1. To investigate the prevalence of Log4Shell in real-world projects, besides data from MCR, an additional dataset was created by cloning Java repositories on GitHub managed by Maven (with POM files) and filtering out those with fewer than 20 stars to ensure their popularity. As of April 1, 2023, a total of 13,638 repositories were collected, and dependency trees were extracted using the Maven command *mvn deptree*. The dependency trees of 9,220 repositories were successfully extracted.

6.3.1 RQ1: Analysis of Persistent Vulnerabilities

The impact of persistent vulnerabilities on downstream affected libraries is demonstrated by their long-tail prevalence. We used Log4Shell as an example to showcase the metrics we used and then evaluated all vulnerabilities to demonstrate the persistence.

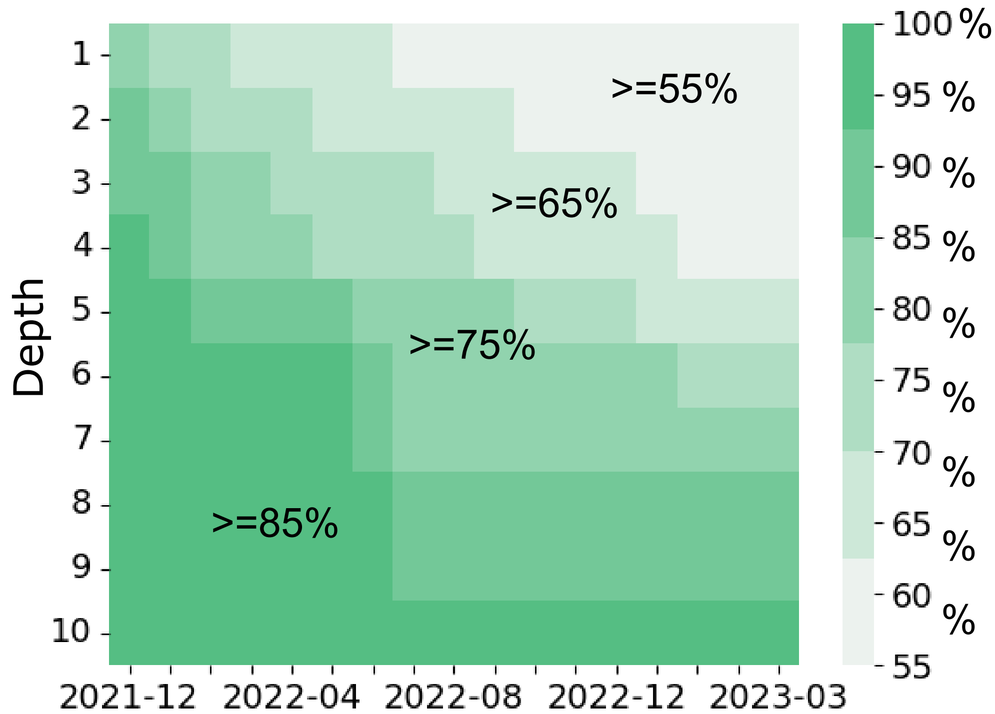


FIGURE 6.1: Heatmap of Proportion of Affected Libraries by Log4Shell

6.3.1.1 Log4Shell Analysis

First, we retrieved the affected library and version vertices associated with release dates with ALSearch. Because usually, the latest version of a library is the release currently maintained by the developers, a library is considered affected if the latest version depends on vulnerable *log4j-core*. The downstream libraries were categorized into 3 categories (1) **Affected**: The downstream library's latest version is affected. The proportion of these libraries is denoted as P_{vul}. (2) **Patched**: The library's latest version is not affected, but at least one of its previous versions was affected. The proportion of them is called P_{patch}. (3) **Removed**: The older versions of the library were affected, and the latest version does not depend on *log4j-core* anymore.

The P_{vul} of Log4Shell over time is demonstrated in the heat map Figure 6.1. In the heatmap, the x-axis refers to the timeline from the publishing date at NVD to 01 Apr 2023 based on months, while the y-axis refers to the depth. It is shown that P_{vul} at depth 1 decays faster than at other depths. It is because those libraries serve

as first-level dependents, which would be quickly aware of the vulnerable versions of *log4j-core* in their dependencies. With the depth increasing, the downstream libraries are less likely to be aware of the transitive vulnerability and less likely to execute the vulnerable code of *log4j-core*. Thus, the decaying rate decreases as the depth goes deep.

In Figure 6.2, P_{vul} and P_{patch} are depicted by days. The sum of P_{vul} and P_{patch} is nearly 100% because the number of the third category, **removed**, is negligible. The P_{vul} reached 50% in Oct 2022 and decayed much slowlier than before. Since P_{vul} decays in a decelerating manner, Log4Shell would remain persistent in the ecosystem without abating for a long time. Hence, we define a metric, **Half-life**, to measure the time that P_{vul} decays to 50% from its initial value. The Half-life of Log4Shell can be measured based on days as 308 days.

Although the P_{vul} decays slowly, the number of newly released affected versions decreases more quickly than P_{vul} as in the same figure at the right axis. The number of new versions gradually decreases from the peak of 361 when Log4Shell was initially exposed. It is seen that there were still new affected versions published after 15 months of exposure. We further investigated the depths of these versions and found out that 94% of them were not first-level dependents. It suggests that the upstream dependencies of these affected libraries failed to upgrade *log4j-core* in time. Note that the number of new vulnerable versions has been fluctuating because the numbers are usually small on weekends.

To assess the prevalence of Log4Shell in real-world Java projects, we searched for vulnerable versions of *log4j-core* in the dependency trees of the $9,220$ Maven projects we collected earlier. Our search revealed that 973 (10.55%) of these repositories had used *log4j-core* in their dependency trees, out of which 392 (40.28%) were using the vulnerable versions of *log4j-core*. We confirmed that none of these repositories had published vulnerable versions to MCR, which indicates that, besides libraries, end users were still using vulnerable *log4j-core* versions in their projects after 15 months of disclosure.

Finding 1: The P_{vul} of *log4j-core* decayed rapidly to 65% in the first 3 months upon disclosure. However, the decaying was decelerating, and it took 308 days to reach its *Half-life*. Log4Shell was still affecting 392 GitHub Maven projects after 15 months.

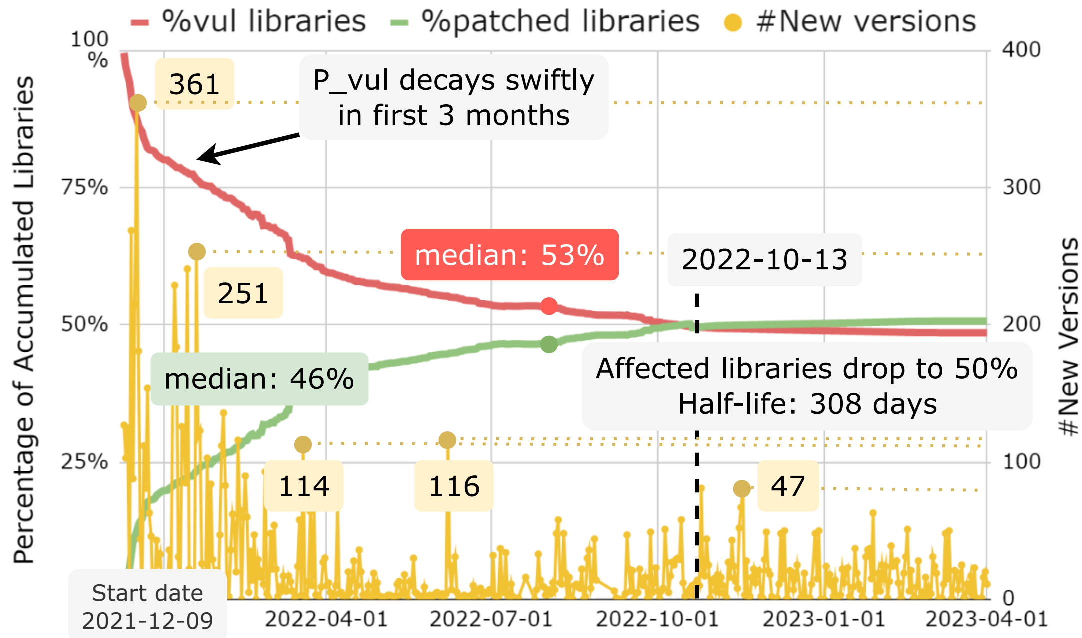


FIGURE 6.2: Accumulated Affected and Patched Libraries for Log4Shell

6.3.1.2 Other Java Vulnerability Analysis

To find out if the decelerating decaying of P_{vul} is prevalent for other vulnerabilities, we measured the P_{vul} for all collected Java vulnerabilities as illustrated in Figure 6.3. Based on P_{vul}, the *Half-lives* of vulnerabilities were derived. Since the exposure duration (from publishing date to data collection date 01 Apr 2023) varies greatly among vulnerabilities, we normalized *Half-life* by dividing the exposure duration. While the **New Release Span** (NRS) was calculated by days from the CVE publishing date to the last date that affected versions are released. *New Release Span* was also normalized by the same exposure duration per vulnerability. Because the number of affected libraries and versions vary greatly among vulnerabilities, the vulnerabilities with exceptionally few affected versions could bring deviations to the distributions. To ensure the representativeness of the vulnerability data set, we filtered out vulnerabilities that affected fewer than 100 versions and plotted the same normalized distributions in Figure 6.3 as *Filtered*. The number of filtered vulnerabilities was $1,319$.

The normalized Half-lives can be negative if the P_{vul} already drops below 50% before the vulnerability is published. Also, the normalized Half-lives can be 100% if the P_{vul} is still above 50% by the data collection date. According to Figure 6.3, only

17.78% of vulnerabilities have their P_{vul} dropped below 50% before the publishing of vulnerabilities, which means the rest 82.22% of vulnerabilities affect 50%+ downstream libraries when they were disclosed. Even by the data collection date, 58.73% of vulnerabilities still maintain over 50% P_{vul}. Hence, it is concluded that most vulnerabilities persist and continue to affect downstream libraries, as seen in the case of Log4Shell. In Figure 6.3, *Filtered half-life*, denoted by light green bars, exhibits the distribution of filtered vulnerabilities. Because the numbers of filtered and pre-filtered vulnerabilities in most intervals are close to each other, it means vulnerabilities that were filtered out did not cause deviations. It is noteworthy that both ends of the distribution are higher than those in between, which means most vulnerabilities either were quickly remediated by downstream libraries or persisted in the ecosystem.

The normalized *New Release Span* is used to indicate the impact of vulnerabilities in new releases. In Figure 6.3, normalized *New Release Span* is depicted by yellow lines. 39% of vulnerabilities still have new affected versions released in the month of data collection (normalized *New Release Span* is 100%), which indicates that nowadays there are still a non-trivial number of downstream developers who fail to upgrade their vulnerable dependencies. Although the distribution of *New Release Span* is dissimilar to the half-life, they both have valley-like shapes, which proves the polarization in vulnerability remediation of the Maven ecosystem. We further measured **Full-life** (the number of days that P_{vul} drops to zero) instead of *Half-life*, and it turned out that only 196 (9.08%) of vulnerabilities have finite *Full-lives*, which means only 9% have all downstream libraries' latest versions fixed regardless of how long time it took.

Finding 2: 82.22% of vulnerabilities affected 50%+ downstream libraries when they were disclosed. The *vul rates* of vulnerabilities have been decaying in a decelerating manner over time, but till our data collection date, 58.73% of them still maintained 50%+ *vul rates*. There are 39% of vulnerabilities that still affect the new versions of downstream libraries that were released in the month of data collection. Only 9% of vulnerabilities terminated their persistence.

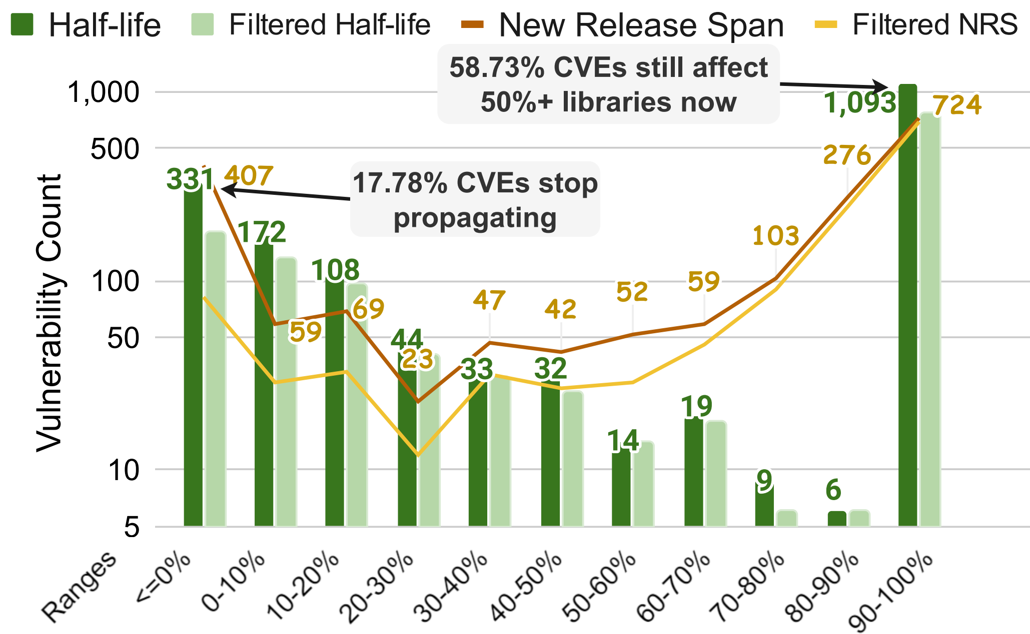


Figure 6.3: Distributions of Normalized Half-lives and New Release Span

6.3.2 RQ2: Study of Underlying Causes

We aim to uncover underlying causes in this RQ. Inspired by the fact that 94% of new versions are transitively affected by other vulnerable downstream libraries, we attempted to investigate the causes based on vulnerability propagation paths.

6.3.2.1 Distribution of the causes

We used a general model that included roles from the source of vulnerability to end users in the vulnerability propagation path as depicted on the left in Figure 6.4. The roles are ***Vulnerable libraries***, ***Medium dependents***, and ***End users***. From RQ1, it is known that the misbehavior of these roles may block the patches from downstream libraries, which leads to persistent vulnerabilities. Hence, we further investigated what kind of misbehavior blocked the patches. To clearly clarify causes without overlapping, the blockage of patches ascribes to the first role that conducts misbehavior during a bottom-up investigation because downstream roles automatically inherit configurations from the upstream.

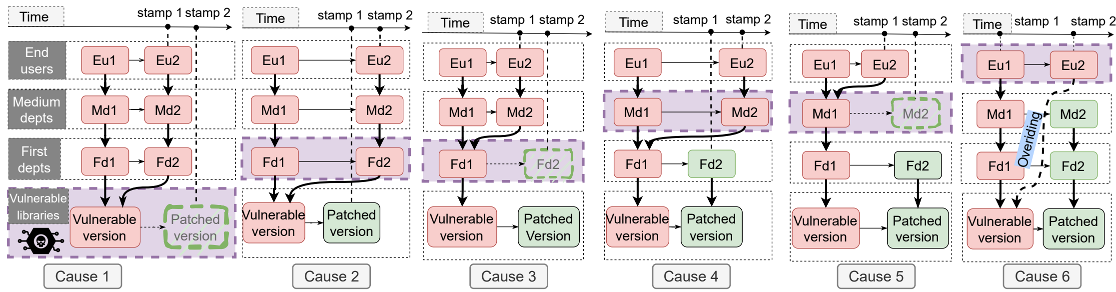


FIGURE 6.4: Scenarios of Different Causes

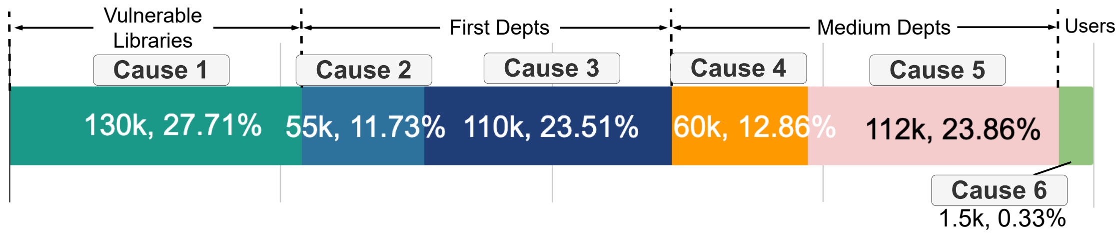


FIGURE 6.5: Proportions of Each Cause

Based on the sequence of release time of two parties on each dependency relationship, we could summarize 6 causes out of three types of dependency relationships among 4 roles as illustrated in Figure 6.4. In the figure, the rectangular box with dashed lines refers to the absent version vertex that is supposed to be present. And the boxes filled with purple color refer to the roles that are to blame for the patch blockage. For example, in the first column, the **Cause 1** is presented: The downstream dependents are affected by the vulnerability because the vulnerable library fails to release the patched version in time. Note that the ***First Depts*** are split apart from ***Medium Depts*** as an independent role because the first-level dependents directly determine the versions of the vulnerable libraries for the other medium dependents and can explicitly select the strategy between version ranges and SoftVers.

In Figure 6.4, **Cause 2** refers to the *First Depts* still using the vulnerable versions even if the patched versions are available. The **Cause 3** refers to the *First Depts* failing to release new versions that depend on the patched versions so that the downstream dependents are forced to use the non-patched versions of *First Depts*. Similarly, **Cause 4** and **Cause 5** refer to the corresponding misbehavior of *Medium Depts*. The last **Cause 6** stands because the versions explicitly overridden by *End users* are still vulnerable versions.

Next, with the causes summarised, the importance of each cause is embodied by its proportion of occurrences. To avoid duplicated counts, we only counted the number of paths where blockage of patches occurred. Figure 6.5 illustrates the proportions of each cause over all valid paths. Firstly, the *Cause 1* is ruled out, because it is caused by the absence of patches instead of the blocked patches. Secondly, it is seen that the *Medium Depts* account for most of the paths, which is 36.72%, and *First Depts* account for a very close number of paths, which is 35.24%. Considering that the misbehavior of any role along the path could lead to the blocked patches, it is remarkable that *First Depts* could affect the similar amount (165k and 172k) of paths as the rest all *Medium Depts* at 2-15 depths, which proves that *First Depts* is the most critical role regarding facilitating the patch adoption than other roles. Finally, although the proportion of *Cause 6* is small, it proves that *End users'* decisions are not always reliable. In fact, much domain knowledge and manual efforts are required for *End users* to select the best version against all vulnerabilities.

> **Finding 3:** It is concluded that misbehavior by *First Depts* (35.24%) and *Medium Depts* (36.72%) are guilty of the majority of affected paths. The *First Depts* are the most significant role in terms of unblocking patches.

In reality, most developers are only concerned by the vulnerabilities in their direct dependencies according to a study [56]. If a *First Dept* uses the vulnerable version, the downstream libraries would automatically inherit the vulnerable version, which means that developers of *First Depts* should be aware of the vulnerability and promptly upgrade vulnerable direct dependencies to patched versions instead of relying on downstream developers. Unfortunately, due to widely used SoftVers (99%) in Maven, the versions specified for vulnerable libraries offer limited flexibility to upgrade against vulnerabilities. It would be unrealistic to force developers in Maven to swerve to version ranges abruptly because SemVer is not properly complied with in Maven and the backward compatibility has to be manually assured. Thus, to avoid reliance on developers, an automatic and scalable way to introduce flexibility to dependency versioning is required to mitigate persistent vulnerabilities.

> **Finding 4:** The root cause of the misbehavior is the widely used SoftVers which greatly limit the flexibility of dependency version selection. Without flexible

version ranges, the downstream libraries and applications are automatically prone to vulnerable dependencies even if patched versions are released.

6.4 Methodology and Evaluation

Because the limited flexibility hinders the spread of patches, we aim to introduce the flexibility to unblock the patches. First, we reviewed the existing solutions to identify the pros and cons, based on which, our solution *Ranger* is proposed and evaluated to answer the following research questions:

- **RQ3**: ***How do existing solutions address persistent vulnerabilities?***
- **RQ4**: ***How effective is Ranger regarding mitigating the persistence of vulnerabilities?***

6.4.1 RQ3: Review of Existing Solutions

The solution recommended by Maven is the semantic version ranges [225]. Unfortunately, considering that SemVer is not properly complied with, instability could be introduced by ranges so that ranges are rarely adopted. Apart from ranges, the most used approach is the transitive version override. If the versions of transitive dependencies are vulnerable, any dependent can override the transitive vulnerable versions by *dependencyManagement*. Hence, as solutions supported by Maven, ranges and version overriding can be used to mitigate the persistence of vulnerabilities. Note that besides Maven, other popular Java Package Managers, Gradle [228] and Ivy [229] implement similar overriding mechanisms, *Dependency Constraints* and *Dependency Overriding* respectively to determine the versions of transitive dependencies if the transitive dependencies exist. Thus, we refer to this overriding mechanism as *dependency version Overriding*. There are also other workarounds, such as tampering with local libraries of vulnerable dependencies to manually backport patches for deployment environments. But temporary workarounds are too infrequently used to be discussed. Furthermore, *exclusion* supported by Maven is not discussed either, because it is used to exclude unused transitive dependencies which are not worth mitigating the vulnerabilities for.

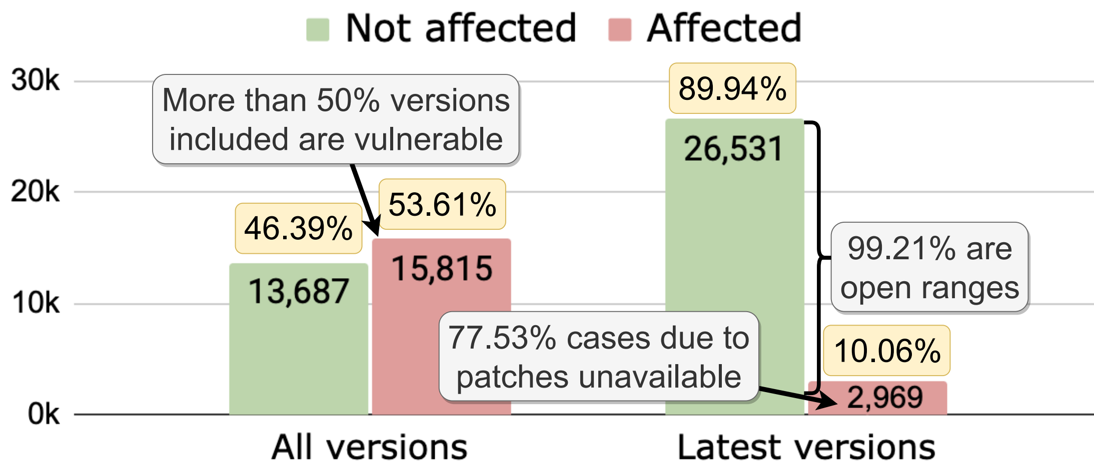


FIGURE 6.6: Usage of Vulnerability Related Version Ranges

6.4.1.1 Study of the Usage of Ranges

From the 82m collected dependency relationships, only 637,783 (1.02%) of them use the semantic version ranges. Out of the range-use dependencies relationships, 29,556 (4.63%) are specified for the vulnerable libraries by *First Depts*. We further investigated these vulnerability-related ranges to reveal how many vulnerabilities can be automatically bypassed.

As illustrated by the left two bars in Figure 6.6, considering all versions within the ranges, 53.61% of versions are vulnerable. However, Maven would usually select the latest (semantically highest) version in a version range as the resolved version of the dependency. Thus, if only the latest versions in these ranges are considered, the proportion of vulnerable latest versions drops to 10.06% in the right 2 bars in Figure 6.6, which proves that version ranges can effectively free dependents from vulnerabilities if patched versions are included. To understand how patched versions were introduced, we went through the ranges whose latest versions are not vulnerable. Out of the 26,531 non-vulnerable versions, 99.21% of them belong to ranges that are actually right open ranges, such as *[1.1,)* without defining the upper bounds. Because the right open ranges would always be resolved to the latest version, the potential breaking changes could be introduced to the dependents whenever any incompatible new versions are released.

Finding 5: Although the version ranges allow flexible upgrades of vulnerable dependencies, they are rarely used in Maven (1.02%). The fact that the latest

versions of 89.94% of version ranges of vulnerable libraries were no longer vulnerable proves the effectiveness of version ranges. However, 99.21% of ranges that successfully bypassed vulnerabilities were open ranges that are subject to unpredictable incompatibility issues. Hence, to properly use version ranges, compatibility has to be assured.

6.4.1.2 Study of the Version Overidding

Because only *Medium Depts* and *End users*, the indirect dependents of vulnerable libraries, would use *dependency version Overriding* to control the versions of transitive dependencies, we study the effectiveness of *dependency version Overriding* for them. In total, there are 639, 710 (6.50%) POM files for versions that use *dependencyManagement*, from which we extracted the overridden versions per POM file. After matching the overridden versions with vulnerability mappings, we found 295, 951 POM files have overridden versions of vulnerable libraries. Then, these files were categorized into 2 cases in Table 6.1 regarding whether they bypassed the vulnerabilities: (1) *Affected*: Any overridden version in the POM file was still vulnerable. (2) *Bypass*: The default version was vulnerable but the overridden was not. Note that there was overlapping because a POM file may have multiple overridden versions.

It turned out 86% of POMs bypassed the vulnerabilities in transitive dependencies because probably their developers were aware of the vulnerabilities and explicitly addressed them with *dependencyManagement*. However, it is surprising that 72% (214, 933) POM files both bypassed some CVEs and introduced other CVEs at the same time. Only 14% of POMs completely bypassed all vulnerabilities. It implies that fixing vulnerabilities with version overriding is a non-trivial job, which is the first weakness of version overriding, **Knowledge and efforts**. The developers must equip with extensive domain knowledge of vulnerabilities and invest efforts to ensure their eradication. Although version overriding is able to address vulnerabilities for the current project within a POM file, it is not inheritable according to Maven Specification [220] so that it does not benefit the downstream libraries. Another weakness of version overriding is **Non-inheritability**, because of which, Since the version overriding only works for current projects instead of dependents that depend on the projects, the vulnerable versions are still being used by downstream libraries unless all developers along the propagation path conduct the

Table 6.1: Counts of POMs with *dependencyManagement*

With vul libraries	Affected by CVEs	Bypass CVEs	Overlapping
295,951	254,043 (86%)	256,841 (87%)	214,933 (72%)

same overriding. Therefore, the patch versions cannot be automatically adopted by downstream users. In conclusion, the version overriding can only serve as a temporary workaround instead of boosting the self-healing of the ecosystem.

Finding 6: The adoption rate of dependency version overriding is 6.50% and only 14% of adopters completely bypassed all vulnerabilities. Because dependency version overriding requires knowledge and manual effort and is unable to benefit downstream users due to non-inheritability, it is not effective in eliminating persistent vulnerabilities.

Another solution worth discussing is Plumber [54] which addresses the persistent vulnerabilities in the NPM ecosystem. Plumber employs a dependency graph to identify the dependents that block fixes of vulnerabilities. Subsequently, it endeavors to upgrade the blocking dependents to compatible versions. If upgrading is not possible within the bounds of compatibility, Plumber generates remediation suggestions, such as backporting and migration, both of which require manual intervention. However, Plumber is not applicable to Maven, because it relies on compatible ranges that are pre-specified by developers, a feature that is prevalent in NPM [119] but not in Maven, which further necessitates the compatible version ranges for Maven.

6.4.2 RQ4: Methodology and Evaluation of *Ranger*

6.4.2.1 Requirements of the Solution

Based on the previous research question, existing solutions, such that open version ranges are subject to breaking changes and dependency version overriding is non-inheritable and requires intensive manual efforts. Despite the limited usage, version ranges were proven to be effective for unblocking the patches. However, due to legacy reasons, developers predominantly utilize SoftVers, making it impractical to mandate a shift toward version ranges, not to mention that version ranges have

to be manually curated by developers. Therefore, our objective is to propose an automated solution for restoring version ranges of both vulnerable libraries and dependencies that transitively depend on vulnerable libraries from SoftVers. By restoring the version ranges, vulnerability fixes within the ranges can propagate smoothly and automatically to downstream users. Moreover, for the purpose of ecosystem-level implementation, *Ranger* should possess the ability to continuously monitor the Maven ecosystem for blocking dependents and promptly provide the restored version ranges along with corresponding suggestions to the developers of such blocking dependents. This approach would expedite the propagation of patches throughout the ecosystem.

6.4.2.2 Design of *Ranger*

To this end, we have proposed *Ranger*, which comprises a server-side edition and a client plug-in. The client plug-in for *Ranger* can be integrated into a developer's workflow as a Maven plug-in. For a Maven project, this plug-in can automatically replace the SoftVers in the POM file with curated compatible version ranges, and the developer can effortlessly publish the updated POM file with version ranges to benefit downstream users. On the other hand, the server-side edition of *Ranger* employs the ALSearch algorithm to continuously monitor an up-to-date dependency graph for instances of vulnerability fix blockage caused by SoftVers. When a blockage is detected, *Ranger* calculates compatible version ranges for the vulnerable constraints and reports this suggestion of version ranges to the relevant developers.

As depicted in Figure 6.7, we first introduce the plug-in that accepts a dependency with SoftVer and class files of the project as input. Given that version ranges specified by developers typically consider compatibility and functionality, *Ranger* aims to ensure them for the restored version ranges. Specifically, given a SoftVer v_s, *Ranger* retrieves sorted candidate versions $V_{cand} = \{v_1, v_2, ..., v_n\}$ from the MCR as a list as well as the version and vulnerability mappings from the dependency graph. Then *Ranger* determines which versions from V_{cand} should be included in the restored range V_r to ensure V_r is more secure, flexible, and compatible. We further formulate the problem into a Multi-Objective Optimization problem:

- **Objective 1** (Primary): The maximum number of vulnerabilities for all versions in V_r is minimized to guarantee any version resolved by Maven is more secure than v_s.
- **Objective 2** (Secondary): V_r should include as many candidate v as possible for better flexibility.
- **Constraint 1**: V_r must be compatible with v_s.
- **Constraint 2**: any v_r in V_r must has not greater vulnerabilities than v_s to ensure the effectiveness of restoration.

$$\begin{aligned} min \quad & f_1 = max(\sum_{n=1}^{dt(v_r)} count_{vul}(n)) \\ max \quad & f_2 = |V_r| \\ s.t. \quad & c_1 : compatibility(v_s, v_1) = 1 | \forall v_r \in V_r \\ & c_2 : \sum_{n=1}^{dt(v_r)} count_{vul}(v_r) \leq \sum_{n=1}^{dt(v_s)} count_{vul}(v_s) | \forall v_r \in V_r \end{aligned}$$

where $dt(v) = \{n_1, n_2, ..., n_t\}$ is to resolve a dependency tree from the version v. the total vulnerabilities of v are the sum of numbers of vulnerabilities associated with each node in $dt(v)$ to include transitive vulnerabilities.

We implemented Algorithm 3 in *Ranger* to solve the problem above. From L1-L8, *Ranger* first queries the number of vulnerabilities for the resolved dependency tree of SoftVer v_s and each version in V_{cand}. To adhere to constraint c_2, the versions with more vulnerabilities than v_s are filtered out. Then from L9-L11, *Ranger* sort the filtered versions in a SemVer order and split the list of versions into the upper and lower parts by v_s to add potential versions bi-directionally for more candidates. For both the upper and lower parts, *Ranger* checks the compatibility between candidates and v_s. Note that the compatibility checkers employed by *Ranger* can handle all types of code-based compatibility as specified in the Oracle documentation [230], including Source, Binary, and Behavioral Compatibility. The Source and Binary Compatibility are ensured by two commonly used tools, revapi and jcp [36, 37] with high accuracy. For Behavioral Compatibility, we used the only static detector Sembid [1]. Specifically, *Ranger* calculates incompatible APIs with the checkers and compares them with the reachable APIs collected from the

Algorithm 3: Algorithm of *Ranger*

Input: SoftVer v_s, candidate versions V_{cand}, class files f

Output: Restored version range V_r

```
1  V' ← set(v_s)
2  dt_{v_s} ← dependencyTree(v_s)
3  vul_{v_s} ← queryCVE(dt_{v_s})
4  foreach v_cand in V_cand do
5      dt ← dependencyTree(v_cand)
6      vul ← queryCVE(dt)
7      if & vul ≤ vul_{v_s} then
8          V' ← v_cand
9      sort(V')
10     V_upper ← V'.truncate(v_s, v_last)
11     V_lower ← V'.truncate(v_first, v_{v_s})
12     foreach v in V_upper do
13         api = compatibilityCheck(v_s, v)
14         if ¬reachable(f, api) then
15             V_r ← v
16     foreach v in reverse(V_lower) do
17         api = compatibilityCheck(v_s, v)
18         if ¬reachable(f, api) then
19             V_r ← v
20 foreach v_r in V_r do
21     if queryCVE(dt(v_r)) > min(Vul(V_s)) then
22         V_r remove v_r
23 foreach v_r in V_r do
24     if ¬unitTest(v_r) then
25         V_r remove v_r
26 return V_r
```

call graphs. The call graphs are constructed with Soot Spark [231] from the class files of the project and byte code of the dependency to determine whether any incompatible API is reachable. If a candidate version has no reachable incompatible APIs, it is included in the range. In L13, compatibility checkers serve as a pre-filter for the final validation because they are static and more efficient than testing. In L20-L22, *Ranger* excludes versions that have more vulnerabilities than the minimum required in order to satisfy Objective f_1. In L23-25, given the heavier resource demands of unit tests, *Ranger* further excludes versions failing the unit test serving as the final validation.

Regarding the server-side edition of *Ranger*, the initial step involves identifying the

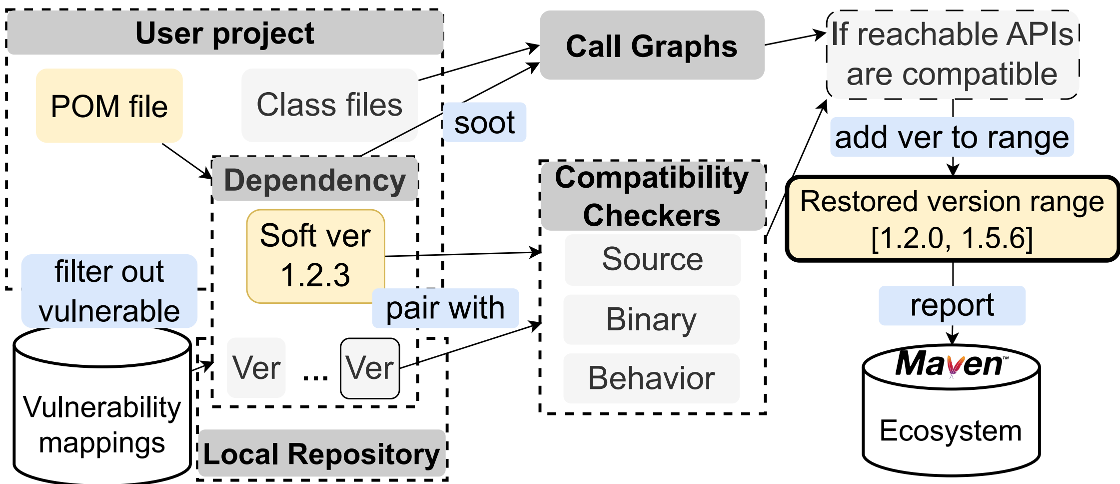


FIGURE 6.7: Overview of *Ranger*

blocking dependents, denoted as *First Depts*, by means of the ALSearch algorithm. The plug-in running on the server proceeds to calculate and test the compatible version ranges using the repositories stored in our database. If a version range covering the patched version is successfully restored, *Ranger* generates a report that is sent to the relevant developer. In cases where range restoration fails, *Ranger* attempts to locate the *Second Dept* of the failed *First Dept* from the dependency graph and calculates the restorable range towards the *First Dept* instead of the vulnerable library. This is because *First Dept* is the direct dependency of *Second Dept* and only specified versions of direct dependencies are transitive for the rest of the dependents. This process is repeated 10 times until no range can be restored.

6.4.2.3 Evaluation of *Ranger*

To showcase the effectiveness of *Ranger* in real-world scenarios, we initially evaluated the plug-in on a dataset of 252 GitHub repositories that included vulnerable versions of *log4j-core* in their dependency trees, as of 01 Apr 2023. Subsequently, we conducted a large-scale evaluation on another dataset to demonstrate the effectiveness of *Ranger* for mitigating persistent vulnerabilities in the Maven ecosystem.

• **Evaluation of Plug-in: Dataset:** From the 9,220 repositories in Section 6.3, we retrieved the dependency by Maven command and check if any vulnerable log4j version was still in use. 374 repositories were derived. Only 252 of them could be successfully compiled and tested. **Results:** We ran *Ranger* to only restore the version ranges for *log4j-core* in 252 parent POM files to control the variables. 160

TABLE 6.2: Results of *Ranger*

	Restored	Failed Unit Tests	Restore Rate	Recall/Precision
Ground Truth	171	N.A.	67.85%	N.A.
Ranger	160	19	63.49%	93.57%/100.00%

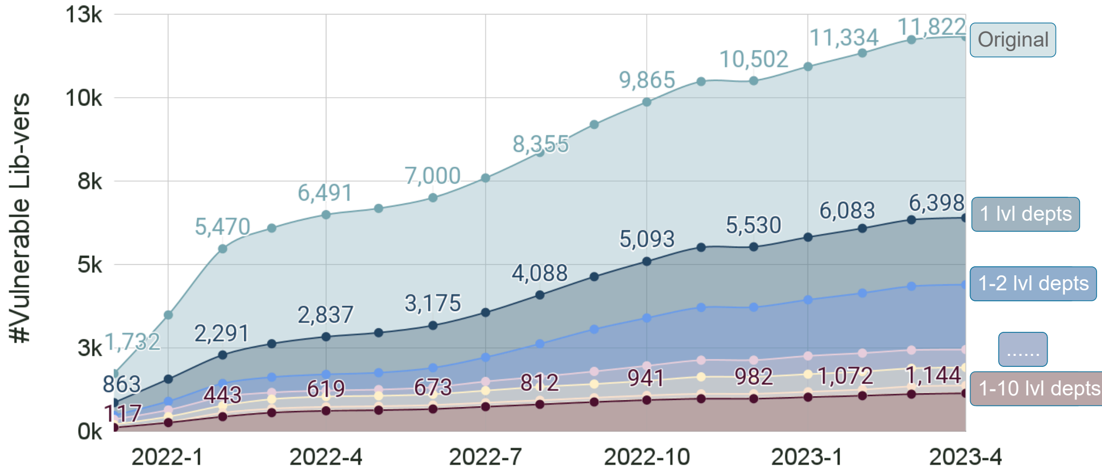


FIGURE 6.8: Number of Vulnerable Lib-vers over Months after Applying *Ranger* to Dependents at 1-10 Depths

secure ranges of them were restored with a 63.49% restoration rate. Before running the compilation and unit tests, 179 raw ranges were statically calculated, which means unit tests could reduce 19 false positives. It should be noted that false negative cases were present in our evaluation, as false alerts produced by static compatibility checkers exclude the potential candidate versions, and unit tests only narrow the range but not widen it. Thus we manually checked the failed cases and found that 11 could have been restored. The results were summarized in Table 6.2 with the actually restorable repositories accounting for 67.85%. Although *Ranger* had some false negatives, it hardly introduced false positives that could break current and downstream projects. It was proven that 93.57% of restorable version ranges could be automatically restored by *Ranger*.

• **Evaluation of Server-side Edition: Dataset:** To simulate a scenario where developers of downstream dependents adopt the version ranges generated by *Ranger* upon the disclosure of a vulnerability, we compiled a list of all affected libraries and versions published after the disclosure of Log4Shell. This resulted in a total of 11,822 library-version pairs, denoted as lib-vers. **Results:** Regarding the lib-vers, *Ranger* first restored ranges for the *First Depts* at Depth 1. This resulted in a successful restoration of 486 out of 668 dependent lib-vers. The generated ranges

were then applied to the dependency graph, and we performed ALSearch again to retrieve the affected *Second Depts* at Depth 2 on an updated graph. Out of 927 *Second Depts*, 731 were successfully restored. We repeated this process for a total of 10 depths and evaluated the number of vulnerable lib-vers over time, as shown in Figure 6.8.

In total, it took 4,110 iterations to successfully restore 3,109 version ranges with a 75.64% restoration rate. As a result, 90.32% of the vulnerable lib-vers were successfully remediated from Log4Shell, leaving only 1,144. It is clear that the number of vulnerable lib-vers increases much more slowly over time after applying *Ranger* to the 10th dependents than the primitive state. This suggests that the propagation of the vulnerability was effectively suppressed from the beginning upon disclosure of Log4Shell. Moreover, it is observed that the number drops 45.95% when *Ranger* is only applied to the *First depts* at Depth 1, which indicates that *First depts* have a significant impact on downstream libraries yet not enough to suppress the propagation. Also, the marginal effect of *Ranger* drops fast as depth goes deep in the Figure and there were only 8 ranges to restore at Depth 9 and 10, which means Depth 10 is effective enough against persistent Log4Shell.

However, there still remained 1,144 unfixed lib-vers requiring manual intervention. We categorized the remaining cases into three: (1) **No compatible patched versions to upgrade** (481 cases, 42%): *Ranger* found there was no version satisfying the constraints. For these cases, *Ranger* generated a report with breaking APIs and call chains with suggestions of manual fixes for developers to resolve the incompatibility. (2) **No secure versions available** (592 cases, 52%): This mostly happens for dependents at Depth 2+ because their direct dependencies may not have published a secure version that transitively depends on a patched version of *log4j-core*. It is a common case, especially for a newly disclosed vulnerability, for which, *Ranger* would suggest the developers find a substitution if the vulnerable library is reachable. If not reachable, a suggestion to exclude the vulnerable library would be suggested. On the other hand, *Ranger* will continue to monitor the availability of patched versions. (3) **Internal error** (71 cases, 6%): These were caused by issues irrelevant to the design of *Ranger*, namely, failed jar downloading, failed call graph generation, and the errors of compatibility checkers.

Finding 7: Our evaluation demonstrated that *Ranger*, as a plug-in, was successful in restoring secure version ranges for 63.49% of the 252 real-world GitHub repositories, with a high recall of 93.57%. In a simulated experiment, the server-side edition of *Ranger* was able to restore version ranges for 3,109 (75.64%) of the dependents which successfully remediated 10,678 (90.32%) of downstream vulnerable projects.

6.5 Discussion

• **Compatibility check should be aligned with SemVer, especially in Maven.** Many studies [1, 27, 130–132] have revealed that SemVer has not been well adhered to by developers in the Maven ecosystem, leading to prevalent SoftVer. Although *Ranger* could restore version ranges to include patched versions, the patched versions must be those already published. To timely apply the patches upon releases, version ranges have to be open ranges or semi-open ranges, e.g. caret range ˆ1.2.3 [232], which requires strict compliance with SemVer to assure compatibility. Therefore, in the long run, *Ranger* could mitigate the persistent vulnerabilities, but only the widely used and strictly backward compatible open version ranges could nip them in the bud.

• **More efforts and resources should be leaned on widely used but poorly maintained libraries.** As revealed by our evaluation in Section 6.4.2.3, there were 592 cases without patched versions to upgrade to. Following a manual investigation, it was discovered that several libraries served as dependencies in a large number of projects or libraries, but their maintenance was inadequate. To address persistent vulnerabilities, it is imperative that the maintainers explicitly release patched versions for the benefit of downstream users. Therefore, this kind of libraries should arouse the collective awareness of the community, and the resources of open-source software governance should be directed towards these widely-used but poorly-maintained libraries to promote a more secure ecosystem.

• **Differences between *Coral* and *Ranger*** The differences lie in 3 perspectives: (1) The scenarios of usage: *Coral* is used to remediate the vulnerabilities within individual user projects, while *Ranger* is only used to restore version ranges from the single versions (Soft version) without considering the vulnerabilities to increase

the flexibility of version selection by Maven. (2) The target subjects: *Coral* works on the dependency graph and bytecode of user projects, and *Ranger* only works on the single versions specified for individual dependencies, and bytecode of user projects. (3) The expected output: *Coral* returns the updated versions of all dependencies within the dependency graph, but *Ranger* only returns a version range with multiple versions for a single version to include more compatible versions in the range. Therefore, although they both employ similar compatibility checking tools, they serve for totally different purposes (*Coral* for vulnerability remediation and *Ranger* for increasing version selection flexibility).

• **Synergy between *Coral* and *Ranger*** *Coral* and *Ranger* are complementary to each other. The *Coral* is a tool that can be used to identify the vulnerable libraries and their dependents, while *Ranger* is a tool that can be used to restore the version ranges for the vulnerable libraries and their dependents. While designing *Coral*, a compromise was made to use the compatibility checking tools and secondary objectives, such as minium major upgrades, to minimize the risks of incompatibility, as version ranges are not predominantly used in the Maven ecosystem like other modern one, such as NPM [16] and Pypi [233]. However, *Ranger* can restore the version ranges contextually for dependencies which can be used by *Coral* to accurately determine the compatible ranges for the vulnerable libraries and enlarge the flexibility of version selection. Therefore, despite the different perspectives, the synergy between *Coral* and *Ranger* can be used to effectively maintain the security of the Maven projects and ecosystem.

6.6 Threats of Validity

The primary threat of the study is the assumption that dependents of vulnerable libraries were considered affected without fine-grained reachability or triggerability analysis. Because analyzing the reachability of all vulnerabilities in the entire Maven ecosystem at a large scale is quite expensive, we did not take it into consideration. Furthermore, vulnerable libraries are also packaged into the deployment environment, and having vulnerability is not a secure practice because they could be exploited someday given the evolving source code. Hence, to promote the best security practice in the Maven ecosystem, we made such an over-assumption.

Another threat is the assumption that successful compilation and passing unit tests after applying version ranges generated by *Ranger* are sufficient to confirm successful version range restoration. However, in real-world software development, unit tests have limited coverage, and passing them does not necessarily guarantee that the restored version ranges satisfy all requirements of developers. Despite this limitation, unit tests are a critical component of deployment and are currently the most convenient validation approach available.

The last threat is the accuracy of the algorithm ALSearch that is used to track the downstream libraries. The first factor affecting the accuracy is the dependency graph sourced from MCR, and a few POM files in MCR could be unavailable leading to incomplete dependency edges in the graph. Another factor is that the environment requirements of dependencies were ignored, which could lead to false positives because some dependencies are only installed in certain environments, such as Windows. However, these factors were proven to be corner cases in the validation experiment in Section 6.2.2.2 so that the overall conclusions are not undermined.

6.7 Conclusion

In order to find a solution that addresses ecosystem-wide persistent vulnerabilities, we conducted an empirical study that revealed that 58.73% of vulnerabilities still impacted more than 50% of downstream libraries in the Maven ecosystem nowadays. Through this study, we quantitatively substantiated that blocked patches caused the persistence of vulnerabilities. The existing solutions are either not scalable or subject to breaking changes. Hence, we proposed *Ranger* as a scalable and automatic approach with compatibility assurance to unblock the vulnerability patches. Through evaluation, *Ranger* achieved 93.57% recall and restored 3,109 (75.64%) ranges, which remediated 10,678 (90.32%) vulnerable downstream projects.

Chapter 7

Golang Vulnerability Life Cycle Analysis

7.1 Overview

In this chapter, we investigate the life cycle of vulnerabilities in the Golang ecosystem. First, we aim to understand the prevalence and impact of lags in the fixing process of vulnerabilities. Next, we focus on the lag between the fixing commit and the subsequent version release, the lag between the version release and the indexing time, and the lag between the fixing commit and the dependent fixing time. Finally, we explore the factors that influence the fixing lags and propose strategies to mitigate the lags.

7.2 Background and Motivating Example

7.2.1 Background

For the clarity and conciseness of the following sections, the following key concepts used throughout this chapter are explained:

Module: A collection of packages [234] that are released and versioned together, similar to the concept *library* in other ecosystems.

Version: A version identifies an immutable snapshot of a module. A version tagged by the maintainer usually starts with the letter *v*, followed by a semantic version [17].

Pseudo-version: A pseudo-version [235] [1] is a uniformly formatted pre-release version based on a commit in a version control repository, such as GitHub. A pseudo-version comprises three parts: (1) A base version prefix (*vX.0.0* or *vX.Y.Z-0*), which is either derived from a semantic version that precedes the revision; (2) A UTC timestamp of the commit time; (3) A commit hash.

Index: When a module is declared as a dependency by any user, it is registered in the Go Module Index. Using `module@latest` retrieves the latest version from the Go Module Index. When no version is available, a pseudo-version is retrieved.

Fixing Commit: A commit involving the code changes that address a vulnerability. The commit could be incorporated into the subsequent version releases. Note that, in a repository, the fixing code changes could be accommodated into multiple commits for multiple pipelines of releases which forms a complex commit graph. It was challenging to calculate the non-vulnerable ranges of commits and versions on a complex commit graph.

Patch Version: It refers to the first subsequent stable non-pre-release version that incorporates the fixing commit following the vulnerable versions.

File go.mod & go.sum: *go.mod* is a Golang project's dependency declaration file, listing direct and indirect modules with their versions. *go.sum* is automatically generated during project building, recording dependencies and their checksums. *go.sum* is vital for secure and reliable dependency management.

7.2.2 Motivating Example

Since versions could only be automatically pushed to users when they are officially indexed by Go Module, the time interval between the release and indexing could cause unnecessary lags. If the new version includes patches for vulnerabilities, the lag could result in a window period for attackers. For example, a widely-used module *go.etcd.io/etcd* (*etcd*), which owns 43.0*k* stars on GitHub, is known to have

[1] For example, v0.0.0-20191109021931-daa7c04131f5 is a pseudo-version.

a vulnerability, *CVE-2020-15113*. The patched version, *v3.3.23*, was released on July 17, 2020, in response to this vulnerability. For downstream users, obtaining the patch version *v3.3.23* through the Go Module client is not possible if the version is not indexed by Go Module beforehand. Due to the absence of a real-time registry for Go Module, the explicit specification requires developers to manually search *v3.3.23* in the *etcd* repository. Worse, since *v3.3.23* version was quickly superseded by *v3.3.24* on August 19, 2020,*v3.3.23* was not even used by any dependent. After indexing patch versions after one month, 109 dependents performed the fixing by either upgrading *etcd* or migrating to other modules. To understand to what extent the lag affects vulnerability fixing and how to mitigate the lag, we conducted this study.

7.3 Methodology

We first constructed an infrastructure for data analysis given multiple data sources. Then, we derived four critical periods and two types of lag in the life cycle of a vulnerability with algorithm 4.

7.3.1 Analytical Infrastructure Construction

As illustrated in Figure 7.1, the left part demonstrates the structure of our analytical infrastructure. This research investigates the life cycles of vulnerabilities, starting with the acquisition of vulnerability data. Relevant references from public vulnerability databases were examined to identify the fixing commits in the repositories of the affected modules. Also, the subsequent patched versions were obtained from the commit history. The fixing commits and versions were used to initiate the lag analysis process. Additionally, the indexing time of versions was collected, given the unique indexing mechanism of Go Module. To gain insight into the fixing operations of downstream dependents, the commit history of their repositories was further analyzed to obtain the dependent fixing time.

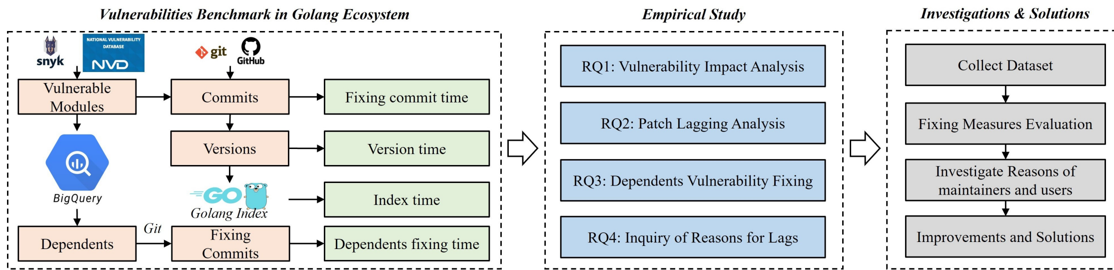


FIGURE 7.1: Overview of the Empirical Study

7.3.1.1 Vulnerability

We collected 1,837 Golang vulnerabilities from two databases: Snyk Advisory [236] and NVD [237]. To locate the fixing commits in the vulnerable modules, we manually scrutinized the reference links and successfully derived 1,269 vulnerabilities with fixing commits, involving 441 unique modules.

7.3.1.2 Dependency Relation

Based on the vulnerable modules, we acquired the dependents that directly and indirectly use vulnerable modules as dependencies from Open Source Insight [41]. For each dependency relationship, the module name and module versions of both dependencies and dependents were kept. The study identified dependent module names belonging to vulnerable modules.

7.3.1.3 Indexing time of version

The Go Module Index [63] registers the timestamp and dependency relationship of a newly released version when the version is used by dependents for the first time. These timestamps refer to the moment when the new versions are publicly known to Golang developers. The crawled records include timestamps, module names, and versions. However, versions that are not indexed were not recorded.

7.3.1.4 Commit history

We collected the commit history for both vulnerable modules and dependents, including the commit ids, commit subjects and messages, associated versions, commit

time, and the relationships among commits. Atop the commit history, the fixing lag analysis will be conducted.

7.3.2 Fixing Lag Analysis

In this subsection, we aim to precisely identify the time lag of vulnerability fixing. As illustrated in Figure 7.2, there are three major timestamps, fixing commit time T_{fix}, version release time T_{ver}, and indexing time T_{index}. T_{fix} refers to the time when fixing commits are merged or committed to the main branch. For a vulnerable module, the duration between T_{fix} and T_{ver} (following version release time), is denoted as the Lead Time LT_{ver}. Although LT_{ver} usually accommodates the time for code review and testing, it would still cause risky lags for downstream libraries if it is unusually long, we name such lags as version lag, Lag_{ver}. Moreover, we denote the first time of patch versions indexed in Go Module Index as the indexing time (T_{index}), and the time from T_{ver} to T_{index} is denoted as the index lag Lag_{index}. From the dependent side, the time when the vulnerable dependency is addressed is denoted as T_{dept}.

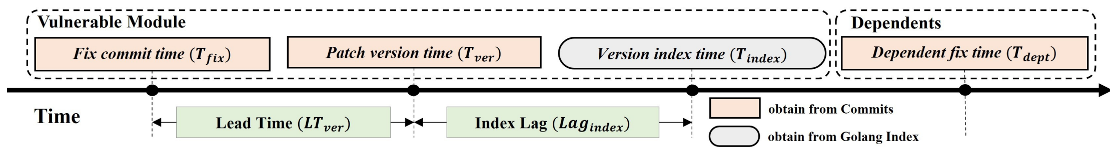


FIGURE 7.2: Fixing Lag

7.3.2.1 Lead Time

Lead Time is the time interval between T_{fix} and T_{ver}. To obtain T_{fix} and T_{ver}, we first downloaded repositories of vulnerable modules and used git commands to obtain the commit trees. Then, For each vulnerability, from reference links, fixing commits are obtained. Because the code changes that fix the commits could be accommodated in other commits, we searched for similar commits from the vulnerable module's commit tree based on the subjects of fixing commits in Alg 4 L2-L4. Among these commits, the time of the earliest fixing commit is considered as T_{fix}. As fixing commits could be committed on non-release branches for development instead of direct use, we further searched the subsequent merging commit along the commit chain. If a merging commit is spotted, T_{fix} is updated to the time of

it as in L8. For each fixing commit, we matched the commit id with the versions' commit ids to locate the timestamps of the first subsequent version after the fixing commit as in L9-L14. The timestamp of the first version is denoted as T_{ver}. Finally, we calculated the LT_{ver} for each vulnerability.

7.3.2.2 Version Lag

Since unusually long LT_{ver} could easily postpone the tagging of patch versions, we further empirically measured such *version lag*, i.e., Lag_{ver}, based on LT_{ver}. We narrow down vulnerabilities by two prerequisites: ① $LT_{ver} > 1$ week; ② current release cycle is greater than the normal version release cycle. For these vulnerabilities, Lag_{ver} is defined as the exceeding time of the current cycle over the normal release cycle in L23. Specifically, the process retrieves the time interval between each pair of adjacent tags on the commit tree. Based on this, The quartile formula [238] is then utilized to ascertain the upper bound of the normal length of version release cycle ranges by $Q3 + 1.5 * (Q3 - Q1)$, as the filter of calculating Lag_{ver}.

7.3.2.3 Index Lag

Index Lag refers to the time interval between the earliest version release time T_{ver} and the earliest index time of versions in Go Module Index T_{index}. Besides T_{ver}, we determined T_{index} by searching for the index time of versions at Go Module Index [239] by module names and versions. The earliest timestamp is denoted as T_{index}. Then, Lag_{index} could be calculated as in L25.

7.3.2.4 Dependent Fix Time

Dependent Fix Time (T_{dept}) is the time of the first dependent that fixes the vulnerability in its dependencies. To measure T_{dept}, we first identified all the dependents that used the vulnerable versions of vulnerable modules. For each dependent, we retrieved the git commit history of the *go.sum* lock file, which specifies dependencies and their versions. Our goal was to locate the commit where the vulnerable version was excluded (Algorithm 4 L26-L32). To identify the fixing commit for

vulnerabilities within a project's dependency tree, we traversed the modification history of *go.sum* for each dependent. Starting from the latest commit that altered the *go.sum*, we exhaustively searched for the existence of the vulnerable version. If not found, we move to the previous commit that altered *go.sum*, continuing until no such commit was found. Once the vulnerable version was located, the following commit in which the vulnerable versions are excluded was considered as the dependent's fixing commit, and its timestamp was denote as the T_{dept} for the dependent. Note that the fix commit may have removed the vulnerable dependency or upgraded the vulnerable versions to clean versions. Based on Algorithm 4, we can calculate the two types of lags, Lag_{ver} and Lag_{index}, for vulnerable dependents, as well as T_{dept} for each dependent.

7.4 Empirical Study

Considering the distinctive package management of Golang (i.e., decentralized registries and Go Module Index), the propagation of vulnerability patches could be lagging. In this case, we carry out an ecosystem-wide empirical study to investigate the vulnerability impact and the propagation of their corresponding patches in this section. Specifically, we unveil the vulnerability impact in Golang by answering the following research questions:

- **RQ1: Vulnerability Impact Analysis.** To what extent could TPL vulnerabilities affect modules in the Golang ecosystem?
- **RQ2: Patch Lagging Analysis.** How long does it take to release and index the patch versions?
- **RQ3: Dependents Vulnerability Fixing.** What are the fixing lags by dependents, and quantitatively what factors could facilitate the fixing?
- **RQ4: Inquiry of Reasons for Lags.** What are the reasons for the patch lagging regarding both module maintainers and users?

First, RQ1 measures to what extent the Golang ecosystem could be influenced by TPL vulnerabilities. Based on this, we analyze the reaction of maintainers of both

```
Algorithm 4: Calculation of Lags
Input: dep_commits: commits of a vulnerable dependency, dept_commits: commits
       of a dependent
Output: Tag_Lags, Index_Lags
1  fix_commit ← fromNVD(vul)
2  foreach commit ∈ dep_commits do
3      if commit.subject == fix_commits.subject then
4          fix_commits.add(commit)
5  commit_stack ← fix_commits
6  T_fix = MIN(fix_commits.release_time)
7  if subsequenct(T_fix) is merge then
8      T_fix = subsequenct(T_fix)
9  commit ← commit_stack.pop
10 while commit ≠ φ do
11     non_vul_commits ← non_vul_commits + commit
12     foreach son_commit ∈ commit.son_commits do
13         commit_stack ← commit_stack + son_commit
14     commit ← commit_stack.pop
15 foreach commit ∈ non_vul_commits do
16     if isVersion(commit) is True then
17         non_vul_versions.add(commit)
18 normalCycle = MAX(quartiles(tagIntervals))
19 T_ver = MIN(non_vul_versions.release_time)
20 currentCycle = T_previous_ver − T_ver
21 LT_ver = T_ver − T_fix
22 if LT_ver > 1week&&currentCycle > normalCycle then
23     Lag_ver = currentCycle − normalCycle
24 T_index = fromIndex(non_vul_version)
25 Lag_index = T_index − T_ver
26 com_stack′ ← depts_commits
27 while com_stack′ ≠ φ do
28     com ← com_stack′.pop
29     if ! hasVulCommit(vul, com) and hasVulCommit(vul, com_prev) then
30         T_dept = com.release_time
31         Break
32     com_prev = com
33 return LT_ver, Lag_ver, Lag_index, T_dept
```

vulnerable packages and their downstream dependents in RQ2 and RQ3, respectively. After this, we demystify the time gaps between vulnerability exposures and downstream recipients adopting the patch upgrades and find out possible improvements to promote the mitigation of vulnerability impact in downstream dependents in RQ4.

7.4.1 RQ1: Vulnerability Impact Analysis

7.4.1.1 Data Preparation

We first collected $1,837$ Golang vulnerabilities from two mainstream databases, i.e., Snyk Advisory [236] and NVD [237]. To substantiate the representativeness of our vulnerability dataset (1,837), we further compare the vulnerabilities with OSV [20] (1,659) and Github Advisory [21] (1,251) by OCT 15, 2023, the exceeded numbers of Golang vulnerabilities proves the representativeness of our dataset. Subsequently, we utilized the dataset from Open Source Insight [20], which maintained package and dependency data from various sources, to retrieve modules and their dependency relations of the Golang ecosystem. Given the absence of centralized registry in Go Module, only modules that are imported as dependencies would be indexed, we are only able to take them as the entire scope of the Golang ecosystem. As a result, $725,286$ modules and $7,892,152$ versions are captured till May 2023, and $475,638,176$ dependency relations associated with the 1,837 vulnerabilities are identified. Moreover, Comparing to other mainstream platform (i.e., *libraries.io* [240], $472K$ Golang modules by OCT 15, 2023), our dataset ($725k$) is also more complete and representative.

7.4.1.2 Impact of Golang Vulnerabilities

The $1,837$ vulnerabilities pose a notable impact on the Golang ecosystem that they affect $479,411$ modules (66.10% of all modules) and $6,313,404$ versions (80.00% of all versions). By the data collection date, $455,813$ modules and $6,071,096$ versions have still not fixed vulnerabilities, which account for 62.85% and 76.93% of all in the Golang ecosystem.

From the time perspective, the proportion of affected downstream dependents increases over the years as well. In Figure 7.3, the histogram plots the number of dependents still affected by vulnerabilities identified in each year, while the broken line plots the percentage of dependents which had been affected by vulnerabilities in each year. As can be observed, despite a notable increase in the number of dependents of vulnerabilities from 2020, it is anticipated that this trend will continue in 2023, given that the data was collected on May 30th. This trend is supported by the rapidly growing proportion of accumulated dependents.

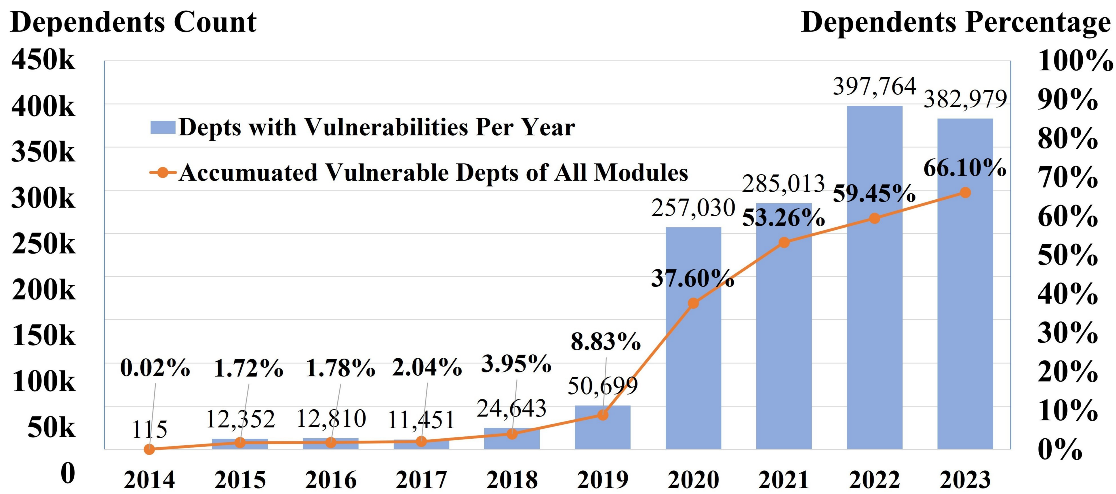


FIGURE 7.3: Distribution of Affected Depts

Finding-1: The vulnerabilities affected a significant number of downstream dependents (479, 411 66.10%) by the data collection date. 62.85% of the dependents have still not fixed the vulnerabilities.

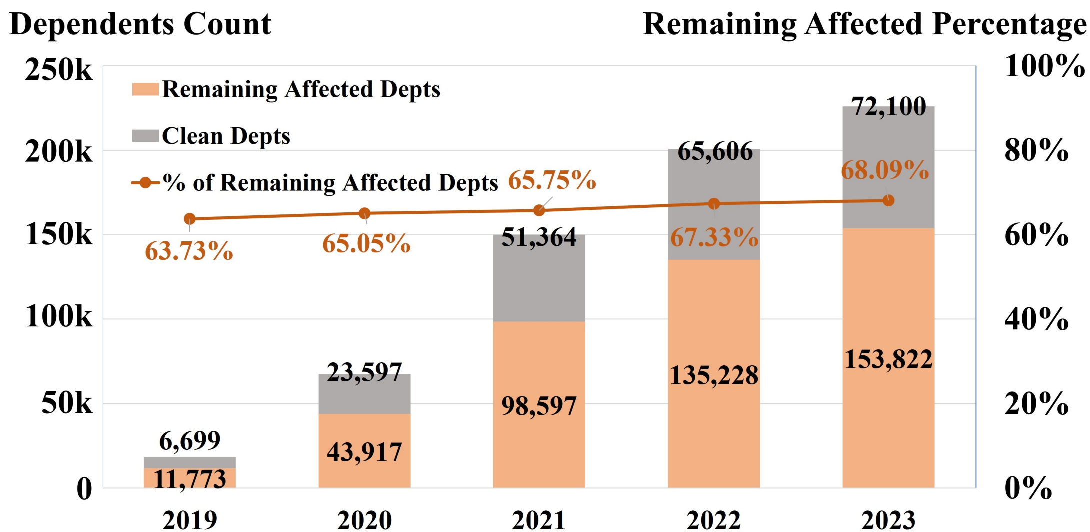


FIGURE 7.4: Impact over Time of Vulnerabilities of 2019 and Before

To demonstrate the impact of vulnerabilities occurring after 2019, we depicted the distribution of affected dependents in Figure 7.4. Specifically, affected dependents modules that have vulnerable dependencies, and clean dependents are on the opposite. It is seen that both affected and clean dependents increased substantially over time. Even if some of the vulnerable dependents got fixed over time, more dependents were developed and involved with the development of the Golang ecosystem. Hence, the proportions of affected dependents maintained at a steady level, but

the number of affected dependents increased remarkably. It is proven that the legacy vulnerabilities have been persistently affecting increasing dependents in the ecosystem.

It is seen that both affected and clean dependents increased substantially over time. Even if some of the vulnerable dependents got fixed over time, more modules that depend on these vulnerable modules were introduced into the Golang ecosystem. Hence, the proportions of affected dependents maintained at a steady level, but the number of affected dependents increased remarkably. It is proven that the legacy vulnerabilities have been persistently affecting increasing dependents in the ecosystem.

Finding-2: Even for vulnerabilities of 2019 and before, the affected dependents have been increasing over the years while keeping the proportion of affected vulnerabilities steady.

7.4.2 RQ2: Patch Lagging Analysis

Due to the significant amount of unresolved vulnerabilities in downstream dependencies over a long period, we further investigated the lags from the perspective of module maintainers. In this section, we employed the timestamps T_{fix} (fixing commit time), T_{ver} (version release time), and T_{index} (Go Module Index publish time) to calculate the lags Lag_{ver} and Lag_{index}.

7.4.2.1 Data Preparation

Besides the vulnerability data from RQ1 Section 7.4.1, to compute the lags between T_{fix} and T_{index}, the fixing commits with timestamps were derived from the associated repositories and the publish time of the patched versions was crawled from Go Module Index [61]. Specifically, we successfully identified the fixing commits for $1,269$ vulnerabilities. Besides the fixing commits, we searched for additional commits that shared the same commit message. These newly identified commits were then treated as supplementary fixing commits. Among these commits, the earliest fixing commit release time was denoted as T_{fix} Considering that the Go Module Index was launched on April 10, 2019, we excluded 96 vulnerabilities that were

fixed before this date, as the calculation of Lag_{index} is not applicable for them after the launching date. After exclusion, we obtained a dataset of $1,014$ vulnerabilities for further analysis. These vulnerabilities were distributed across 387 repositories. Notably, 85.2% of them were assigned CVEs.

7.4.2.2 Analysis of Lag of Version

We observed that modules of 40 vulnerabilities have not released the patch version so that Lag_{ver} could not be calculated. In Section 7.4.4 (RQ4), we will clarify the reasons why no version was released. Among the remaining 974 vulnerabilities that had at least one released patch version, we plotted the distribution of them based on the LT_{ver} (time interval between the fixing commit and the release of the patch version) and Lag_{ver} (Lead Time exceeds one week and the current cycle is greater than the normal cycle) in Figure 7.5. To demonstrate the timeliness of version releases upon fixing commits, we roughly split the vulnerabilities into two segments, the LT_{ver} within one week and beyond one week, as the testing and Continuous Integration checks could take time. We assume versions with $LT_{ver} < 1week$ are timely patch releases for downstream users. Out of the 974 vulnerabilities analyzed, 32.0% of vulnerabilities did not have their patch versions released in a timely manner. It is noteworthy that 11 vulnerabilities release patch versions over 1 year after the fixing commit submission.

Moreover, the distribution of LT_{ver} by time intervals is presented in Figure 7.5. Notably, almost 47.64% of vulnerabilities are fixed and tagged within 1 day, and 16.32% of these vulnerabilities were even fixed and tagged simultaneously, which is pretty in time for downstream users. While there are also 130 vulnerabilities had their patch versions released over 1 month after the fixing commits, i.e., $LT_{ver} > 1month$. Because Golang would not automatically use the fixing commits as versions for dependencies to incorporate patches unless users explicitly declare them, the delayed version releases could result in the lag of fixing within the Golang ecosystem.

Finding-3: 130 vulnerabilities released versions after fixing commits for 1 month, which could impede patch propagation in the Golang Ecosystem.

[2] Only when the time exceeds one week, will the yellow histogram appear, as we have set one of the requirements for Lag_{ver} to be the LT_{ver} greater than one week.

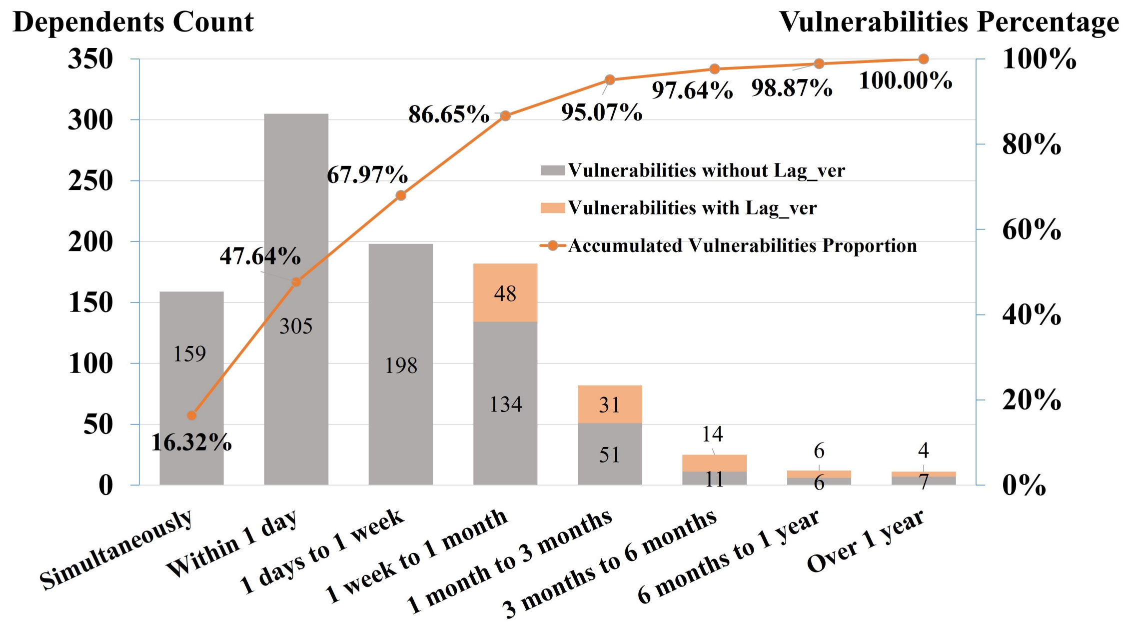


FIGURE 7.5: Distribution of LT_{ver} and Lag_{ver}[2]

To unveil the reasons for delayed patch version releases, we assessed Lag_{ver} within these vulnerable modules. After excluding vulnerabilities whose modules have no version released before fixing commit, 954 vulnerabilities were retained for analysis. Among them, 734 vulnerabilities had patch versions released within the expected cycles, while the rest 220 had unusually long LT_{ver}. Notably, 103 of them are with $LT_{ver} > 1week$, accounting for 10.8% of all analyzed vulnerabilities, and their Lag_{ver} as illustrated in Figure 7.5. We manually scrutinized the 62 repositories that these vulnerabilities belong to, and observed that 58 modules are still actively maintained in 2023, indicating that the delayed patch version releases have little correlation with the activeness of repositories.

Moreover, because it is common that the fixing commits (i.e., pseudo-versions) are directly used as a temporary patch versions, we further verified if pseudo-versions are prevalent in these 62 vulnerable modules so that it is not urgent to release patch versions. Out of 49 vulnerable modules that are ever been imported by other projects, only 23 repositories had dependents that used pseudo-versions to address vulnerabilities, which accounted for less than half of the vulnerabilities. This suggests that there could be lack of countermeasures for downstream users to patch vulnerable dependencies if the patch version tags are delayed.

Finding-4: Modules of 10.8% of vulnerabilities had obvious patch version release lags, while only less than half of the corresponding modules are proactively updated by dependents via pseudo versions, posing a lack of countermeasures to promote the distribution of patches.

7.4.2.3 Analysis of Lag of Index

From the 1,014 vulnerabilities. We first excluded 66 vulnerabilities which had negative Lag_{index} due to the version re-tagging after indexing. Out of the remaining 948 vulnerabilities, 67.09% of them have the Lag_{index} within a week, indicating that the patch versions were published in the Go Module Index promptly. Specifically, 297 (31.33%) of them have Lag_{index} within one hour. We manually went through repositories of these vulnerabilities and found that 135 out of 297 within 65 repositories have integrated continuous integration. The rest was highly likely to be performed by maintainers themselves to declare the versions in `go.mod` or use `go get` to index versions right after the release. This proactive approach by maintainers ensured the registration of the patch version in the Go Module Index, making it promptly available to downstream users.

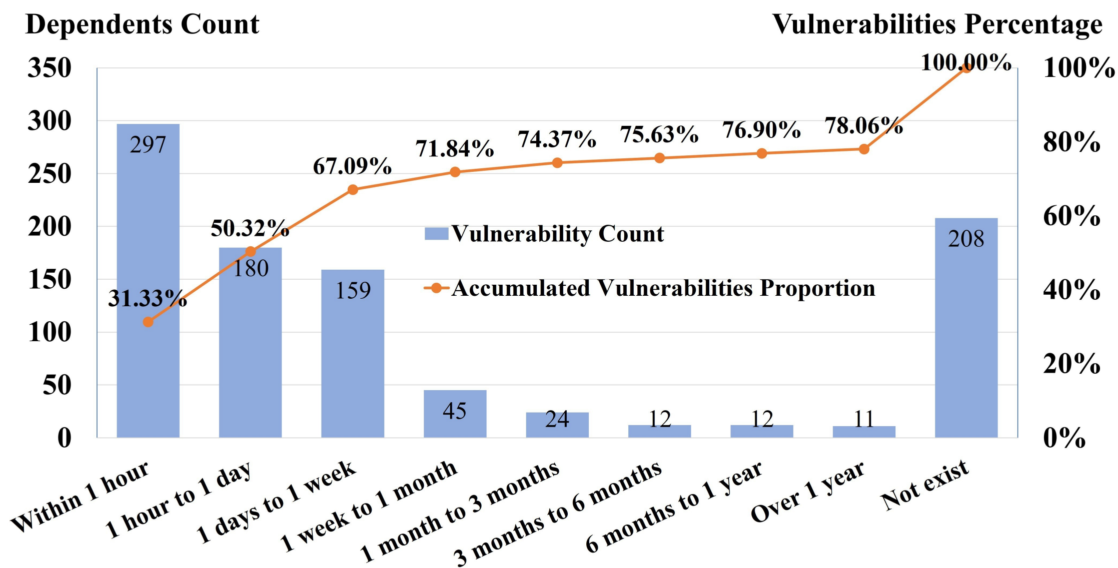


FIGURE 7.6: Distribution of Lag_{index}

On the contrary, there were 208 (21.94%) of vulnerabilities whose T_{index} was not available. These vulnerabilities had patch versions and the subsequent versions never utilized by users, and consequently, were not recorded in the Go Module

Index. As a result, users were unable to automatically access these patch versions via the Go Module commands, hindering their ability to effectively address the associated vulnerabilities. Considering that the patch versions are only indexed upon usage, we surmise that these modules could have limited users. We then collected their dependents to understand whether and how they addressed the vulnerabilities. It turned out that modules of 127 of these vulnerabilities had a total of 136, 480 dependents, and surprisingly, only 136 dependents have fixed the vulnerabilities via updating pseudo versions. Hence, the unindexed patch versions were mostly caused by rare usage and absent remediation.

Finding-5: A majority (67.09%) of vulnerabilities, demonstrated a swift patch version indexing. However, it is concerning that the patch versions of 21.94% of vulnerabilities were not indexed in the Go Module Index.

7.4.3 RQ3: Dependents Vulnerability Fixing

Apart from the perspective of maintainers, the adoption of patch versions by downstream users also greatly determines the life cycles of vulnerabilities. In this section, we conducted an analysis of how users address the vulnerabilities at a finer granularity by investigating the modification history of the Go Module dependency manifest file `go.sum` for each affected dependent module. We first summarized the methods employed by users and calculated the proportions of them over all affected dependents.

7.4.3.1 Data Preparation

To begin our analysis, we gathered and downloaded all dependents stored in GitHub that met the criteria of having a vulnerability with a T_{fix} date after April 10, 2019. This entailed downloading a total of 451, 013 dependents and 411, 860 succeeded for further analysis and investigation. Out of them, 253, 865 dependents had both `go.mod` and `go.sum` files, indicating that they were using Go Modules to manage their dependencies. However, for other dependents that lacked the *go.sum* file, we were unable to obtain the modification history of the dependency relations. Therefore, we excluded these dependents from our analysis. We applied Algorithm

4 to determine T_{dept}, timestamp of the latest commit that changes `go.sum` for each dependent.

7.4.3.2 Analysis of Fixes by Dependents

Generally, it is worth noting that only 11,446 (4.51%) successfully resolved all vulnerabilities we had collected. However, a significant number of dependents, 182,461, (71.87%) had not addressed any of the vulnerabilities.

In order to account for the number of dependents each vulnerability has, we quantified the vulnerability and dependent mappings using a unit that amalgamates a vulnerability with its dependent (referred to as a vulnerability-dependent, or VD). This approach resulted in a total of 744 vulnerabilities being included in our further examination. As illustrated in Figure 7.7, our analysis revealed that 163,930 VDs (7.81%) addressed the vulnerability by removing the vulnerable module from the `go.sum` file. Additionally, 319,961 VDs (15.24%) resolved the vulnerability by updating the vulnerable version. Among these VDs, 93,823 (29.32%) used the pseudo-version mechanism, while 226,138 (70.68%) used the patch version. Interestingly, the majority of VDs (76.95%) still retained the vulnerability without taking any measures.

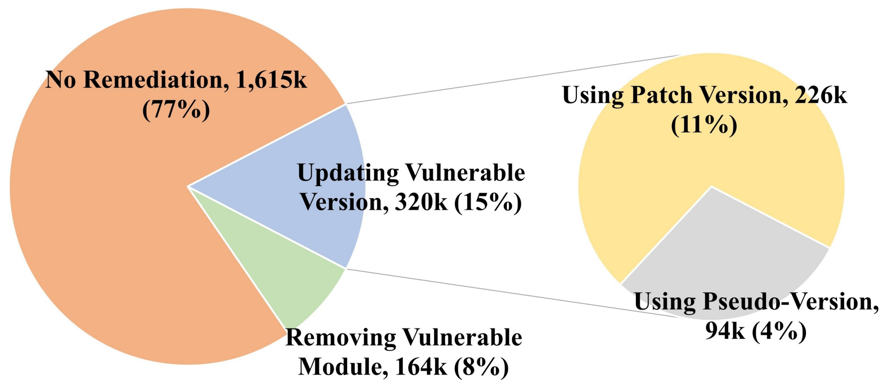


FIGURE 7.7: Distribution of Vulnerability-Dependent

Finding-6: Only nearly a quarter of the VDs have implemented measures to address the vulnerabilities. Among these VDs, the majority (66.12%) opted to update the vulnerable version. Out of them, 70.68% utilized the patch version to fix the vulnerabilities instead of utilizing the pseudo-version.

To assess the effectiveness of fixing commits and indexed patch versions, we tracked changes in the number of dependents over time. Using algorithm 4, we obtained the T_{dept} for each VD. To highlight the variations over time, we divided the timeline into 4 periods related to T_{dept}: *Before* T_{fix}, *Between* T_{fix} *and* T_{index}, *Within one month after* T_{index}, and *After one month from* T_{index}, as depicted in Figure 7.8. Note that 127 out of the 744 vulnerabilities did not have patch version indexed thus no T_{index} was available. Thus, they were categorized in *Between* T_{fix} *and* T_{index}.

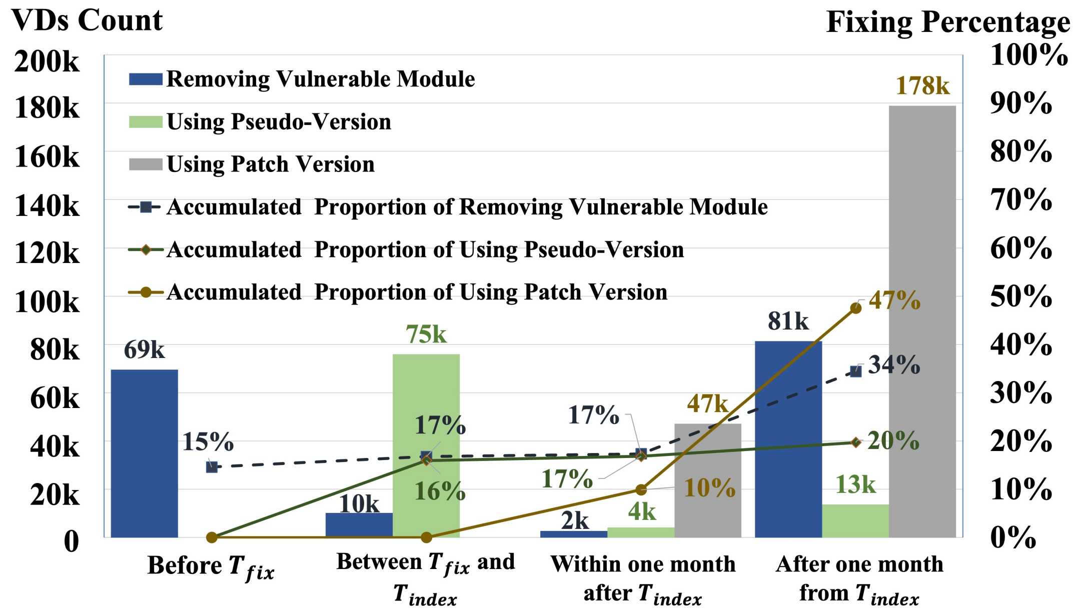


FIGURE 7.8: Distribution of Vulnerability-Dependents

When T_{dept} is earlier than T_{fix}, removing the vulnerable modules is the only method to address the vulnerabilities. We observed that 14.61% of 483,891 VDs that adopted addressed vulnerabilities opted to remove vulnerable modules during this period. Overall, we found that 34.42% of VDs addressed vulnerabilities with such an approach, making the usage even out over the timeline. We speculate the reasons that removing modules could be caused by debloating or other maintenance considerations so that removing modules have an insignificant relationship with fixing vulnerabilities.

When T_{dept} is earlier than T_{fix}, removing the vulnerable modules is the only method to address the vulnerabilities. Our observations indicate that 14.61% of the total 483,891 VDs employed this approach during the specified period. Overall, we found that 34.42% of VDs addressed vulnerabilities by removing the vulnerable

modules, making the usage even out over the timeline. We speculate the reasons that removing modules could be caused by debloating or other maintenance considerations so that removing modules has an insignificant relationship with fixing vulnerabilities.

In cases where T_{dept} falls between T_{fix} and T_{index}, the only available option to address the vulnerabilities while still preserving the modules was to upgrade to a pseudo-version. In this interval, pseudo-versions were employed to rectify vulnerabilities for 15.95% of VDs. It's noteworthy that according to the Go Module documentation [235], a pseudo-version does not constitute a stable release. In totality, a mere 19.70% of VDs opted for this method, the majority of which occurred during the period between T_{fix} and T_{index}.

When the indexed patch version is out, T_{dept} falls behind T_{index}. It is noteworthy that 9.92% of VDs upgraded vulnerable modules to patch versions within the first month, but there is a boost of patch version adoption after the first month by 47.49%. This indicates that when patch versions were released and made available in the Go Module Index, users were more inclined to fix the vulnerabilities by updating vulnerable modules with patch versions.

In conclusion, when the patch version is available, users are more likely to adopt the patch version for vulnerability fixing. While the patch version is unavailable, users prefer the pseudo-version over removing modules. Although Go Module allows the pseudo-version to serve as the fix at the first available moment, users still prefer the patch version based on SemVer, which further emphasizes the need to release the patch versions as soon as the fixing commit is out.

Finding-7: Using the patch version was the preferred method for fixing vulnerabilities among most users, with 47.49% of all VDs. Maintainers should release patch versions as soon as possible when fixing commits are available.

7.4.4 RQ4: Fixing Lagging Inquiry

In the preceding sections, we quantitatively highlighted the lags associated with vulnerability fixes and explore how both maintainers and dependents contributed to these lags. Specifically, in this section, we delve into the reasons for 1) the absence of patch version releases, 2) the delays of patch version releases, 3) the missing

indexing of patch version to Go Module Index, and 4) the missing patches to vulnerable dependencies, by combining inquiries to maintainers and manual analysis. Note that, generally, in all inquirying issues, we follow the similar structure: First, we declared us as a research team (to keep anonymous), and presented the information of specific vulnerabilities, including the CVE id and reference links. Next, for each specific types of lags, we explain the corresponding lags and potential threats to downstream users. After that, we either provided our recommendations on countermeasures and ask for their opinion, or directly asking for the reasons for corresponding delays. Since the community has been continuously replying to our questions, we would update the latest feeds at our website [191].

7.4.4.1 Absent Patch Version Releases

In our analysis of the 40 vulnerabilities that were not released with a patch version, we proactively engaged with the maintainers by submitting issues to inquire about the reasons for the absence. Among these, modules of 20 vulnerabilities have never released any tag, indicating that all dependents have to use pseudo-versions. These modules were excluded because the Lag_{ver} was not applicable.

For the remaining vulnerabilities, we submitted 20 queries via GitHub issue and received responses from 8 maintainers. In these issues, we ① initiated elaborate the situation that no patch versions were released for the vulnerabilities even if they were already fixed with commits, ② such missing patch version could result in users being unable to conveniently retrieve the latest patch versions, i.e., via the Go Module tool "go list", and ③ we recommend that maintainers to tag patch versions for these fixing commits to facilitate downstream users fixing the vulnerabilities. The detailed query structures are available on our website [191].

Among these responses, developers of 3 repositories agreed and replied to us with the patch versions. Regarding the other 5 queries, one maintainer agreed with our observations and proposed that users could use the branch name *master* as the version to obtain the latest pseudo-versions. However, this is considered provisional as Go Module would automatically resolve *master* as the latest pseudo-version within the branch and update the `go.mod` file with the pseudo version number without assurance of compatibility. Thus the recommended course of action is for maintainers to release a dedicated patch version for downstream users. In contrast,

one maintainer responded that the project is not designed to be used as a library for other users, and therefore, the maintainer does not prioritize addressing the vulnerability for external users. The rest 3 maintainers closed the queries without providing any response.

Finding-8: Considering that developers from 3 repositories have not previously disclosed their patch versions, it could facilitate the adoption of patches if maintainers explicitly mention the vulnerability fixing in patch releases.

7.4.4.2 Delayed Patch Version Releases

Next, we investigated the reasons behind the delayed release of patch versions. Conservatively, all vulnerabilities with $LT_{ver} > 1$ month were included in this analysis. Specifically, we manually examined the release notes and commit branches of the patch versions, resulting in 103 vulnerabilities to further categorize the reasons. Typically, there are two typical types of delays for tagging patch versions. 1) For vulnerabilities whose fixing commits are followed with commits that changes functional codes before version tagging commits, we suspect these delays are waiting for ① **Other commits have to be incorporated into one release**. 2) For vulnerabilities whose fixing commits are not followed with commits of code-related changes till version tagging commits, there could be two types of reasons that are subjective to maintainers, ② **Issues with testing and CI checking**: the maintainers need more time to examine the changes, and ③ **By convention, infrequent release of versions**: they are just waiting for regular releases.

Therefore, we take them as the primary reasons for patch version tagging delays, and since patch versions are just delayed instead of absent, after elaborating on the delays of specific vulnerabilities, we directly ask the maintainers for their reasons by choosing from these primary reasons or give ④ **Other** reasons, via GitHub issues. After excluding 7 vulnerabilities we are unable to reach maintainers (i.e., read-only repositories or no issues allowed), and 9 vulnerabilities are fixed before the first version tag (i.e., no release cycles), issues are submitted to the repositories of 87 vulnerabilities.

Till Nov 2023, we only received responses of reasons for 11 vulnerabilities. However, surprisingly, all these responses have proactively provided much more detailed reasons to us. 1) (3 cases) The patch versions we found delays are actually not tagged

on the version branch where the vulnerabilities are first discovered and patched, and they also provided the first patch versions. However, based on our observation, it still takes a very long time for them to be merged into the main branches. 2) (2 cases) the vulnerabilities were first identified in a development branch for the next minor versions, and they had to wait for the first stable versions before tagging, however, they have quickly back-ported the patches to previous version tracks and the follow-up patch versions are quickly tagged. 3) (2 cases) These vulnerabilities are not already patched by their intended users, so no need to push the patch versions to the public in a fast pace. 4) (2 cases) These patch versions are lagged due to the lack of availability of maintenance.

These reasons indicate the correlation between the complex repository management strategies and the delays of patch version tagging. Therefore, we conducted a manual analysis of the locations of fixing commits and patch versions for all patch version tagging, and further categorized these reasons. (1) Multiple version tracks, sometimes even the minor version tracks, are maintained on different branches, while the delays of patch version tagging among them varies (37 cases). (2) Vulnerabilities are fixed during the development of new versions, these patch commits are unable to be tagged before stable version tags (12 cases). (3) Fixing commits are committed in different branches, and it takes more time to backport patches to other branches (9 cases). (4) Some patches are introduced when conducting breaking changes, such as upgrading to next major version (3 cases). Apart from these, there are still 40 cases we are unable to identify explicit reasons, and we will continue to update the feedback from maintainers on our website [191].

Finding-9: The complex repository management strategies contribute the most to the delays of patch version tagging, especially the inconsistent version management on branches for different purposes.

7.4.4.3 Not Pushing Patch Version to Go Module Index

As revealed in Section 7.4.2.3, we identified 208 vulnerabilities whose released patch versions were not indexed in Go Module. Since modules could be indexed only when they are imported by other projects, we excluded vulnerabilities whose patch versions are not in Go Module Index, and analyze the usage of vulnerable versions of the corresponding modules of the rest 112 vulnerabilities from 30 repositories.

Since maintainers could easily index the patch versions by `go get` command, it is critical to understand the reasons for this absent action and inform them of the criticality. Before submitting inquiries, we manually examined the repositories to seek possible reasons. We found 4 repositories (6 vulnerabilities) stated that they were applications, not third-party modules, implying no intent to index patch versions. Additionally, 6 repositories (7 vulnerabilities) used version tags that do not follow Go Module versioning requirements [241], rendering their patch versions unindexable.

After excluding these cases, we open issues for the remaining 99 vulnerabilities. Specifically, after stating our observations that their patch versions for specific vulnerabilities are not indexed in Go Module Index, we advised maintainers to leverage Go Module commands like *go get module@version* or similar methods to facilitate patch version indexing after tagging patch version for vulnerabilities.

Till Nov 2023, we received responses for 37 vulnerabilities. It is noteworthy that we get positive responses on the necessity of indexing patch versions. Specifically, responses on 10 vulnerabilities indicated their willing to take our suggestions to execute the *go get* command, 3 of them have pushed patch versions to Go Module Index and the rest 7 sent relative information to their security teams. The responses on the rest vulnerabilities did not take actions due to various reasons: 1) responses of 15 vulnerabilities replied that their projects are products instead of third-party modules, while according to our data, they have hundreds of dependents importing vulnerable versions; 2) responses of 7 vulnerabilities indicated that their modules are not directly imported via Go Module and they have already provided necessary countermeasures to facilitate downstream use. 3) responses of 5 vulnerabilities stated they are not the correct repositories to raise issues due to migration or vulnerability reference error. Given that all maintainers replied with positive feedback, it is anticipated that the timely indexing of patch versions should be welcomed by the community. However, considering that not all maintainers are willing to voluntarily do so, there still lacks proper mechanisms to actively search and index such new patch versions to facilitate in-time distribution.

Finding-10: All maintainers who responded agreed with our observations and suggestions. Nevertheless, to facilitate patch propagation, the Go Module release mechanism should index new versions as soon as possible without relying on maintainers.

7.4.4.4 Not Fixing the Vulnerability

To ensure the popularity, we first selected the top 120 most-starred dependent repositories of vulnerable modules for analysis, and for each dependent, we localize the vulnerabilities in their dependencies based on our dependency relations. If vulnerable dependencies remained unaddressed by the data collection date, we notified maintainers by submitting issues detailing the persistence of vulnerable dependencies and the available patch versions with recommended dependency upgrades. Considering that different vulnerabilities have different impact and exploitability, we made inquiries respectively based on VDs. In total, 337 VDs (120 dependents) were queried.

Among the 337 VDs analyzed, we received responses from maintainers of 235 VDs (72 dependents). Specifically, maintainers of 202 VDs (61 dependents) accepted suggestions to update the vulnerable dependencies. However, in the responses of 16 VDs (8 dependents), maintainers disagreed with update suggestions for the following reasons: (1) The projects were not using the vulnerable functions (10 VDs, 4 dependents); (2) The vulnerable modules were rarely used, and addressing the vulnerabilities would require updating the Go version, which could be troublesome (4 VDs, 2 dependent); (3) There was a compatibility issue preventing upgrading the module (1 VD, 1 dependent); (4) The maintainer suggested downstream users address the vulnerability by themselves (1 VD, 1 dependent).

Finding-11: The majority of maintainers (89.95%) addressed the vulnerabilities in a timely manner once they became aware of the presence of vulnerabilities in their projects.

7.5 Discussion

7.5.1 Recommendations

As an early adopter of decentralized package management, Golang originally encouraged direct repository imports as dependencies by commit hashes (i.e., pseudo versions) without releases to public registries [242]. However, with the growing

complexity of project size, Go Modules introduced the SemVer tags for better version control management [243], leading to a chaotic situation where both SemVer for compatibility concerns and pseudo-versions for swift development run on parallel as revealed in RQ1. Therefore, we highlight the recommended solutions for different stakeholders, as well as future research directions, to mitigate such severe situation.

Maintainer Perspective. Several challenges arise due to the module distribution and versioning practices in the Golang ecosystem. Our study highlights significant delays in releasing vulnerability fixes (i.e., Lag_{ver} and Lag_{index}), hindering the adoption of patch versions. To mitigate this, maintainers should prioritize timely tagging and indexing of updates. Additionally, as revealed in RQ4, some maintainers favor users importing directly from the *master* branch to keep updated, which could result in compatibility issues. This calls a stronger focus to ensure compatibility. Lastly, maintainers often delay tagging fixing commits until sufficient feature additions are made, risking user exposure to vulnerabilities. Balancing the urgency of vulnerability fixes with feature development in version releases is essential for maintaining ecosystem security.

User Perspective. Vulnerabilities persist in many modules that once imported vulnerable dependencies, even though users often update to patch versions when available. This issue arises because *go.sum* naturally locks the dependencies if they still satisfy the declarations in *go.mod* to avoid incompatibility. However, this can delay updates that include essential fixes. We suggest users periodically unlock their dependencies to allow timely updates. Additionally, our findings show that many users address vulnerabilities because they get alerts from other users or SCA tools like Dependabot. Therefore, we recommend incorporating vulnerability checks from third-party auditors or SCA tools during their testing, i.e., integrated in CI/CD pipelines, to ensure security updates are promptly applied.

OSS governance Perspective. The absence of a centralized registry in Golang ecosystem limits its ability to track vulnerabilities in modules. This is crucial given the increasing emphasis on supply chain security. Other ecosystems, like Maven and NPM, have integrated security tools and databases, such as CVE mappings and *npm audit*, enhancing their security posture. Go Module could benefit from similar features. Implementing a database for vulnerability and patch data would aid in distributing fixes to users. Moreover, addressing compatibility issues arising from

using pseudo versions for fixing vulnerabilities could be achieved by incorporating compatibility-checking tools for dependency upgrades.

We also highlight some possible research directions. 1) Vulnerability Data Enhancement: In the data preparation, considerable vulnerabilities could not be associated with their partial or all fixing commits. Vulnerability data quality would be critical to conduct finer-grained monitoring and governance for the ecosystems. Therefore, techniques for identifying these patches, or even the root cause (i.e., vulnerable functions) and proof of concepts (POCs), could be further researched. 2) In-depth Vulnerability Impact Analysis: Given the frequent reuse of vulnerable modules in Golang, it is essential to introduce more nuanced analysis, such as vulnerability reachability analysis. These would boost identifying and prioritizing the most critical vulnerabilities for timely remediation. 3) Robust Compatibility Checks: Compatibility concerns are paramount when updating pseudo versions in Golang. Implementing rigorous compatibility checks prior to upgrades can alleviate user apprehensions and encourage the adoption of patches. Furthermore, SCA tools should include pseudo-version upgrades as part of their vulnerability remediation toolkit, particularly when patch versions are not available and compatibility can be thoroughly verified.

7.5.2 Limitations and Threats to Validity

1) Since Go Module Index only record modules that are ever imported as dependencies, we are unable to find Golang modules if they are not never used by others. To ensure data completeness, we retrieve modules and dependencies from Open Source Insight [41], which collects these data from various sources. Our dataset is also statistically more complete than *libraries.io*, which further ensure the representativeness. 2) There could be vulnerabilities that are never discovered or reported, we are unable to perfectly find out all vulnerabilities. There are still considerable vulnerabilities that we are unable to locate their fixing commits based on existing information. Even so, we have tried our best to collected vulnerability information from the most mainstream platforms and validate the coverage by comparing with OSV and GitHub advisory. 3) Downstream users could accidentally remove vulnerable dependencies for other purpose instead of excluding vulnerabilities, which could bias our analysis. However, this would not influence the conclusions since

the majority of dependents are still vulnerable even if we take that as a fix. 4) Apart from the 66 cases whose version tag time is later than indexing time, there could be more cases of re-tagging on GitHub, which could underestimate the existence of Lag_{index}, while There is no practical solution to accurately check potential re-tagging in all repositories. Moreover, even so, our conclusion on the existence of Lag_{index} would not be affected. 5) We introduced ourselves in the issues raised for maintainers by mentioning we were a research team on Go Module, which might cause biases in the responses. We have manually processed the responses to attempt to eliminate the biases from the summarized results.

7.6 Conclusion

Our analysis revealed that vulnerabilities significantly affected 66.10% of the modules, while 62.85% of the dependents had not addressed these vulnerabilities. We quantitatively proved that the timely patch release and indexing could greatly facilitate the patch adoption by downstream users. Through the inquiries about reasons behind lagged patch release, indexing, and adoption, we identified the deficiencies of countermeasures to fulfill the capability to accelerate the propagation of vulnerability patches in the Golang ecosystem. Practical recommendations for the perspectives of different stakeholders and possible research directions, are also concluded to facilitate the enhancement of security for the Golang ecosystem.

Chapter 8

Future Work

Looking ahead, following the track of OSS security, I aim to extend my research to encompass comprehensive and holistic remediation. Over time, academic and commercial vulnerability detection tools have significantly evolved, and numerous security alerts have been initiated to draw maintainers' attention. However, the number of these alerts can be overwhelming and time-consuming to address for human developers, let alone remediating various forms and sources of TPLs. However, solving the potential conflicts and issues brought by remediation presents its own set of challenges. There is no well-recognized remediation solution so far, and existing tools provide various but ineffective remediation strategies.

It is anticipated that the remediation should handle OSS formats like source code, bytecode, and binary. Apart from upgrading, the approaches will also include patching and migration for those vulnerabilities without secure versions. Notably, besides the PM-based dependencies, the remediation will also include the clone-based dependencies which are introduced by copy-pasting. Taking all the above factors into account, remediation becomes a complex task.

As illustrated in Figure 8.1, the remediation includes four procedures: Cross-scope conflict resolution, Context-aware and code-centric prioritization, Incompatibility and license conflicts precaution, and Comprehensive remediation with global optimum.

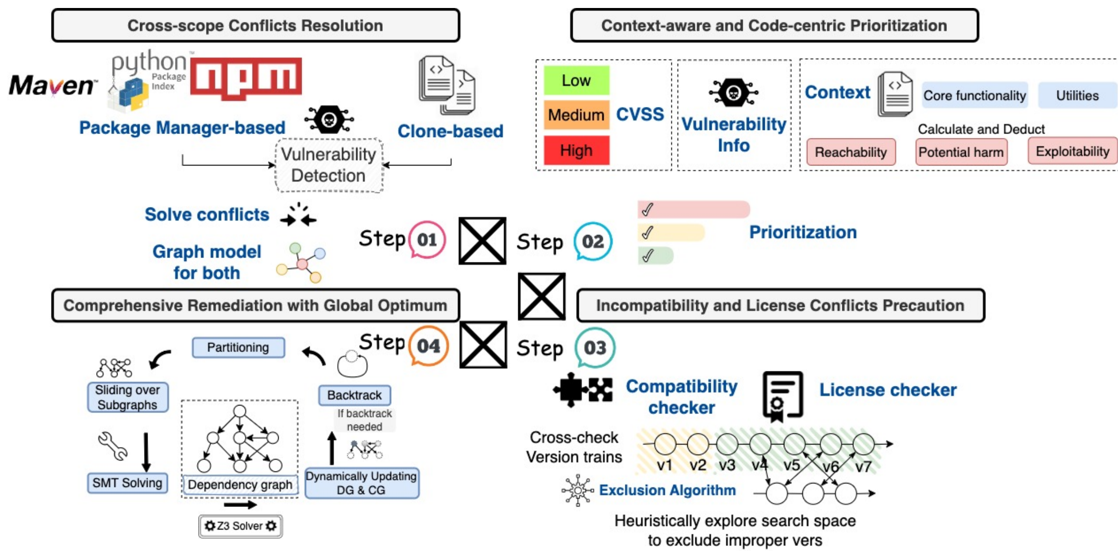


FIGURE 8.1: Roadmap of Future Work

- **Cross-scope conflict resolution**: Besides the PM-based TPLs, according to a study conducted by Lopes et al. [244], the code-clone of OSS projects is prevalent in software projects. Specifically, cloned code and self-defined dependencies are managed in different scopes. Due to the different scopes, the remediation approach varies. The clone-based TPLs in source code form should be fixed by patching or backporting, and the PM-based TPLs in any form are usually fixed by upgrading or migration. Considering that the main challenge is the potential conflicts caused by the dependency relationship between two scopes and the large search space, tentatively, I plan to use a genetic algorithm with constraints of non-conflict outcome starting from the initial states of TPLs and attempt to derive the plausible solution during evolution.

- **Context-aware and code-centric prioritization**: Given the detected vulnerabilities, my plan aims to prioritize the severe vulnerabilities over minor threats. Traditional prioritization practices involve assigning a severity score to rank vulnerabilities in descending order. However, these scores can differ substantially across tools due to varied algorithms and factors that they consider. Moreover, pinpointing the most pressing vulnerabilities among an overwhelming number of alarms remains an ongoing research challenge. Typical factors that influence the calculation of severity scores include the Common Vulnerability Scoring System scores [213], which offer a standardized metric to gauge the severity of vulnerabilities, the exploitability and the potential harm that can arise once the

vulnerability is exploited. However, I recognize that the severity and accessibility of vulnerabilities are subject the code context. In other words, different functionality of code should be treated accordingly. The target of this research is to differentiate the severity of vulnerabilities based on code context.

- **Incompatibility and license conflicts precaution:** Remediation is conducted by patching, upgrading, and migration, which inevitably alters the software. The altering could sometimes induce other problems, such as incompatibility and license conflicts. Incompatibility usually occurs during upgrading. The common practices of detecting incompatibility issues across versions rely on calculating the incompatible APIs of TPLs that have breaking changes between the code of two versions and then checking if the APIs are used by the user project. If any incompatible API is in use, the risk should be circumvented during remediation.

- **Comprehensive remediation with global optimum:** With the additional clone detection, remediation prioritization, and induced issue detection, holistic remediation is possible to be achieved. We aim to transcend local solutions and achieve a global optimum derived by calculating based on conditions from previous steps. The inherent challenge in remediation is achieving a balance between optimality and efficiency, because the solution search space is enormous. Like our previous attempt, we plan to split the major task into unrelated or weakly related sub-tasks to break down the primary problem.

Chapter 9

Conclusion

The primary aim of this thesis was to explore and address the vulnerabilities and incompatibilities within OSS libraries, both in individual applications and across ecosystems. My initial research introduced an innovative method for mitigating vulnerabilities in TPLs of Java projects. This approach not only maintained backward compatibility but also demonstrated that strategic upgrades could resolve the majority of vulnerabilities in widely used Java TPLs. Our tool, *Coral*, delivered a comprehensive remediation solution for Maven-managed projects, outperforming state-of-the-art tools in both vulnerability resolution and compatibility assurance.

Our subsequent study expanded to address more complex types of incompatibilities, specifically behavioral incompatibilities, also referred to as semantic breaking changes. Our semantics-enriched methodology allowed for static detection of such incompatibilities. Through in-depth analysis, we developed a reliable strategy to identify potential semantic breaking changes within API pairs, thereby preventing likely disruptions during software updates. Our findings revealed a higher incidence of incompatibility risks in upgrades of popular Maven libraries over those detected by traditional tools.

Shifting focus to the broader Maven ecosystem, we observed that vulnerabilities in the Maven ecosystem were often not patched through automated means. Investigation into the underlying causes revealed that Soft Version Constraints were hindering the application of available patches along dependency paths. In response, we developed a tool, *Ranger*, which recalibrated compatible version ranges to leverage

Maven's predefined rules for automatic vulnerability remediation. Our evaluations showed that *Ranger* could effectively auto-fix over 90% of disclosed vulnerabilities.

The final segment of research turned to the Golang ecosystem. Despite its innovative attempts at decentralizing TPL management, these fresh mechanisms inadvertently introduced lags in vulnerability remediation, thus extending the lifecycle of vulnerabilities. Our study of the prevalence and underlying causes of these lags led to actionable recommendations for both library maintainers and end-users, aimed at expediting the resolution process.

Looking ahead, my commitment is to further the work on OSS security by devising comprehensive remediation solutions for security issues detected by SCA and Static Application Security Testing (SAST) tools. This ongoing endeavor will contribute to enhancing the security and governance of OSS and the software supply chain.

List of Author's Awards, Patents, and Publications[1]

Awards

- **ACM SIGSOFT Distinguished Paper Award**, "Has my release disobeyed semantic versioning? static detection based on semantic differencing" *the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE22)..*
- **ACM SIGSOFT Distinguished Paper Award**, "Compatible remediation on vulnerabilities from third-party libraries for java projects" *the 45th IEEE/ACM International Conference on Software Engineering (ICSE23)*

Conference Proceedings

- **Lyuye Zhang**, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu, "Has my release disobeyed semantic version- ing? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software En- gineering, 2022.*
- **Lyuye Zhang**, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu, "Compatible remediation on vulnerabilities from third-party libraries for java projects," in *Proceedings of the 45th Interna- tional Conference on Software Engineering, 2023.*

[1] The superscript * indicates joint first authors

- **Lyuye Zhang**, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu, "Mitigating persistence of open-source vulnerabilities in maven ecosystem," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, 2023.*

- Jinchang Hu, **Lyuye Zhang** (equal contribution as 1st author), Chengwei Liu, Sen Yang, Song Huang, and Yang Liu, "Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem," in *Proceedings of the 46th International Conference on Software Engineering, 2024.*

Bibliography

[1] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556956. URL `https://doi.org/10.1145/3551349.3556956`. vii, 97, 102

[2] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. Compatible remediation on vulnerabilities from third-party libraries for java projects. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2540–2552. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00212. URL `https://doi.org/10.1109/ICSE48619.2023.00212`. 21

[3] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–203. IEEE Computer Society, 2023. 22

[4] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. Empirical analysis of vulnerabilities life cycle in golang ecosystem, 2023. vii

[5] Github report. `https://octoverse.github.com/`, 2021. 1, 8

[6] Software Composition Analysis. `https://snyk.io/series/open-source-security/software-composition-analysis-sca/`, 2023. 2, 21

[7] Eclipse Steady. `https://projects.eclipse.org/proposals/eclipse-steady`, 2023. 2

[8] Scantist. `https://scantist.com/`, 2023. 2

[9] Snyk. `https://snyk.io/`, 2023.

[10] Dependabot. `https://github.com/dependabot`, 2023. 11

[11] OWASP Dependency Check. https://owasp.org/www-project-dependency-check/, 2023. 11, 18

[12] Blackduck. https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html, 2023.

[13] White Source. https://www.whitesourcesoftware.com/, 2023. 2, 18

[14] Maven. https://maven.apache.org/, 2023. 2, 7, 8, 10

[15] Golang, 2023. URL https://go.dev/. (Accessed on 04/09/2023). 2, 7

[16] Node package manager (npm). https://www.npmjs.com/, 2023. 7, 103

[17] Semantic Versioning. https://semver.org, 2021. 8, 10, 19, 26, 82, 106

[18] Maven repositories. https://mvnrepository.com/, 2023. 8, 10, 11, 14, 28, 43, 63, 65, 82

[19] National vulnerability database. https://nvd.nist.gov/, 2023. 9, 12, 68

[20] Osv. https://osv.dev/, 2023. (Accessed on 02/17/2023). 9, 113

[21] Github Security Advisory. https://github.com/advisories, 2023. 9, 82, 113

[22] Snyk Vulnerability Database. https://security.snyk.io/, 2023. 9, 82

[23] Definition of software compatibility. "https://en.wikipedia.org/wiki/Backward_compatibility", 2022. 10

[24] Linking in java. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html, 2013. 10

[25] Apache Hadoop. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2019. 10

[26] Hadoop HDFS api breaking issue. https://issues.apache.org/jira/browse/HDFS-14595, 2019. 10

[27] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in Maven central. *arXiv preprint arXiv:2110.07889*, 2021. 10, 19, 21, 102

[28] Ponta, Serena Elisa and Plate, Henrik and Sabetta, Antonino. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020. 11, 17

[29] Katheryn A Farris, Ankit Shah, George Cybenko, Rajesh Ganesan, and Sushil Jajodia. Vulcon: A system for vulnerability prioritization, mitigation, and management. *ACM Transactions on Privacy and Security (TOPS)*, 21 (4):1–28, 2018. 11, 18

[30] Behavioral Compatibility Definition. https://wiki.openjdk.java.net/display/csr/Kinds+of+Compatibility, 2019. 12

[31] Jour. http://jour.sourceforge.net/signature.html, 2008. 12

[32] Japitools. https://savannah.nongnu.org/projects/japitools/, 2004.

[33] clirr. https://www.mojohaus.org/clirr-maven-plugin/index.html, 2016. 43

[34] jchecker. https://github.com/trohovsky/japi-checker, 2015. 43

[35] sigtest. https://docs.oracle.com/javacomponents/sigtest-3-1/user-guide/toc.htm, 2014. 43

[36] revapi. https://revapi.org/revapi-site/main/index.html, 2021. 43, 63, 97

[37] japi-compliance-checker. https://lvc.github.io/japi-compliance-checker/, 2019. 12, 43, 63, 97

[38] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 112–124, 2020. 12, 19, 46, 65

[39] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 215–225, 2017. 12, 19, 27, 36, 42, 65

[40] Log4j Remote Code Execution. https://www.netskope.com/blog/cve-2021-44832-new-vulnerability-found-in-apache-log4j, 2021. 12

[41] Google Open-source Insight. https://blog.deps.dev/, 2023. 12, 21, 108, 129

[42] Log4j Vulnerability News. https://www.securityweek.com/one-year-later-log4shell-remediation-slow-painful-slog/, 2023. 12

[43] Log4j Vulnerability News. https://thenewstack.io/one-year-of-log4j, 2022.

[44] Log4j Vulnerability News. https://securityintelligence.com/articles/log4j-vulnerability-changed-oss-cybersecurity/, 2023.

[45] Log4j Vulnerability News. https://asia.nikkei.com/Spotlight/Datawatch/Cyberattacks-on-Japan-soar-as-hackers-target-vulnerabilities, 2023.

[46] Log4j Vulnerability News. `https://www.cybersecuritydive.com/news/cves-rise-2023-struggle-to-patch/641955/`, 2023. 12

[47] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zhou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *45th International Conference on Software Engineering*, pages 1–12, 2023. 12, 20, 21, 22

[48] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: a large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173. IEEE, 2021. 21

[49] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531, 2020. 12, 13, 20, 21

[50] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. *arXiv preprint arXiv:2301.07972*, 2023. 21

[51] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 333–343. IEEE, 2019.

[52] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 344–348. IEEE, 2019. 20, 21

[53] Dongdong Du, Xingzhang Ren, Yupeng Wu, Jien Chen, Wei Ye, Jinan Sun, Xiangyu Xi, Qing Gao, and Shikun Zhang. Refining traceability links between vulnerability and software component in a vulnerability knowledge graph. In *International Conference on Web Engineering*, pages 33–49. Springer, 2018. 12, 21

[54] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2023. 12, 13, 20, 21, 95

[55] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 2023. 12

[56] 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*, pages 1–10, 2018. 13, 18, 91

[57] Home - open source security foundation. `https://openssf.org/`, 2023. (Accessed on 02/12/2023). 13, 21

[58] ossf/wg-best-practices-os-developers: The best practices for oss developers working group is dedicated to raising awareness and education of secure code best practices for open source developers. `https://github.com/ossf/wg-best-practices-os-developers`, 2023. (Accessed on 02/12/2023). 13, 21

[59] Openssf scorecard. `https://securityscorecards.dev/#what-is-openssf-scorecard`, 2023. (Accessed on 02/14/2023). 13, 21

[60] About - git, 2023. URL `https://git-scm.com/about/info-assurance`. (Accessed on 03/29/2023). 13

[61] Go modules services, 2023. URL `https://proxy.golang.org/`. (Accessed on 03/27/2023). 13, 115

[62] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 672–684, 2022. 13

[63] Go modules reference - module-proxy, 2023. URL `https://go.dev/ref/mod#module-proxy`. (Accessed on 03/25/2023). 13, 108

[64] Maven. `https://https://maven.apache.org/`, 2024. 13

[65] Using go modules - the go programming language, 2023. URL `https://go.dev/blog/using-go-modules`. (Accessed on 03/27/2023). 15

[66] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. On the use of dependabot security pull requests. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 254–265. IEEE, 2021. 17

[67] 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. 18

[68] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018. 17

[69] César Soto-Valero, Thomas Durieux, and Benoit Baudry. A longitudinal analysis of bloated Java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1021–1031, 2021. 17

[70] Dependabot. `https://docs.github.com/en/code-security/dependabot/dependabot-security-updates/about-dependabot-security-updates`, 2023. 17, 54

[71] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2021. 17, 18

[72] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for OSS vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 2021. 18

[73] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE international conference on software analysis, Evolution and Reengineering (SANER)*, pages 446–457. IEEE, 2021.

[74] Sourceclear. `https://www.sourceclear.com`, 2023.

[75] Sonarqube. `https://www.sonarqube.org/`, 2023.

[76] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has my release disobeyed semantic versioning? static detection based on semantic differencing, 2022. URL `https://arxiv.org/abs/2209.00393`.

[77] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1695–1707. IEEE, 2021. 21

[78] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for Android applications: Are we there yet? In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 919–930. IEEE, 2020. 22

[79] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[80] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 146–157, 2019.

[81] Guangke Chen, Sen Chenb, Lingling Fan, Xiaoning Du, Zhe Zhao, Fu Song, and Yang Liu. Who is real bob? adversarial attacks on speaker recognition systems. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 694–711. IEEE, 2021.

[82] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 60–71. IEEE, 2019.

[83] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 120–131, 2018.

[84] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[85] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[86] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 245–256, 2017.

[87] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 627–637, 2017.

[88] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2095–2108, 2018.

[89] Run Wang, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yihao Huang, Jian Wang, and Yang Liu. Fakespotter: A simple yet robust baseline for spotting ai-synthesized fake faces. *arXiv preprint arXiv:1909.06122*, 2019.

[90] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[91] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. Masterkey: Automated jailbreaking of large language model chatbots.

[92] Yang Liu, Armin Sarabi, Jing Zhang, Parinaz Naghizadeh, Manish Karir, Michael Bailey, and Mingyan Liu. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 1009–1024, 2015.

[93] Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *International symposium on leveraging applications of formal methods, verification and validation*, pages 307–322. Springer, 2008.

[94] Hua Qi, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, Wei Feng, Yang Liu, and Jianjun Zhao. Deeprhythm: Exposing deepfakes with attentional visual heartbeat rhythms. In *Proceedings of the 28th ACM international conference on multimedia*, pages 4318–4327, 2020.

[95] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 678–689, 2016.

[96] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676, 2018.

[97] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE transactions on information forensics and security*, 11(2):289–302, 2015.

[98] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. Prompt injection attack against llm-integrated applications. *arXiv preprint arXiv:2306.05499*, 2023.

[99] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 477–487, 2019.

[100] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[101] Felix Juefei-Xu, Run Wang, Yihao Huang, Qing Guo, Lei Ma, and Yang Liu. Countering malicious deepfakes: Survey, battleground, and horizon. *International journal of computer vision*, 130(7):1678–1734, 2022.

[102] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic extraction of web authentication protocols from implementations. 2013.

[103] Lan Fu, Changqing Zhou, Qing Guo, Felix Juefei-Xu, Hongkai Yu, Wei Feng, Yang Liu, and Song Wang. Auto-exposure fusion for single-image shadow removal. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10571–10580, 2021.

[104] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[105] Jianwen Sun, Tianwei Zhang, Xiaofei Xie, Lei Ma, Yan Zheng, Kangjie Chen, and Yang Liu. Stealthy and efficient adversarial attacks against deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5883–5891, 2020.

[106] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

[107] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020.

[108] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *2019 IEEE/ACM 41st international conference on software engineering (ICSE)*, pages 596–607. IEEE, 2019.

[109] Qing Guo, Jingyang Sun, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Wei Feng, Yang Liu, and Jianjun Zhao. Efficientderain: Learning pixel-wise dilation filtering for high-efficiency single-image deraining. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 1487–1495, 2021.

[110] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182, 2020.

[111] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 2020.

[112] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[113] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. Patchfinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 590–602, 2024. 18

[114] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the NPM package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018. 18, 22

[115] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 2022. 18

[116] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE, 2019. 18

[117] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[118] 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.

[119] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 672–684, 2022. 95

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

[121] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213, 2016.

[122] 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. 18

[123] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. Towards understanding third-party library dependency in C/C++ ecosystem. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[124] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 810–822. IEEE, 2019.

[125] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1310–1322, 2020.

[126] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. Comparison and evaluation on static application security testing (sast) tools for java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 921–933, 2023.

[127] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. A comprehensive study on quality assurance tools for java. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 285–297, 2023.

[128] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. Static application security testing (sast) tools for smart contracts: How far are we? *Proceedings of the ACM on Software Engineering*, 1(FSE):1447–1470, 2024. 18

[129] Patrick Lam, Jens Dietrich, and David J Pearce. Putting the semantics into semantic versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 157–179, 2020. 19

[130] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014. 19, 21, 102

[131] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158, 2017. 19

[132] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47 (6):1226–1240, 2019. 19, 21, 102

[133] Rabe Abdalkareem, Md Atique Reza Chowdhury, and Emad Shihab. A machine learning approach to determine the semantic versioning type of npm packages releases. *arXiv preprint arXiv:2204.05929*, 2022. 19

[134] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. How Android developers handle evolution-induced api compatibility issues: a large-scale study. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 886–898. IEEE, 2020. 20

[135] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for Android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 532–542, 2018. 20

[136] Kamil Jezek, Jens Dietrich, and Premek Brada. How Java apis break–an empirical study. *Information and Software Technology*, 65:129–146, 2015.

[137] Kamil Jezek and Jens Dietrich. Api evolution and compatibility: A data corpus and tool evaluation. *J. Object Technol.*, 16(4):2–1, 2017. 20, 43

[138] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017. 65

[139] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[140] Jens Dietrich, Kamil Jezek, and Premek Brada. What Java developers know about compatibility, and why this matters. *Empirical Software Engineering*, 21(3):1371–1396, 2016.

[141] Evolving Java-based APIs. `https://wiki.eclipse.org/Evolving_Java-based_APIs`, 2007. 20

[142] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. Apidiff: Detecting api breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 507–511. IEEE, 2018. 20

[143] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. CID: Automating the detection of api-related compatibility issues in Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018. 20

[144] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect API compatibility issues in Android: An empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, 2019. 20

[145] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 501–510. IEEE, 2016. 20

[146] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 702–714, 2016. 20

[147] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)*, pages 1–10. IEEE, 2016.

[148] Marius Kamp, Patrick Kreutzer, and Michael Philippsen. Sesame: A data set of semantically similar Java methods. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 529–533. IEEE, 2019. 20

[149] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 520–524. IEEE, 2015. 21

[150] Raula Gaikovina Kulaa, Coen De Rooverb, Daniel M Germanc, Takashi Ishiob, and Katsuro Inouea. Modeling library dependencies and updates in large super repository universes.

[151] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136. IEEE, 2014.

[152] Fabio Massacci and Ivan Pashchenko. Technical leverage in a software ecosystem: Development opportunities and security risks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1386–1397. IEEE, 2021.

[153] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Dismal code: Studying the evolution of security bugs. In *LASER 2013 (LASER 2013)*, pages 37–48, 2013.

[154] Sultan S Alqahtani, Ellis E Eghan, and Juergen Rilling. Sv-af—a security vulnerability analysis framework. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 219–229. IEEE, 2016. 21

[155] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 2023 31th acm sigsoft international symposium on foundations of software engineering*, 2023. 21

[156] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on third-party libraries in Android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering*, 2021.

[157] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 270–282. IEEE, 2023. 21

[158] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359. IEEE, 2019. 21, 82

[159] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 319–330, 2018. 21

[160] Yueming Wu, Chengwei Liu, Zhengzi Xu, Lyuye Zhang, Yiran Zhang, Zhiling Zhu, and Yang Liu. The software genome project: Unraveling software through genetic principles. 2024. 22

[161] Chenghao Li, Yifei Wu, Wenbo Shen, Zichen Zhao, Rui Chang, Chengwei Liu, Yang Liu, and Kui Ren. Demystifying compiler unstable feature usage and impacts in the rust ecosystem. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[162] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. Demystifying the composition and code reuse in solidity smart contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 796–807, 2023.

[163] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023.

[164] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 408–419, 2018.

[165] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. A large-scale empirical study on industrial fake apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 183–192. IEEE, 2019. 22

[166] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 28(3):59, 2023. 22

[167] David Reid, Mahmoud Jahanshahi, and Audris Mockus. The extent of orphan vulnerabilities from code reuse in open source software. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2104–2115, 2022. 22

[168] Brian Fitzgerald, Audris Mockus, and Minghui Zhou. *Towards Engineering Free/Libre Open Source Software (FLOSS) Ecosystems for Impact and Sustainability: Communications of NII Shonan Meetings*. Springer, 2019. 22

[169] Xin Tan, Yuan Zhang, Jiajun Cao, Kun Sun, Mi Zhang, and Min Yang. Understanding the practice of security patch management across multiple branches in oss projects. In *Proceedings of the ACM Web Conference 2022*, pages 767–777, 2022. 22

[170] Meiqiu Xu, Ying Wang, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. Insight: Exploring cross-ecosystem vulnerability impacts. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022. 22

[171] Muhammad Shahzad, M Zubair Shafiq, and Alex X Liu. Large scale characterization of software vulnerability life cycles. *IEEE Transactions on Dependable and Secure Computing*, 17(4):730–744, 2019. 22

[172] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. Hero: On the chaos when path meets modules. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 99–111. IEEE, 2021. 23

[173] Filipe R Cogo, Gustavo A Oliva, Cor-Paul Bezemer, and Ahmed E Hassan. An empirical study of same-day releases of popular packages in the npm ecosystem. *Empirical Software Engineering*, 26(5):89, 2021. 23

[174] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:384–417, 2018. 23

[175] 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 *USENIX security symposium*, volume 17, 2019. 23

[176] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 2022. 23

[177] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–414. IEEE, 2018. 23

[178] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 26: 1–28, 2021. 23

[179] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 228–237. IEEE, 2020.

[180] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. A formal framework for measuring technical lag in component repositories—and its application to npm. *Journal of Software: Evolution and Process*, 31(8):e2157, 2019.

[181] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[182] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017. 23

[183] Http-core motivating example. `https://issues.apache.org/jira/browse/HTTPCORE-367`, 2013. 26

[184] Http-core. `https://hc.apache.org/httpcomponents-core-4.4.x/index.html`, 2021. 26

[185] Bcel. `https://commons.apache.org/proper/commons-bcel`, 2021. 27

[186] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010. 27, 32, 61

[187] Refactoring. `https://en.wikipedia.org/wiki/Code_refactoring`, 2021. 28

[188] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180: 1–15, 2019. 28, 33

[189] Java Polymorphism. `https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html`, 2023. 28

[190] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 97–106, 2017. 29

[191] Data set. `https://sites.google.com/view/ase23maven`, 2023. 31, 43, 70, 82, 123, 125

[192] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992. 31

[193] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using s park. In *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12*, pages 153–169. Springer, 2003. 32

[194] Jimple. `https://en.wikipedia.org/wiki/Soot_(software)#Jimple`, 2012. 32

[195] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 2010. 32, 62, 83

[196] Eduardo Cunha Campos and Marcelo de Almeida Maia. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 404–413. IEEE, 2017. 36

[197] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. 36

[198] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011. 38, 39

[199] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–942. IEEE, 2020. 38, 39

[200] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V Sichi. Jgrapht—a java library for graph data structures and algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 46(2):1–29, 2020. 41

[201] Dijkstra Algorithm. `https://en.wikipedia.org/wiki/Dijkstra_algorithm`, 2021. 41

[202] Httpclient. `https://hc.apache.org/httpcomponents-client-5.1.x/`, 2021. 43

[203] Java Reflection. `https://www.oracle.com/technical-resources/articles/java/javareflection.html`, 2021. 44

[204] Java 8. `https://www.oracle.com/java/technologies/java8.html`, 2021. 45

[205] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019. 45

[206] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to Java program analysis with Soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, 17, 2008. 50

[207] Symbolic Execution. `https://en.wikipedia.org/wiki/Symbolic_execution`, 2021. 51

[208] Unit Testing. `https://en.wikipedia.org/wiki/Unit_testing`, 2021. 51

[209] Optimization problem. `https://en.wikipedia.org/wiki/Optimization_problem`, 2022. 53

[210] Commons-lang. `https://github.com/apache/commons-lang`, 2023. 54

[211] Dependabot upgrade resulted in build failure. https://github.com/apache/commons-lang/pull/826, 2021. 54

[212] Wala. https://github.com/wala/WALA, 2023. 56

[213] Common Vulnerability Scoring System. https://nvd.nist.gov/vuln-metrics/cvss, 2023. 59, 132

[214] Maven Scope. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope, 2023. 61

[215] Soot Spark Call Graph. https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm#phase_5_2, 2021. 61

[216] Uber-jar. https://imagej.net/develop/uber-jars, 2023. 61

[217] Z3 Solver. https://github.com/Z3Prover/z3, 2023. 62

[218] Multi-objective optimization. https://en.wikipedia.org/wiki/Multi-objective_optimization, 2022. 63

[219] japicmp. https://siom79.github.io/japicmp/, 2023. 63

[220] Maven Versions. https://maven.apache.org/pom.html, 2023. 63, 65, 94

[221] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. Detecting and augmenting missing key aspects in vulnerability descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–27, 2022. 68

[222] Hao Guo, Zhenchang Xing, Sen Chen, Xiaohong Li, Yude Bai, and Hu Zhang. Key aspects augmentation of vulnerability description based on multiple security databases. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1020–1025. IEEE, 2021. 68

[223] Common Platform Enumeration. https://nvd.nist.gov/Products/CPE, 2023. 70

[224] Apollo project. https://github.com/ApolloAuto/apollo, 2023. 72

[225] Maven Version ranges. https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, 2023. 82, 92

[226] Maven Soft Version Constraint. https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, 2023. 82

[227] Google Open-source Database. https://docs.deps.dev/bigquery/v1, 2023. 82

[228] Gradle Dependency Constraint. https://docs.gradle.org/current/userguide/dependency_constraints.html, 2023. 92

[229] Ivy Dependency Override. https://ant.apache.org/ivy/history/2.3.0/ivyfile/dependencies.html, 2023. 92

[230] Oracle Java Compatibility Documentation. https://www.oracle.com/java/technologies/javase/8-compatibility-guide.html, 2023. 97

[231] Soot spark. https://www.sable.mcgill.ca/soot/doc/soot/options/SparkOptions.html, 2023. 98

[232] Caret Ranges. https://docs.npmjs.com/cli/v6/using-npm/semver#caret-ranges-123-025-004, 2023. 102

[233] Pypi. https://pypi.org/, 2024. 103

[234] Go modules reference - modules, 2023. URL https://go.dev/ref/mod#modules-overview. (Accessed on 03/25/2023). 105

[235] Go modules reference - pseudo-versions, 2023. URL https://go.dev/ref/mod#pseudo-versions. (Accessed on 03/24/2023). 106, 122

[236] Snyk — developer security — develop fast. stay secure. — snyk, 2023. URL https://snyk.io/. (Accessed on 03/29/2023). 108, 113

[237] Nvd - home, 2023. URL https://nvd.nist.gov/. (Accessed on 03/29/2023). 108, 113

[238] Quartile - wikipedia, 2023. URL https://en.wikipedia.org/wiki/Quartile. (Accessed on 07/09/2023). 110

[239] https://index.golang.org/index?since=, 2023. URL https://index.golang.org/index?since=. (Accessed on 03/29/2023). 110

[240] Libraries.io, 2023. URL https://libraries.io/. 113

[241] Go modules reference - versions, 2023. URL https://go.dev/ref/mod#versions. (Accessed on 07/29/2023). 126

[242] Understanding dependency management in go. https://lucasfcosta.com/2017/02/07/Understanding-Go-Dependency-Management.html, 2023. (Accessed on 11/17/2023). 127

[243] Semantic version tags in go.mod file — by shivam rathore — the startup — medium. https://medium.com/swlh/semantic-version-tags-in-go-mod-file-f6ad903a972d, 2023. (Accessed on 11/17/2023). 128

[244] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 1–28, 2017. 132